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

K. Sears

 BSc Computer Science

 January 1, 2016

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 dictates the overall architectural design of the software. However, as the size and complexity of software grows 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 to test with varying amounts of success the ability assist and automate refactoring.

It seems the use of AI will become ever more necessary as demand for interoperability for developing advancements such as multi-agent systems and big data becomes more prevalent in the software industry. These more recent developments 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 the OOP paradigm originally allowed.

## Search Based Software Engineering

Harman (Harman, 2012) one of the leading experts in the field of Search Based Software Engineering and who coined its name describes 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.”*

Search is defined in terms of Evolutionary Computation Search Methods which are techniques that can be broken into three distinct categories: Optimization, Modelling and Simulation (Eiben and Smith, 2015). Optimization is the driving force behind the refactoring of software, 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.

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.

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). 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 and will be the focus of this investigation. The area of design however has been largely overlooked as it accounts for only 10% of the entire field according to a Survey by Harman (Harman, 2009), with this said some success with EC search has already 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).

The reason why EC is used over exhaustive search in design refactoring is due to the nature of the problem domain 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 all possible combinations of attributes and methods distributed over a given number of classes. For succinctness attributes and methods will collectively henceforth be referred to as “elements” belonging to a model.

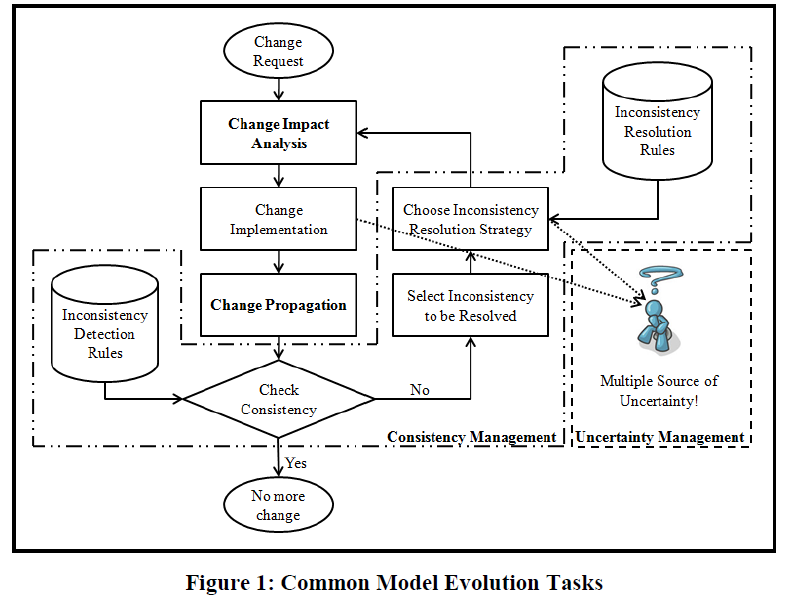
It is in this way that adding additional requirements attributing to extra elements produces a new set of combinations increased by an exponential factor, described by the binomial co-efficient equation:

 (x; y)=(x!)/(y!(x-y)!) 

Where x = the number of classes, and y = the number of elements and the result is the number of combinations present. This is excluding empty sets and discounting ordering, as elements to a class aren’t held privy to their sequence of occurrence as would be the case in permutations.

E. Swanson (Swanson, 1976) outlines the three primary types of evolution these being corrective, adaptive and perfective which respectively: repair faults in design, add further extensions or repurpose design and finally optimize design. This was extrapolated in a survey in the support of evolution of UML models conducted by Khalil (Khalil and Dingel, 2013) which further discusses four common tasks identified in model evolution which encompass the field well enough to structure the remaining background of this investigation around. [[1]](#footnote-1) These will be discussed in the following sections being, Change Impact Analysis, Change Propagation, Consistency Management and uncertainty Management. But in respect to Swanson’s work, considering its age, it was most likely more concerned with procedural programming rather than OOP, as perfective evolution following OOP principles would also potentially fall into the same category as adaptive, as an optimal solution is also one which is adaptable within the problem domain. Issues surrounding conflict of quality measure will be further discussed later, however caution was taken to avoid other potential archival inferences being included in this investigation.

These four tasks from Khalil’s work are best depicted in the flow diagram extract from the survey below.



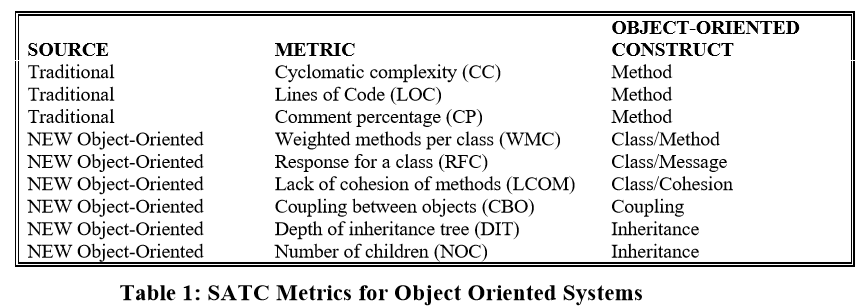
## Change Impact analysis

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

### Measuring software quality

Quality of software is divided into functional and non-functional requirements. The functional being to what degree the software accomplishes the business logic requirements which it set out to implement. This is often determined by testing the software under a multitude of predetermined test conditions.

Non-functional quality is measured through the use of Object Orientated quality metrics.

Dr. Linda H. Rosenberg presents in a table some of these metrics along with some of the previously successful ways in which procedural programming measured efficiency of design.

In order to evaluate computationally the fitness of a software model either before or after its refactoring, in order to compare and establish whether a change has improved its design, a good spread of metrics are required to be comprehensive in determining its quality. However, there are a significant number of metrics available and selecting all of them is unfeasible.

[*Distance from the Main Sequence*] (Amoui, 2006), Simons (Simons, 2010), [intra-modular coupling density (ICD), external relations penalty (ERP), groups/components ratio (GCR)] Bowman (Bowman, 2015),

### 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.

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. A controller class may provide an intermediary class between the user interface and the business logic. This way a change in requirements can 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 (Vlissides et al., 1995) states, does however add additional complexities to the refactoring process and furthers 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.”*

Another important point Vlissides reasons is that,

*“Object composition is favored over inheritance”*

Composition can give greater flexibility to classes at runtime, such as the creator class, it does however have the disadvantage of making code less intuitive to a developer as traceability of realizations becomes more challenging. This illustrates very well how measures of quality can conflict, in this case software flexibility against human readability. Multi-Objective Optimization is one approach to resolve this conflict of interest.

### 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.

### 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 (Gamma et al 1995). 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 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 identifier properties of the software the poor practice has resulted in. Additionally potential consequences of applying particular “fix” patterns are outlined to avoid inappropriate application of the wrong pattern to the problem. This makes Anti-patterns a more useful tool in the SBSE of software design than anti-patterns, as the search techniques may better exploit the explorative nature of their stochastic evolutionary operators without inhibition while providing a way of avoiding local optima which results in “false positives” of good quality according to metrics but poor design in terms of what has proven to not be effective architectures, e.g. the “blob” anti-pattern which will be discussed in more depth later. Being able to use anti-patterns in the context of automated design is also a key component to justify refactoring steps to the user which the AI has made.

It is hoped that the kinds of refactoring assistance that using EC can provide will help developers understand on a higher level the internal structure of the system without need for comprehensive knowledge of the system in its entirety to add functionality by employing design patterns. A good example of this is Jensen and Cheng’s (Jensen and Cheng, 2010) “RE-MODEL” tool. The tool which uses genetic programming (GP), whereby models are encoded in tree form with each node representing an attribute or operation for the refactoring UML class diagrams, 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”. A tree form is emphasized as a better approach over iterative refactoring as iterative refactoring does “not support the composition and interaction of multiple design changes in order to construct design artifacts such as design patterns”. Other than the refactored candidate solution decoded as output, the program provides the transformations which were performed, enabling traceability for the user to review.

## Change Propagation

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.

## Consistency management

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.

## Uncertainty Management

Uncertainty Management 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.

# 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.

# Development Methodology

Development of software systems can be categorized into various methodologies each with its own set of advantages and disadvantages. In this section we cover some of the most popular approaches to projects and conclude which would be best suited to this investigation.

## Waterfall

The waterfall methodology aims to comprehensively plan prior to any implementation all aspects of the project, including: summarizing requirements, identifying key components, designing architecture and preemptively resolving issues and errors before they occur.

The drawbacks of course are that new requirements to the system cannot be integrated once development has begun and so the methodology is inherently inflexible.

Also, being able to preemptively isolate issues before they arise is challenging and requires experienced developers working very closely with stakeholders to ensure that there is consensus to the understanding of the requirements.

Applying Waterfall in practice within industry may be a reasonable approach as the stakeholders know in more fixed terms their requirements, however, its use within academic research, such as this investigation, would be inappropriate due to requirements being dynamically dependent on iteratively referenced works from multiple sources.

## Extreme Programming

Extreme programming is a form of agile methodology whereby frequent releases are developed alongside the stakeholder so that fast feedback to design decisions may be made. The method is often done using pair programming, where a couple of developers work at a single computer in order to program, with the idea of spotting mistakes before they expend resources and to “bounce” ideas on what the best approach to problems would be through combined experience.

Because of this project being a singularly manned, therefore having the stakeholder as the developer and the lack of cooperation from other developers, this methodology, along with most other agile methods, is unsuitable. However, the aspect of generating frequent releases may be adopted if during development complete rewrites would be required due to unforeseen circumstances.

## Rapid Application Development

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.

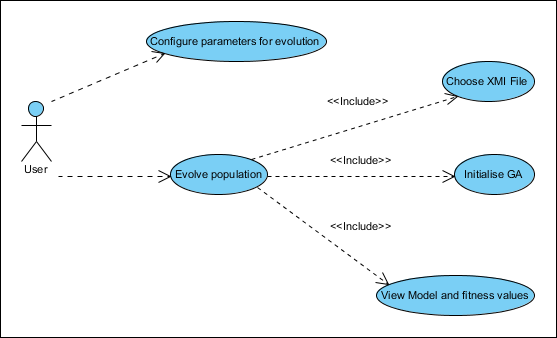
# Test Approach

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.

## Use Cases

Use cases outline the use of the software by the user and describe its functional requirements.



### Use Case 1: Configure parameters for evolution

Pre-Conditions: None

Events:

1. User: sets population size, the number of generations they would like to evolve the model by, the mutation rate per element, and any Anti-patterns they wish to avoid in the search, and whether they want the population genomes randomized on initialization.

Post-Conditions: System is set up for running the Genetic Algorithm with desired parameters.

### Test Case 2: Choose XMI File

Pre-Conditions: None

Events:

User: Clicks File item in Menu bar and selects Load from the drop down menu.

System: Displays File Chooser allowing the selection of a single file.

User: Selects desired file and presses OK button.

System: Displays the fitness of the Model in the Fitness Overview Graph.

Post-Conditions: Model is loaded into memory ready to be evolved. It is able to be viewed in the “current generation” chart, along with its metrics.

### Test Case 3: Initialize Genetic Algorithm

Pre-Conditions: Model has been selected from file and loaded into memory.

Events:

User: user clicks Initialize Genetic Algorithm button.

System: generates population of candidate solutions and displays the total fitness of the population in the Overall Fitness Chart.

Post-Conditions: current generation is prepared to be evolved again, including with changes made to parameters to alter performance.

### Test Case 4: View Model and Fitness Values

Pre-Conditions: Model has been selected from file and loaded into memory.

Events:

User: selects the tab view for Current Generation and selects a given candidate solution from the current generation AreaChart.

System: opens in new window the model selected and displays it as a class diagram alongside a chart of its fitness metrics.

Post-Conditions: current generation is prepared to be evolved again, including with changes made to parameters to alter performance.

# Design

## Evolutionary Computation

Use of a Genetic Algorithm seemed appropriate, despite the reservations of Jensen and Cheng’s research which suggests Genetic Programming as the best candidate for capturing Design Pattern meta-data. This decision was made due to the success GAs have had in previous works, Simons (2010), Bowman (2010), Amoui et al. (2006) to name a few. As well as the work by Cortellessa (2010) which suggests a methodology for removing performance anti-patterns based upon OCL which does not require the use of a tree structure representation which can often become complex and less cost effective to search compared to discrete representations, but this discussion is outside of the scope of this investigation.

The use of a population based search technique also seemed advantageous due to the way it lends itself to exploit Multi-Objective Optimization and in turn even co-evolution. As candidates can explore different areas of the search space simultaneously and alter their search according to the areas which appear to provide better results, exploring before prematurely converging on areas of local optima. This becomes even more relevant with the use of Anti-patterns which may dictate areas within the search space which appear cost-effective from a metrics standpoint, but which otherwise are unprofitable in terms of good architectural design which has been tried and proven to be of less value.

## Metrics

Bowman uses dependencies among methods and attributes to calculate the coupling and cohesion between classes. Because of coupling and cohesion being considered of the most use to the problem domain in respect to OOP for their simplicity as well as their ability to abstractly capture the essence of even rudimentary architectural designs, these were the first to be employed as metrics in this prototype design.

Due to the limited nature of the project not including the dynamic behaviour requirements capture from sequence diagrams, role-based modeling language, or Object Constraint Language, it was boldly decided to convert composition and aggregation associations between classes into attribute “collections” of the target classes type within the source classes attribute collection. From this a primitive form of Method-Attribute coupling [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.], could be obtained. This unfortunately will limit the accuracy of the project by not accounting for Method-Method Coupling and Method-Generalisation Coupling, but due to the vast number of metrics available, and the time constraints given, only what was considered the most essential form of coupling could be accommodated at this stage.

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 of Cohesive Interactions.

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.

## Evolutionary Operators

Due to the use of a GA, the evolutionary crossover operator does become problematic, as the crossover of two encoded representations may produce duplicates which would require identifying and removing, which both increases the time of running as well as provokes the question of which duplicate would be best to remove, which without considerable input from the user, would be difficult to discern with no semantic understanding of the underlying model’s logic. For this reason as well as due to time constrains, crossover has been left out of this prototype as it is the less essential of the two evolutionary operators with it not providing

## Parameters

Too high mutation rate turns into random search!

Quite large population to avoid genetic drift

## 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.

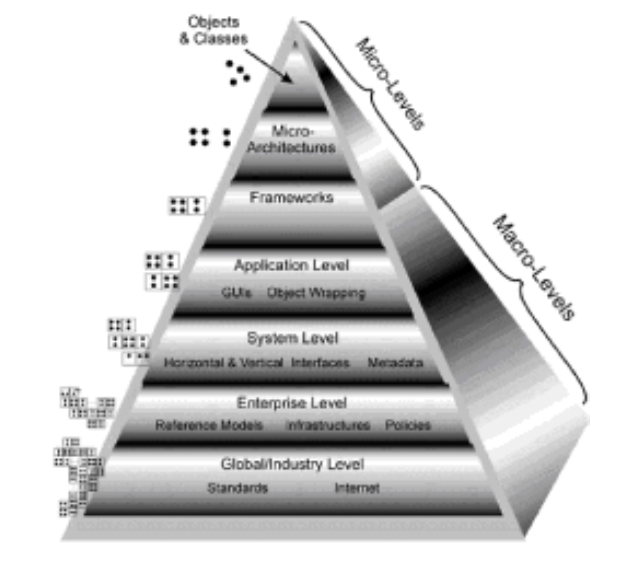
### Repository

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

## 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.

1. 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 assisting developers in the refactoring process. 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.

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

   [↑](#footnote-ref-1)