An Investigation into Patterns and Anti-Patterns in Search based Refactoring

K. Sears

 BSc Computer Science

 January 1, 2016

*Abstract*

Contents

[1 Background 4](#_Toc445225914)

[1.1 Model Driven Approaches and UML 5](#_Toc445225915)

[1.2 Search Based Software Engineering 6](#_Toc445225916)

[1.3 Interactive Optimization 6](#_Toc445225917)

[1.4 Multi-objective Optimization 7](#_Toc445225918)

[1.5 Co-Evolution 8](#_Toc445225919)

[1.6 Test Suites 8](#_Toc445225920)

[1.7 Design Patterns 13](#_Toc445225921)

[1.8 Search Based Software engineering Focusing on Design 19](#_Toc445225922)

[1.9 GRASP Patterns (General Responsibility Assignment Software Patterns) 20](#_Toc445225923)

[1.10 SBSE of UML 22](#_Toc445225924)

[1.11 Mutation Operators 24](#_Toc445225925)

[1.12 Crossover Operators 25](#_Toc445225926)

[2 Project Scope 25](#_Toc445225927)

[2.1 Hypothesis 25](#_Toc445225928)

[2.2 Objectives 26](#_Toc445225929)

[2.3 Success criteria 26](#_Toc445225930)

[3 Methodology 27](#_Toc445225931)

[4 Requirements Analysis 28](#_Toc445225932)

[4.1 Functional Requirements 28](#_Toc445225933)

[4.1.1 Must have 28](#_Toc445225934)

[4.1.2 Should have 29](#_Toc445225935)

[4.1.3 Could have 29](#_Toc445225936)

[4.1.4 Won't have 29](#_Toc445225937)

[4.2 Non-Functional Requirements 29](#_Toc445225938)

[5 Test Approach 30](#_Toc445225939)

[5.1.1 Test Cases 30](#_Toc445225940)

[5.1.2 Test Cases 1 30](#_Toc445225941)

[6 Design 31](#_Toc445225942)

[6.1 Languages 31](#_Toc445225943)

[6.1.1 XMI 31](#_Toc445225944)

[6.1.2 Java 32](#_Toc445225945)

[6.2 Tools 32](#_Toc445225946)

[6.2.1 Interactive Development Environment 32](#_Toc445225947)

[6.2.2 Repository 32](#_Toc445225948)

[6.2.3 Logging 32](#_Toc445225949)

[6.3 Platform 32](#_Toc445225950)

[6.4 Framework 33](#_Toc445225951)

[6.5 Evolutionary Computation Method 33](#_Toc445225952)

[6.5.1 Selection 33](#_Toc445225953)

[6.5.2 Mutation 33](#_Toc445225954)

[6.5.3 Fitness Metrics 33](#_Toc445225955)

[6.6 Anti-Patterns 33](#_Toc445225956)

[7 Logical Model 37](#_Toc445225957)

[8 Physical model 37](#_Toc445225958)

[8.1 What issues arose during implementation? 37](#_Toc445225959)

[8.2 How will you show what has been achieved? 38](#_Toc445225960)

[8.3 Has anything not been finished? 38](#_Toc445225961)

[9 Conclusion 38](#_Toc445225962)

[9.1 Do your findings support or refute your claim? 38](#_Toc445225963)

[9.2 Have you met your original project goals? 38](#_Toc445225964)

[9.3 What limitations does your product have? 38](#_Toc445225965)

[9.4 What thoughts do you have now about your product? 38](#_Toc445225966)

[10 Future Work 38](#_Toc445225967)

[10.1 How could you improve your product? 38](#_Toc445225968)

[10.2 How could you improve the development process? 38](#_Toc445225969)

[10.3 What would you do differently? 38](#_Toc445225970)

[11 References 38](#_Toc445225971)

# Background

Software engineering in the past decades has evolved to work within the Object Orientated Programming (OOP) paradigm due to the high level advantages it brings in its flexibility to redesign and repurpose. OOP modularizes code into class “blueprints” which capture state related attributes and closely related behaviors which act upon data known as methods. These classes are actualized into objects whose interaction with one another brings about the functionality of the software and dictate the overall architectural design of the software. However, as the complexity of software grows with each additional functionality the maintenance of classes becomes exponentially more challenging as the amount of work dedicated to restructuring software architectural design, a process known as refactoring, rapidly increases. In order to tackle this issue of complexity putting ever more demands on developers, namely architectural software engineers, the use of Artificial Intelligence techniques such as Evolutionary Computation (EC) (Eiben and Smith, 2015) have been applied with varying amounts of success to assist and automate refactoring.

It seems the use of AI will become ever more necessary as demand for interoperability (systems of systems) for developing advancements such as multi-agent systems and big data becomes more prevalent in the software industry. These more recent developments act to further multiply the complexity of additional changes and demands placed upon the software to unprecedented heights.

Approaches of EC, however, require refining and expanding to adapt to these ever growing workloads in order to converge more effectively on well-formed and robust architectures that still retain the flexibility OOP originally brought.

The design phase in the software lifecycle constitutes one of the earliest stages in software engineering. Problems addressed at this point can significantly reduce issues downstream and reduce the amount of refactoring often needed after implementation (Fowler, 1999). Some success with EC search has been shown to provide relief on the workload faced by developers during this phase (Harman et al, 2014), (Arcuri et al, 2008), (Bowman et al, 2010), (Brosch, et al, 2009). It would be reasonable to assume then that efforts focused on this phase of the software lifecycle would be more advantageous than subsequent phases to reduce workload and reliance on software optimization after its implementation.

## Model Driven Approaches and UML

Modeling software abstractly by dissemination and representation of business requirements, with use of diagrams and modeling languages, has played a crucial role in the successful design of software. The benefits of Modeling for managing separation of concerns and providing traceability for artefacts in software not only allows distribution of labour between developer workforces concurrently working on a single project, but also acts as documentation for software comprehension for new developers and stakeholders.

The Unified modeling language (UML) is the defacto-standard of design modeling languages maintained by the Object Management Group (OMG), established and recognized as an effective tool across the software industry. Fowler (Fowler, 2004) supports:

“*UML is the successor to the wave of object-oriented analysis and design (OOA&D) methods that appeared in the late '80s and early '90s.*”

Previous research into the application of EC techniques on software design have used UML Class Diagrams, and Sequence Diagrams as their representation to perform manipulation on software architectures.

Extensions to UML such as Role-Based Modeling language and Object Constraint Language also exist, however, due to this area of research being in its infancy there are limited examples of their success. Still, their significance will be later elaborated on in this paper.

## Search Based Software Engineering

The refactoring of software design is one of many specific areas of focus in the overall field of search based software engineering. Harman (Harman, 2012) one of the leading experts in the field of Search Based Software Engineering defines it in the following way:

*“The aim of Search Based Software Engineering (SBSE) is to move software engineering problems from human-based search to machine-based search, using a variety of techniques from the metaheuristic search, operations research and evolutionary computation paradigms.”*

To give an idea of the scope of SBSE, it aims to address several stages of the software lifecycle, these being: requirements, design, testing, refactoring, project management, maintenance and reverse engineering. According to a Survey by Harman (Harman, 2009):

“*54% of the overall SBSE literature is concerned with [software engineering] applications relating to testing.*”

The area of design has been largely overlooked as it accounts for only 10% of the entire field.

To be clear on the term “Search” here it is defined in terms of Evolutionary Computation Search Methods. These techniques can be broken into three distinct categories: Optimization, Modelling and Simulation (Eiben and Smith, 2015). Although the term modelling lends itself to be the concern of our investigation, optimization is actually the driving force behind the refactoring of software during its design, as it aims to find the best possible architecture in order to produce optimal, or near optimal measures of quality for a piece of software’s non-functional requirements, which will be discussed in greater detail later.

The reason why EC is used over exhaustive search is due to the nature of the problem belonging to a set of problems known as NP-Hard problems. These are problems with an exponential time complexity, whose solutions can only be checked also in non-polynomial time. The search space for the software design refactoring problem is the set of possible combinations of attributes and methods distributed over a number of classes, arranged in such a way as to provide the best quality of architecture based on OOP principles. It is in this way that adding an additional attribute, method or class produces a new set of combinations where this new component may be inserted into the design, each producing a differing quality of architecture dependent the dependencies to other components the new addition may have.

For Example, if we have a design for a Library program which monitors the withdrawal and deposit of books for library members, then the design would include at minimum: classes for member, book, and library; methods to add/delete books and members; and a data structure for storing an index of all available books and members.

Adding the additional functionality of the library providing audio-cassette books, would then include the addition of another class for audiobooks. There would be another data structure for these, else are they added to the book index and considered the same as books? There would be needed additional methods to add and remove these audiobooks, perhaps only certain members may access them?

It is clear from this trivial example that changes to design often require a certain amount of refactoring to the software to accommodate them. Good architectural design will require less changes, however certain requirements will inevitably affect a broad range of components resulting in a large number of changes. It is these kinds of additional functionalities that require complete reworks that are difficult for developers to cater for and which EC is adapt at searching for. Still, certain solutions found by EC often make little inherent sense to designers from the semantic view of the business domain logic, and so guidance for these techniques are necessary for identifying promising solutions from those which, although meet OOP quality metrics which guide the EC, do not provide a robust and sensible architecture. For this, more sophisticated approaches to random search are necessary.

Vlissides (Vlissides et al., 1995) supports this by stating,

“Factors such as encapsulation, granularity, Dependency, flexibility, performance, evolution, reusability can be difficult when considering the decomposition of objects when deciding what is inappropriate architecture for a particular application.”

## GRASP Patterns

GRASP Patterns (General Responsibility Assignment Software Patterns) (Larman, 2005) are one approach to providing guidance when refactoring. They outline practices which adhere to the concepts of best practice in OOP and should be taken into account when performing refactoring to improve non-functional requirements, such as the ease of introducing additional functionality discussed in the example.

GRASP Patterns include creation of classes for the delegation of concerns such as Controller Classes which act as interfaces into the system built around usecases. To link to our example, a controller class may provide an intermediary class between the user interface and the business logic which controls the underlying mechanism for managing books in a database. This way a change in which database technology used could remain isolated from the interface which the user is accustomed to, making migration more streamlined. Creator classes (Factory Pattern) are another way of achieving this, delegating the concern of Object creation at runtime to avoid tight coupling between classes. Abstract classes promote polymorphism and to inherit common behaviors between classes from a shared superclass. And finally protected Variations which are a combination of interfaces and polymorphism in order to isolate changes in one subsystem from other subsystems assist in developing architecture which remains robust yet flexible.

Introducing classes with no real world counterparts but improve composition, Vlissides states, does however add additional complexities to the refactoring process.

Another important point Vlissides reasons is that “Object composition is favored over inheritance”. This will be important to consider as the less tightly coupled the software is the more useful to the developer it will be.

Composition does however have the disadvantage of making code less intuitive to a developer as traceability of realizations becomes more challenging. So changes such as this will need to be documented in some way for the developer if they have been implemented by EC.

## Design Patterns

“It would be natural to suppose that work on design patterns could and should form a foundation for a strand of work on SBSE for design” (Harman et al., 2014).

Vlissides, J., Helm, R., Johnson, R. and Gamma, E known commonly in the field as the “Gang Of Four” outline descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context of OO-design called Design Patterns. They provide a succinct way of categorizing the kinds of changes described in GRASP patterns and are already familiar to OOP developers and so may bridge the gap in understanding more complex changes made to software design on a high level. It is worth noting that architectural styles (Shaw and Garlan, 1996) e.g. message dispatcher and client-server, are separate from design patterns e.g. Façade, Mediator, Strategy, Adapter, etc. according to Räihä (Räihä, 2011), their scope in terms of scale differ, with architectural styles addressing more the overall system behaviour rather than interaction between components within the system, i.e. design patterns are a subset of the Object Orientated Organization subcategory in architectural styles.

Design patterns during the design phase of SBSE have been applied by Amoui, R¨aih¨a and El-Sharqwi among others.

(Amoui et al, 2006) use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. Their goal is to find the best sequence of transformations, i.e., pattern implementations. Best chromosomes are evolved so that abstract packages become more abstract and concrete packages, in turn, become more concrete.

R¨aih¨a et al. [R¨aih¨a, O., Koskimies, K., and M¨akinen, E. Genetic synthesis of software architecture. In Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL08), pages 565–574. Springer, 2008.] have taken the design of software architecture a step further than Simons and Parmee [Simons, C. L. and Parmee, I. C. A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design. Engineering Optimization, 39(5):631–648, 2007 ///////////////// Simons, C. L. and Parmee, I. C. Single and multi-objective genetic operators in object-oriented conceptual software design. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007), pages 1957–1958. ACM, 2007] by starting the design from a responsibility dependency graph. The dependency graph can also be achieved from use cases, but the architecture is developed further than the class distribution of actions and data.

R¨aih¨a et al. [R¨aih¨a, O., Koskimies, K., M¨akinen, E., and Syst¨a, T. Pattern-based genetic model refinements in MDA. Nordic Journal of Computing, 14(4):322–339, 2008.] have also applied GAs in model transformations that can be understood as pattern-based refinements. In MDA, such transformations can be exploited for deriving a Platform Independent Model from a Computationally Independent Model. The approach uses design patterns as the basis of mutations and exploits various quality metrics for deriving a fitness function. They give a genetic representation of models and propose transformations for them. The results suggest that GAs provide a feasible vehicle for model transformations, leading to convergent and reasonably fast transformation process. R¨aih¨a et al. [R¨aih¨a, O., Koskimies, K., and M¨akinen, E. Scenario-based genetic synthesis of software architecture. In Proceedings of the 4th International Conference on Software Engineering Advances (ICSEA09), pages 437–445. IEEE, 2009.] have also later on added scenarios, which are common in real world architecture evaluations, to evaluate the fitness of their synthesized architectures

El-Sharqwi et al [M. El-Sharqwi, H. Mahdi and I. El-Madah, "Pattern-based model refactoring," in Computer Engineering and Systems (ICCES), 2010 International Conference on, 2010, pp. 301-306.] present an approach to apply model refactoring based on design patterns that are defined in XML notation. A design pattern consists of three parts: a Problem Specification describing the context where the design pattern can be applied to improve some quality aspect, a Target Specification describing the design pattern itself, and a Transformation Specification describing a sequence of primitive transformations required to apply the design pattern.

Jensen and Cheng [Jensen, A. C. and Cheng, B. H. C. On the use of genetic programming for automated refactoring and the introduction of design patterns. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2010), pages 1341–1348. ACM, 2010] produced a tool they named “RE-MODEL” which uses genetic programming (GP) for the refactoring UML class diagrams, and aims to introduce design patterns into their architecture in order to “reduce the cognitive labour for software engineers by suggesting design changes that are tailored to the software design in question” which correlates closely to this investigation. The input models are encoded in tree form, where each node is an attribute or operation. A tree form is emphasised as a better approach over iterative refactoring as these do “not support the composition and interaction of multiple design changes in order to construct design artifacts such as design patterns”, sequences of transformations on nodes, they suggest however, can.

Other than the refactored candidate solution decoded as output, the program provides the transformations which were performed. Providing traceability for the user which is often overlooked in this field.

The test cases used were from a Web-based software system known as ReMoDD and apply Remodel to a set of 58 published software designs each comprising 5 to 15 classes. Each test comprised 100 generations with a tournament selection of size 7. A total of 3,103 new design pattern instances evolved in the 290 runs (58 software designs and five runs per design) for an average of 10.70 instances per design. These initial results demonstrate that Remodel is capable of introducing design patterns in a diverse set of software designs.

An average of 12 new design pattern instances evolved in the best individuals, thus providing plenty of suggestions for a software engineer who might be considering the introduction of design patterns to make the ReMoDD design more extensible or maintainable.

However, this broad set of suggestions also highlights the need for human review and demonstrates that this automated approach should not be blindly followed.

Introduced classes suffer from unintuitive names due to lack of natural language processor making understanding for the developer more complex.

CONCLUSIONS

In this paper, we presented Remodel, an approach for automated refactoring of software designs that combines genetic programming, software engineering metrics, and minitransformations to introduce design patterns in existing software designs. Specifically, we showed that the minitransformations support composition of multiple design changes, thus enabling the generation of richer refactoring strategies than were previously possible. We conducted a set of four experiments to determine the optimal coefficients for the terms in our fitness function. Finally, we applied Remodel to a large, Web-based software system. Our results show that the approach is capable of simultaneously improving the quality of a software design with respect to metrics as well as automatically introducing design pattern instances, a combination that was not previously considered. Our ongoing research considers the use of different metrics, domain specific design patterns, and design change mechanisms with different levels of abstraction to explore the impact of these factors, as well as the modularity of Remodel, on different refactoring problems.

This approach is close to those of R¨aih¨a et al. [R¨aih¨a, O., Koskimies, K., and M¨akinen, E. Genetic synthesis of software architecture. In Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL08), pages 565–574. Springer, 2008.] and the approach used here, the difference being that Jensen and Cheng have clearly a refactoring point of view. The simple existence of a pattern is not a reason for reward itself in the fitness function (Sievi-Korte, O., Mäkinen, E. and Poranen, T., 2013. Simulated Annealing for Aiding Genetic Algorithm in Software Architecture Synthesis. Acta Cybern.,21(2), pp.235-265.)

Cortellessa, Vittorio, Antinisca Di Marco, Romina Eramo, Alfonso Pierantonio, and Catia Trubiani. "Digging into UML models to remove performance antipatterns." In Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems, pp. 9-16. ACM, 2010.

## Search Based Software engineering Focusing on Design

Räihä (Räihä, 2010) published a Survey which outlines the design processes involved in SBSE object-oriented systems. The survey describes the simplest conception of SBSE design as focusing on extracting concepts from business type models such as use cases and user stories and deriving them into methods and attributes, which are then distributed into classes, the latter referred to as Class responsibility assignment (CRA) by Bowman (Bowman et al, 2010).

Bowman suggests there is a difference between the CRA problem and refactoring software design in that it does not specifically search for patterns in a problem-driven manner, and although refactoring may change class responsibilities it is ultimately less concerned with fitness metrics, which will be further discussed in later chapters, and emphases more on the later ease of extensibility of the software. In this investigation we make no distinction between CRA and refactoring, viewing them as intrinsically linked as CRA gives rise to the emergence of patterns and anti-patterns, which are merely the formally defined frameworks from which non-functional requirements can be discerned from. Räihä does state that “Redesigning software architectures automatically is slightly easier than building architecture from the very beginning, as the initial model already exists, and it merely needs to be ameliorated”, and although a distinction can be made from generating models from requirements, both Bowman’s study and this investigation focus on the improvement through endogenous transformations of ready designed class diagrams taken from UML. Therefore from now, we shall refer to CRA and refactoring of design models with use of design patterns or anti-patterns as simply refactoring.

Bowman (Bowman et al, 2010) states “Not only is this (refactoring) vital during initial analysis/design phases, but also during maintenance, when new responsibilities have to be assigned to (new) classes or existing responsibilities have to be changed (e.g., moved to other classes)”. Supporting in turn the significance of developing automated methods of refactoring in reducing the developer’s workload in multiple phases of the software lifecycle.

Bowman’s work uses a Pareto based multi-objective Genetic Algorithm, a type of EC search technique, to provide decision aid to improve early object-oriented analysis and design models. The fact they are using EC to manipulate models rather than source code allows us to see how previous attempts that relate to our investigation have been accomplished.

Simons et al, 2010 make use of a Genetic Algorithm for interactive AI which serves to assist developers in the upstream development of system architecture through transformations on Class Diagrams. Candidate solutions are the encoded attributes and methods derived from use cases allocated consecutively through surjections and partitions respectively to classes denoted by end of class “markers” to discretely represented genomes. Genetic Operators isolate discrete zones from the search space

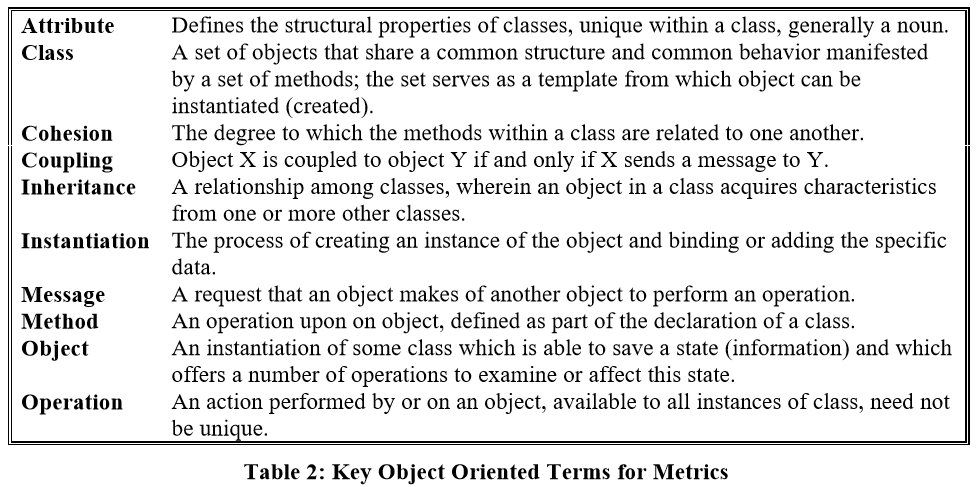
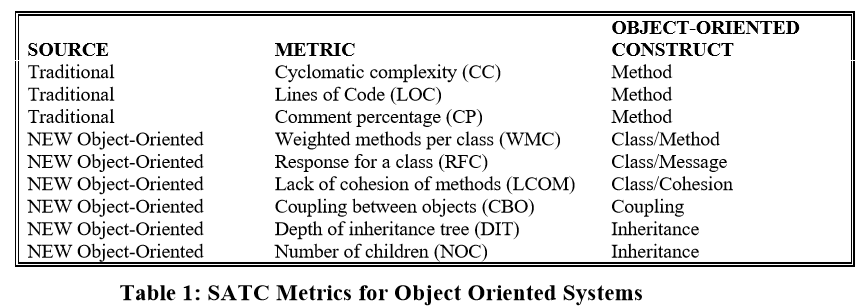
## SBSE of UML

A survey in the support of evolution of UML models conducted by (Khalil and Dingel, 2013) outlines the three primary types of evolution outlined by E. Swanson (Swanson, 1976), these being corrective, adaptive and perfective which respectively repair faults in design, add further extensions or repurpose design and finally optimize design, and discusses four common tasks identified in model evolution.

The first, Change Impact analysis, whereby test suites which measure fitness metrics of the model calculate the basis for change as well as selection for population based evolution.

## Test Suites

[Title: Applying and Interpreting Object Oriented Metrics Presenter: Dr. Linda H. Rosenberg Track: Track 7 - Measures/Metrics Day: Wednesday Keywords: Metrics, Object-Oriented] Presents in a succinct way some of the key metrics when Object Orientated systems were still in their infancy outlined below along with some of the previously successful ways in which procedural programming measured efficiency of design.



[O'Keeffe M. and 0 Cinneide M., "Towards design improvement through combinatorial optimization," Proc. Workshop on Directions in Software Engineering Environments, 2004.] and [Seng I., Stammel J. and Burkhard D., "Search-based determination of refactorings for improving the class structure of object-oriented systems," Proc. Conf. on Genetic and Evolutionary Computation, 2006.] both use Sum of Weighted Objectives, not multi-Objective measures of fitness. Although work has been done on multi-objective SAs there is less evidence of its application to GAs which [Multi-Objective Genetic Algorithm to Support Class Responsibility Assignment; Bowman, Briand, Labiche] demonstrates. Bowman’s work highly relates to our topic as he also aims on improving early OOAD models. But does not focus on problem driven processes in the way of identifying and converting anti-patterns to patterns, instead he centers on class responsibility assignment making it a more general approach to the problem, with this said the fitness metrics applied are still relevant to the problem domain and so will be discussed.

Bowman uses dependencies among methods and attributes to calculate the coupling and cohesion between classes. This information is drawn from Object Constraint Language as well as other UML Diagrams, i.e. sequence diagrams, in order to identify what attributes and association ends can be accessed by which methods. From this three measures of coupling are used from [Briand L. C., Daly J. and Wuest J., "A Unified Framework for Coupling Measurement in Object-Oriented Systems," IEEE TSE, 25 (1), pp. 91-121, 1999.], these being: Method-Attribute Coupling, Method-Method Coupling and Method-Generalization Coupling. And Two measures of cohesion taken from [Briand L. C., Daly J. and Wuest J., "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," Empirical Software Engineering, 3 (1), pp. 65-117, 1998.], which are: Tight Class Cohesion and Ratio if Cohesive Interactions. TODO evaluate these metrics

The metrics evaluate the object oriented concepts: methods, classes, coupling, and inheritance.

The metrics focus on internal object structure that reflects the complexity of each individual entity and on external complexity that measures the interactions among entities. The metrics measure computational complexity that affects the efficiency of an algorithm and the use of machine resources, as well as psychological complexity factors that affect the ability of a programmer to create, comprehend, modify, and maintain software.

Simons et al use Cohesion of Methods (Harrison et al) and Coupling between Objects (Briand et al) as fitness metrics for their Genetic Algorithm. They note the limitation of Cohesion of Methods in its inability to distinguish the number of methods in a given class and compensate by using a multiplier, so that classes with limited methods do not converge on high or low scores due to their limited scope. For instance results normalize between ranges 0.0 and 1.0, if a class has only a single method which accesses an attribute within the same class, its score would equal 1.0 for cohesiveness.

Coupling between Objects is taken as the measure of dependencies between classes if a given method of a class has a dependency on an attribute from another class. Again it’s limitation of not providing the strength of the coupling between two given classes (in terms of the number of methods in class x using attributes in class x) is addressed by first determining the set of all methods the first source class has which are dependent on the second target class and then dividing this value by the set of all uses of methods dependent on attributes.

Both these metrics provide a fair basis for determining coupling and cohesion, but undermine other ways of calculating fitness such as those previously mentioned in this chapter. Although the paper is focused on interactive design with user feedback being the predominant driving force for evolution and not multi-objective global search, which is understandable as often metrics conflict and the discovery of novel useful designs which work are more beneficial to the designer than those taken from a search spaced narrowed by presumptions of optimal architectures made by aggregated metrics.

#### QMOOD

QMOOD [Bansiya, J. and Davis, C. G. A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering, 28(1):4–17, 2002] O’Keeffe and ´O Cinn´eide [M. O’Keeffe and M. ´ O. Cinn´eide. A stochastic approach to automated design improvement. In Proceedings of the 2nd international conference on Principles and practice of programming in Java, pages 59–62. Computer Science Press, Inc. New York, NY, USA, 2003.

M. O’Keeffe and M. ´O Cinn´eide. Search-based refactoring: an empirical study. Journal of Software Maintenance and Evolution: Research and Practice, 20(5), 2008.] also use this tool. 11 individual metrics.

[Jensen, A. C. and Cheng, B. H. C. On the use of genetic programming for automated refactoring and the introduction of design patterns. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2010), pages 1341–1348. ACM, 2010] RE-MODEL’s fitness function uses the QMOOD [Bansiya, J. and Davis, C. G. A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering, 28(1):4–17, 2002] metrics suite to base their selection method on, the reasoning for its selection was based on its rich selection of metrics, as well as the appeal in its prior use by (M. O’Keeffe and M. ´O Cinn´eide. Search-based refactoring: an empirical study. Journal of Software Maintenance and Evolution: Research and Practice, 20(5), 2008). In addition to QMOOD a penalty based on the number of used mini-transformations and reward the existence of (any) design patterns is given.

## Multi-objective Optimization

“SE problems are typically multi-objective problems” (Harman, 2009) as fitness metrics for EC tend to conflict with one another. Finding a balance between multiple objectives which suffer from this difficulty can often be resolved through the use of Pareto optimality, where by each candidate solution is no worse than any of the others in the set of best solutions measured by one metric or another, but also cannot be said to be better. This set of best solutions with one or more optimal metrics is referred to as a Pareto-front. And can be used in conjunction with Interactive Optimization to provide developers with a list of options from which to choose the most applicable semantically to the business domains priorities.

The second, Change Propagation, ensures that inter-related model elements are synchronized following changes applied to them, both vertically as well as horizontally. Vertical changes, for example, would be Sequence Diagrams in UML which describe interaction between classes changing if transformations on the Class Diagram were to migrate methods to other classes. Horizontal changes would be to ensure dependencies between classes depicted in a class diagram were to change under the same circumstances.

The third, consistency management, uses rules which govern the relationships between parts of the model to ensure assertions made on them do not conflict. In the context of design this applies to redundancies and clashes in model diagrams, and in particular to our area of interest common software design standards, i.e. design patterns.

And the forth, Uncertainty Management, which addresses incomplete or inconsistent requirements which lead to false assumptions in the modeling of the software, which may arise from lack of knowledge about the problem domain or conflicting stakeholder opinions.

## Interactive Optimization

Evolutionary computation which utilizes Evolutionary Operators that incorporate human judgement when conducting search are known as forms of interactive optimization.

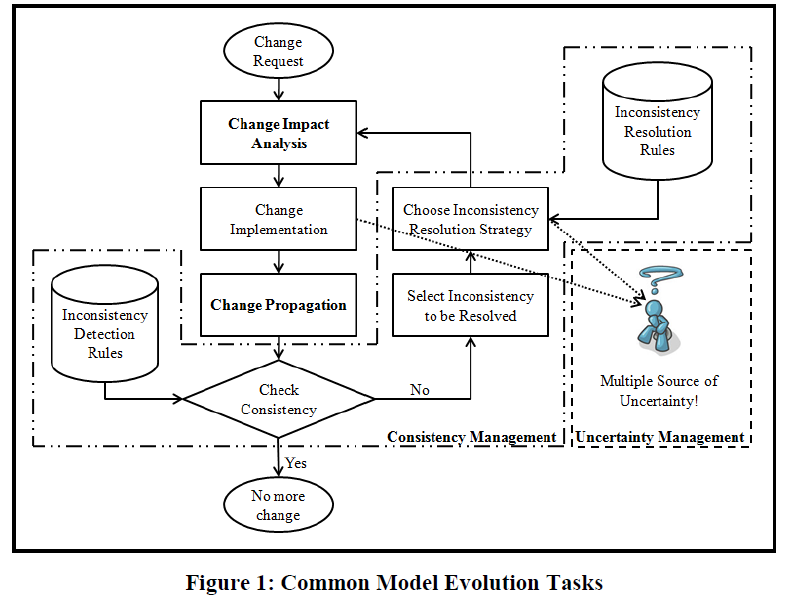
Intuitive value judgements about design preferences made by the user can significantly reduce the computational work made during search (Simons and Parmee 2008), for instance in place of self-adaptive parameter control (Eiben and Smith, 2003) which can be computationally expensive.

During software design often minor details can be missed or terms defined subjectively, Interactive optimization is particularly suited to these kinds of scenarios where feedback from the designer can prevent complications further along the production line from these kinds of ambiguities.

Harman suggests it may also be possible to use a search based approach to explore the implicit design constraints and desirable features by making assumptions in the human assessment of solutions. The drawback to Interactive Optimization is the fatigue and learning-effect bias suffered from frequent referral to the user. “If this fatigue problem can be overcome in the SE domain (as it has in other application domains) then interactive optimization offers great potential benefits to SBSE.”

An example application of this branch of Evolutionary computation was used by Brosch (Brosch et al, 2005) who demonstrated that recordings of refactoring operations acted on a model by users could be used to guide the automation of model transformations.

These four tasks are best depicted in the flow diagram produced by Khalil below.



Design patterns (Gamma et al 1995) have previously proven to be an invaluable asset in software engineering by establishing standardized templates of OOP architectural design which have been repeatedly successful when applied appropriately within industry.

Anti-patterns (Brown et al 1998) take this a step further in that they provide a way of identifying when patterns have not been applied and poor practice has diminished the advantages of OOP. Anti-patterns also provide refactoring steps to manipulate software into appropriate design patterns based on the properties of the software the poor practice has resulted in. Additionally potential consequences of applying such patterns are outlined to avoid inappropriate application of the wrong pattern to the problem domain.

Being able to use anti-patterns in the context of automated design is a key component to justify refactoring steps to the user which the AI has made. Anti-patterns may also reduce the search space in which candidate solutions are sought by reducing “false-positive” solutions that do not effectively reduce developer workload by their design.

## Mutation Operators

[Jensen, A. C. and Cheng, B. H. C. On the use of genetic programming for automated refactoring and the introduction of design patterns. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2010), pages 1341–1348. ACM, 2010] Genetic Program for their RE-MODEL tool uses node ´O Cinn´eide’s six minitransformations: Abstraction. Abstract Access, Delegation, Encapsulate Construction, PartialAbstraction, Wrapper.

## Crossover Operators

[Jensen, A. C. and Cheng, B. H. C. On the use of genetic programming for automated refactoring and the introduction of design patterns. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2010), pages 1341–1348. ACM, 2010] Genetic Program for their RE-MODEL tool uses crossover is applied as exchanging subtrees between selected candidates where non-leaf nodes are selected 90% of the time and leaf nodes are selected the remaining 10% of the time.

# Project Scope

## Hypothesis

It is hypothesized that anti-patterns accelerate beneficial endogenous transformations when refactoring Class Diagrams operated on by a Genetic Algorithm. This will be quantitatively measured through these transformations to reduce coupling and increase cohesion among classes, as well as through the number of changes required to add additional components to the design.

A critical statistical comparison to evolution by a Genetic Algorithm which does not use anti-pattern to pattern heuristics will be used as a control measure as part of this investigation. By comparing the fitness measurements as well as taking qualitative assessments of overall structure of candidate solution architectures, it will be deduced whether the approach using Anti-Patterns does indeed increase the ease of extensibility of the software design. It is implicitly assumed improving extensibility reduces the workload of the software developer.

Components will be randomly allocated but logically related to each individual case study used as input into the system. For instance, a bookshop case study which adds the component which allows the ability for stationary to be purchased as well as books.

## Objectives

Objectives of the investigation are as follows:

* Generate evolved models from input models which follow the same logical business requirements, both with and without the use of Anti-patterns to guide evolution operators.
* Evaluate the effectiveness of anti-patterns as a guide for mutation given the metrics that have been selected for fitness, i.e. coupling and cohesion.
* Incorporate the same additional functionality through use of a software component to both Anti-Pattern guided models and non-anti-pattern guided models to quantitatively assess the ease of additional requirements using the number of changes required to implement them as a measure.

## Success criteria

Success criteria will be based on whether:

* The prototype produces candidate solutions which still fulfill the original logical structure of the input case study used, i.e. contains all original methods, attributes and classes.
* The number of iterative generations required to reach notably higher quality fitness metrics is in fact lower for the anti-pattern guided evolution than the non-anti-pattern guided evolution, if at all.
* The number of changes needed to be made to the evolved models is lower for the anti-pattern models when extending an additional component to the design.

# Methodology

Agile methodology will be used when developing the system. Alternatives considered were iterative design in the form of rapid prototyping, waterfall and lean methodologies. Agile was chosen first and foremost as it is renowned for its success in industry. It also maintains the component based paradigm minimizing risk in terms of addressing bugs and changes as requirements analyses evolve, and can produce at least a version of the work within a given time constraint even if it is not the latest release.

Waterfall would have been a preferred alternative, as the time constraints for the project would have allowed a considerable amount of the planning work to be demonstrated to show understanding of the subject topic even if implementation were left unresolved and incomplete. This notion was repelled however as the nature of an investigation would require at least some experimentation in order to draw any kind of conclusions on the stated hypothesis. Therefore, the most appropriate method was to capture the minimum requirements through planning, building, testing and reviewing, before if having time, doing another iteration of a more refined and extended version.

# Requirements Analysis

*“Requirements engineering is the process of discovering, documenting and managing the requirements for a computer-based system. The goal of requirements engineering is to produce a set of system requirements which, as far as possible, is complete, consistent, relevant and reflects what the customer actually wants.”* [Sommerville, I. and Sawyer, P., 1997. Requirements engineering: a good practice guide. John Wiley & Sons, Inc.]

## Functional Requirements

### Must have

The minimum useable subset of requirements isolated from the literature review are that the prototype software must have: a parser to extract the information from the XMI test cases and represent it in memory, a detection system to identify anti-patterns and patterns already present in the models architecture in memory, an encoder to take the model in memory and represent it in a way which allows it to be manipulated by a GA, a GA to initialize a population of encoded representations and process them through a mutation operator, a measure of fitness of representations for the coupling and cohesion metrics as a basis for selection of candidate solutions, a system to decode the GA representations back into a human readable (preferably graphical) form.

These essential requirements produce an intuitive way to break down the project into smaller subsystems which could be worked on individually which aided in discerning where and if a component of the prototype had failed.

### Should have

Requirements which should be part of the project are: a way of parsing the best solutions back into XMI as an output, a Pareto-front selection system to allow the user to select the fittest subset of solutions and get user-feedback to drive the GA toward solutions which are more applicable to the problem domain and business logic that the meta-model input is attempting to address.

### Could have

Requirements which could be part of the project are a wider selection of fitness metrics other than coupling and cohesion to ensure selection processes are more refined.

### Won't have

The program will not change algorithms within the system, and will only focus on the architecture of the design. It will not be able to generate new functionality to the models. It will not be able to merge the functionality of two systems together. It is not designed to rigorously test the implementation of the software architecture as it deals only with conceptual models.

## Non-Functional Requirements

Non-functional requirements such as the extendibility and maintainability of the prototype were conceived during the conceptual design phase of the project. Class types based on Usecases were formulated and had their responsibilities isolated to increase cohesion so that a good distribution of responsibility was delegated in such a way as to promote loose coupling. In particular the use of a Model-View-Controller architecture was decided upon to ensure a separation of concerns among those classes which were responsible for representing data into data structures, displaying data to the user, and manipulating data in a manageable way.

# Test Approach

Test cases for the functional requirements were written out using Microsoft Excel for succinctness and to easily visualize the success of the project throughout its implementation. Columns break down into the subsystem which is being addressed in the prototype, the task that subsystem is solving, any pre or post conditions needed for the task, the input into the subsystem task including outliers and intentionally faulty information, the expected output of the subsystem task, the actual output of the subsystem task, and the colour coordinated success of the subsystem task where green indicates a successful completion of a test, yellow indicates that the test passed but with unexpected output and therefore needs addressing in future, and red indicates a failed test where output was unexpected and does not meet minimal requirements.

This allows a certain amount of documentation as to what needs changing in future and what works with a given amount of robustness already in terms of the prototypes implementation and error checking capabilities.

### Test Cases

### Test Cases 1

The test data used initially was that of a simple Book Shop test case, to ensure that the simplest of models could be refactored to meet minimum requirements, and in a way which was easy to analyse manually to ensure correctness.

# Design

## Languages

### XMI

To perform automated refactoring in the design phase of the software lifecycle UML, as outlined in the Background chapter, is an obvious starting point to provide the medium in which to transform the modeled abstractions of the software architecture.

Extensible Markup Language (XML) Metadata Interchange (XMI) is the file format also maintained by the OMG which captures the UML model structure as well as any meta-model data in XML. XML is a standard for storing data in both a machine and human readable way and has been highly supported for years. Many modeling tools for UML can export Models into XMI files so that models can be exchanged across platforms and applications. Because of this flexibility and others highlighted by Grose (Grose et al, 2002), it was chosen as the most appropriate choice for test data for this investigation to employ.

Other modeling languages considered were role-based modeling language which has been developed from a bottom up design to process design patterns. It is considered a subset of UML however and still relies heavily on parallels drawn from it. Therefore its use would require UML still and unnecessarily increases the workload needed to parse it into a useable form. Its benefits would be its ability to group class components based on their responsibilities within the software, allowing refactoring to mutate designs in such a way as to preserve the cohesion within substructures in the system.

### Java

The Java programming language will be used as it uses OOP which has already had its advantages outlined. It comes with a vast amount of libraries which make the implementation of data structures very simple, and uses inheritance which can be employed to easily extend additional fitness metrics, anti-patterns and parsers for inputs other than XMI.

## Tools

### Interactive Development Environment

Netbeans IDE was used for the implementation due to its convenience of smart code completion tools, efficient code navigation and its versatile integration with third party software which allowed for repository control, message logging and testing frameworks, all of which were exploited in this project.

### Repository

Records of implementation are given by a GitHub repository. Commits were regularly made after each subsystem task was completed following the initial commit after completion of the XMI parser and the overall Class architecture. Code comments also provide description of algorithms so that others can understand sections of code which are not immediately intuitive as to how they work within the system.

### Logging

## Platform

## Framework

Khalil’s survey notes the lack of standard frameworks and effective and reliable tool support for model Evolution.

## Evolutionary Computation Method

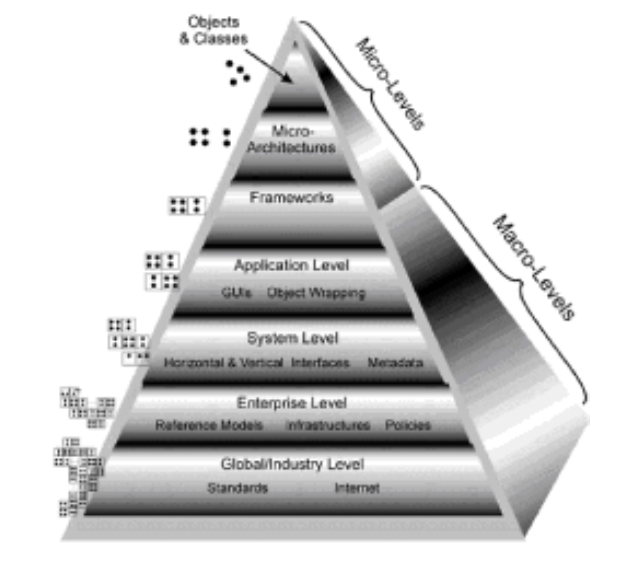
### Selection

### Mutation

### Fitness Metrics

## Anti-Patterns

Being able to implement the conditional statements needed to sufficiently capture the “rules of thumb” provided by (Vlissides et al., 1995), as well as create the database, for anti-patterns and corresponding patterns and be able to provide this information to the mutation operator so that it can apply the necessary transformations, is one of the most challenging aspects of the project. It requires a generous amount of testing for robustness and is not easily broken down into sub-tasks TODO reword : as each anti-pattern requires the feedback from subsequent generations of candidate solutions in order to perform its identification and application of anti-pattern to patterns to the mutation operator.



TODO Talk about scope in relation to “AntiPatterns

Refactoring Software, Architectures, and Projects in Crisis” William J. Brown, Raphael C. Malveau, Hays W. McCormick III, Thomas J. Mowbray

Capturing behaviour / semantics of original models “access”, “update”, and “call” preservation [T. Mens, N. Van Eetvelde, S. Demeyer and D. Janssens, "Formalizing refactorings with graph transformations," Journal of Software Maintenance and Evolution: Research and Practice, vol. 17, pp. 247-276, 2005.]. 1) accesses the same variables after the refactoring as it did before the refactoring, 2) updates the same variables after the refactoring as it did before the refactoring, 3) performs the same method calls after the refactoring as it did before the refactoring.

Constraint with Action Language Ajila and Alam [S. A. Ajila and S. Alam, "Using a formal language constructs for software model evolution," in Semantic Computing, 2009. ICSC'09. IEEE International Conference on, 2009, pp. 390-395.] for the automatic dependency analysis of the model. A formal specification language called TLA (Temporal Logic of Actions) is used to specify the actions and the operations in CAL. A model checker is used to verify and reason about these TLA specifications

Sunyé et al [G. Sunyé, D. Pollet, Y. Le Traon and J. M. Jézéquel, "Refactoring UML models," «UML» 2001—The Unified Modeling Language.Modeling Languages, Concepts, and Tools, pp. 134-148, 2001.]  set for class diagrams and statecharts, which can be defined as OCL constraints at the metamodel level, both the pre-condition and the post-condition expressed in the OCL constraints would ensure that the applied refactorings are behavior preserving. refactorings set included the addition, removal, move, generalization and specialization of modeling elements for class diagrams and operations such as folding incoming/outgoing actions, unfolding entry/exit action, grouping states, folding outgoing transitions, unfolding outgoing transition, moving state into composite state and moving state out of composite state for statecharts

Gorp et al [P. Van Gorp, H. Stenten, T. Mens and S. Demeyer, "Towards automating source-consistent UML refactorings," «UML» 2003-the Unified Modeling Language.Modeling Languages and Applications, pp. 144-158, 2003.]   using OCL, in terms of a precondition of the restrictions that need to be satisfied in the model before applying the refactoring step, a post-condition of the properties to be satisfied in the model by the refactoring, and the “code smells” or the problem that can be improved by the refactoring. (Fujaba tool to apply the two refactorings: Pull Up Method and Extract Method.)

 Van Der Straeten et al [R. Van Der Straeten, V. Jonckers and T. Mens, "A formal approach to model refactoring and model refinement," Software and Systems Modeling, vol. 6, pp. 139-162, 2007]. The authors in this work formalized the behavioral specification of a system represented by UML state machine and sequence diagrams in Description Logic and defined two properties, observation call preservation and invocation call preservation, to check the behavior preservation between a class and its refactored version along with their corresponding state machine and sequence diagrams. Tool support is implemented, as a plug-in for the Poseidon CASE tool, and is tested on small examples.

Astels [D. Astels and others, "Refactoring with UML," in Proc. 3rd Int’l Conf. eXtreme Programming and Flexible Processes in Software Engineering, 2002, pp. 67-70] provides examples on bad design smells in class and sequence diagrams and described a number of refactoring actions for such model smells. The author was motivated with the fact that bad smells detection can be easier in the model level more than in the code level.

Simulated Annealing, which has reasonable performance on many search problems. In this specific problem because of the independent instinct of neighborhoods and discrete search space, this method is not supposed to perform well. [Fowler M., 1999, Refactoring: Improving the Design of Existing Programs, Addison-Wesley.]

“models may be viewed and treated as graphs, the algebraic graph transformation theory [Roz97, EEPT06] may be employed to describe model transformations in a formal, declarative, and rule-based fashion” [Brosch, P., Gabmeyer, S., Kappel, G. and Seidl, M., 2012. On formalizing EMF modeling operations with graph transformations. ACM SIGSOFT Software Engineering Notes, 37(4), pp.1-8.]

AMOUI 2006 - Investigation on the results of the whole evolution process make us con-

clude that search based algorithms, particularly genetic algorithm, are helpful

in improving special design metrics.

# Logical Model

The use of UML Use cases will describe the logical business requirements, these then follow through to Class Types which will be needed within the system. Sequence diagrams will outline how the class types interact, and following this a class diagram of the system will be produced to give a comprehensive view of the entire system which will allow implementation to be measured in terms of its progress, as well as to justify the correctness of the code in terms of its tractability to the original requirements.

# Physical model

## What issues arose during implementation?

How to use dependencies which have been stated from class to class but are actually dependencies which are within a given method which hasn’t been specified? The assumption needs to be made that all dependencies which are within the diagram have been as a result of parameters in operations or as attributes which are within the originating class. Thereby all dependencies which were in the original model need to be deleted in order to know exactly how to refactor the new dependencies in the refactored model once transformations have been applied. How do we identify where the associations are between classes when they are specific to certain operations?

## How will you show what has been achieved?

## Has anything not been finished?

# Conclusion

## Do your findings support or refute your claim?

## Have you met your original project goals?

## What limitations does your product have?

## What thoughts do you have now about your product?

# Future Work

## How could you improve your product?

## How could you improve the development process?

## What would you do differently?

# References

Amoui, M., Mirarab, S., Ansari, S. and Lucas, C. (2006) *A genetic algorithm approach to design evolution using design pattern transformation*. International Journal of Information Technology and Intelligent Computing,, pp.235-244.

[Arcelli, D.,](http://www.refworks.com/refworks2/?r=references|MainLayout::init) [Cortellessa, V.](http://www.refworks.com/refworks2/?r=references|MainLayout::init) and [Trubiani, C.](http://www.refworks.com/refworks2/?r=references|MainLayout::init) (2012) *Antipattern-based model refactoring for software performance improvement.* Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures. ACM.

Arcuri, A., White, D.R., Clark, J. and Yao, X. (2008) *Multi-objective improvement of software using co-evolution and smart seeding*. Simulated Evolution and Learning. Springer, pp.61-70.

Bowman, M., Briand, L.C. and Labiche, Y. (2010) *Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms*. IEEE Transactions on Software Engineering. 36 (6), pp.817-837.

[Brosch, P.](http://www.refworks.com/refworks2/?r=references|MainLayout::init), [Seidl, M.,](http://www.refworks.com/refworks2/?r=references|MainLayout::init)  [Wieland, K.,](http://www.refworks.com/refworks2/?r=references|MainLayout::init)  [Wimmer, M.](http://www.refworks.com/refworks2/?r=references|MainLayout::init) and [Langer, P.](http://www.refworks.com/refworks2/?r=references|MainLayout::init) (2009) *The operation recorder: specifying model refactorings by-example*. [Proceedings of the 24th ACM SIGPLAN conference companion on object oriented programming systems languages and applications](http://www.refworks.com/refworks2/?r=references|MainLayout::init). ACM.

Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W. and Schwinger, W. (2009) *An example is worth a thousand words: Composite operation modeling by-example.* Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg.

Brosch, P., Gabmeyer, S., Kappel, G. and Seidl, M., 2012. On formalizing EMF modeling operations with graph transformations. *ACM SIGSOFT Software Engineering Notes*, *37*(4), pp.1-8.

Brown, W.H., Malveau, R.C. and Mowbray, T.J. (1998) *AntiPatterns: refactoring software, architectures, and projects in crisis*. [John Wiley & Sons](https://en.wikipedia.org/wiki/John_Wiley_%26_Sons), ltd.

[Cortellessa, V.](http://www.refworks.com/refworks2/?r=references|MainLayout::init), [Di Marco, A.](http://www.refworks.com/refworks2/?r=references|MainLayout::init), [Eramo, R](http://www.refworks.com/refworks2/?r=references|MainLayout::init), [Pierantonio, A.](http://www.refworks.com/refworks2/?r=references|MainLayout::init) and [Trubiani, C.](http://www.refworks.com/refworks2/?r=references|MainLayout::init) (2010) *Digging into UML models to remove performance antipatterns*. ACM.

Dawson, C.W. (2005)*Projects in Computing and Information Systems: A Student's Guide*. Pearson Education.

Devos, F. (2004) *Patterns and Anti-Patterns in Object-Oriented Analysis*. Diss. Ph. D. Dissertation, KU Leuven, Department of Computer Science, Leuven, Belgium.

Eiben, A.E. and Smith, J.E. (2015)*Introduction to Evolutionary Computing*. Second Edition. Heidelberg: Springer.

Fowler, M. (2004) *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional.

Fowler, M. (1999) *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Grose, T.J., Doney, G.C. and Brodsky, S.A., (2002)*Mastering Xmi: Java Programming with Xmi, XML and UML (Vol. 21)*. John Wiley & Sons.

[Harman, Mark](http://www.refworks.com/refworks2/?r=references|MainLayout::init); [Jia, Yue](http://www.refworks.com/refworks2/?r=references|MainLayout::init); [Krinke, Jens](http://www.refworks.com/refworks2/?r=references|MainLayout::init); [Langdon, WB](http://www.refworks.com/refworks2/?r=references|MainLayout::init); [Petke, Justyna](http://www.refworks.com/refworks2/?r=references|MainLayout::init); [Zhang, Yuanyuan](http://www.refworks.com/refworks2/?r=references|MainLayout::init) (2014) *Search based software engineering for software product line engineering: a survey and directions for future work*. Proceedings of the 18th International Software Product Line Conference-Volume 1. ACM.

Harman, M., Mansouri, S.A. and Zhang, Y. (2012; 2013) *Search-based software engineering: Trends, techniques and applications*. ACM Computing Surveys (CSUR). 45 (1), pp.1-61.

[Jensen, Adam](http://www.refworks.com/refworks2/?r=references|MainLayout::init), [Cheng, Betty](http://www.refworks.com/refworks2/?r=references|MainLayout::init) (2010) *On the use of genetic programming for automated refactoring and the introduction of design patterns*. ACM.

Khalil, A. and Dingel, J. (2013) *Supporting the evolution of UML models in model driven software development: a survey*. Technical Report 602. School of Computing, Queen’s University, Ontario, Canada.

[Kim,Dae-Kyoo](http://www.refworks.com/refworks2/?r=references|MainLayout::init), [France, R.](http://www.refworks.com/refworks2/?r=references|MainLayout::init), [Ghosh, S.](http://www.refworks.com/refworks2/?r=references|MainLayout::init), [Song, Eunjee](http://www.refworks.com/refworks2/?r=references|MainLayout::init) (2003*) A role-based metamodeling approach to specifying design patterns.* IEEE.

Larman, C. (2005)*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd ed. Upper Saddle River, N.J: Prentice Hall.

Räihä, O. (2010) *A survey on search-based software design*. Computer Science Review. 4 (4), pp.203-249.

Räihä, O., Koskimies, K. & Mäkinen, E. (2011) *Multi-objective genetic synthesis of software architecture*. ACM, pp. 249.

Shaw, M., and Garlan, D. (1996) *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall

Simons, C., Parmee, I. and Gwynllyw, R. (2010) *Interactive, evolutionary search in upstream object-oriented class design.* Software Engineering, IEEE Transactions on, 36(6), pp.798-816.

Swanson, E. (1976) *The dimensions of maintenance*. Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society Press.

Vescan, A., Grosan, C., Pop, H.F. (2008) *Evolutionary Algorithms for the Component Selection Problem*. IEEE.