

Towards Automated Design Improvement Through Combinatorial Optimisation

Mark O’Keeffe and Mel Ó Cinnéide

Department of Computer Science, University College Dublin, Ireland

E-mail: marko@ihl.ucd.ie, mel.ocinneide@ucd.ie

Abstract

In this paper we present a novel approach to the problem of automated design improvement: by treating object-oriented design as a combinatorial optimisation of metrics we have developed a prototype software engineering tool capable of improving a design with respect to a conflicting set of goals. As the prioritisation of different goals is determined by weights associated with each metric, we also describe here a method of assigning coherent weights to a set of metrics based on object-oriented design heuristics.

The combinatorial optimisation approach to automated design improvement is illustrated here by means of a simple case study, which shows the effect of applying our prototype tool to a small inheritance hierarchy. Results indicate that a balance between metrics has been achieved, as several potentially conflicting design goals are accommodated.

1. Introduction

One of the most significant problems with the popular iterative approach to object-oriented software engineering is *design erosion*; that is, the loss of desirable design attributes as new functionality is added during development. This is a systemic as well as practical problem since the optimum configuration for a set of classes clearly cannot be known until the required interactions are established. In addition, it is likely that further domain abstractions will be discovered during the development process and new classes added to the design, potentially invalidating previous design decisions. Design erosion may be even be unavoidable in practice, as noted by van Gurp and Bosch [22]. In order to ameliorate the problem it has been suggested to improve design without changing external behaviour, or *refactor*, after each addition of functionality [12] in order to ensure that the underlying design is sound. While this practice undoubtedly reduces design erosion and makes further addition of functionality easier, it has a high associated cost — reengineering can take up the bulk of available programming time [11]. Hence, there is a need for software engineering envi-

ronments to provide automated tool support for refactoring. At present, such support is limited to automatically carrying out a refactorings at the direction of the user [1][2][3]. While this is a step in the right direction, it is only a partial solution.

The ideal solution to this problem, if it were possible, would be the automation of the refactoring phase of each iteration by the application of an automated design improvement tool. Such a tool would take the current set of classes as input and output a set with the same external behaviour, but having an improved design. Due to the complexity of the task of object-oriented design, with its often conflicting goals and lack of a definitive methodology, it is hard to see a software engineering tool replacing the “insight, taste, experience, and (a) sense of aesthetics” [7] of a seasoned software architect. However, it seems intuitive that there exists some class of general design flaws that are likely to recur in incremental development of object-oriented systems. The difficulty in automatically addressing these is that what constitutes a flaw in an object-oriented design often depends entirely on the context, making the detection and even the definition of design flaws extremely problematic. So we are left with a set of goals such as high cohesion, low coupling and good encapsulation and heuristics on how to achieve these, but little other than experience to guide us in resolving conflicts, making automation of the task difficult. However, if design flaws of this common nature could be automatically removed from a program, the designer faced with a reengineering problem would be free to spend time on the more difficult design decisions that require intuition and experience.

Several methods of automated design improvement that could potentially ameliorate some design flaws have been proposed [4][8][18], but most have a common drawback — they aim to optimise one particular aspect of design while ignoring others. This leads to designs that are excellent in one regard, but poor overall. For example, Casais’ seminal work on inheritance hierarchy restructuring [4] and Moore’s approach to the same problem [17][18] present algorithms that maximise code sharing by rearranging classes within the hierarchy and adding new classes where neces-

sary. However, both methods can result in a loss of cohesion, as ensuring that methods which call similar sets of methods remain in the same class is not considered. In other words, the focus on preventing code duplication results in a loss of the semantic information held in the class structure, which reduces the understandability and hence maintainability of the design.

Our aim is the creation of a tool that aids software maintenance by automatically improving the design of a system in terms of a set of object-oriented metrics powerful enough to capture the numerous and often conflicting aspects of sound design. In this paper we propose a novel approach to automated design improvement; treating design as a combinatorial optimisation problem that can be solved using established techniques.

2. Design as a Combinatorial Optimisation Problem

Although most work in the field of software metrics has been intended to assist in project management [5][10][13], metrics also exist that assist the programmer in making lower-level design decisions. Indeed, many programming heuristics can be expressed as the maximisation or minimisation of metrics. For example, the heuristic ‘All data should be hidden within its class’ [20] could be expressed as ‘Minimise the number of public and protected attributes in each class’. More abstract goals such as high cohesion can be reduced to combinations of metrics such as ‘Minimise the number of methods accessing an attribute that are defined outside the encapsulating class of that attribute’. It is therefore plausible that sets of linear metrics can capture many of the disparate aspects of sound object-oriented design. There is a difficulty in using such information in a constructive manner however, since any refactoring that improves some metric values is likely to degrade others. This is because object-oriented design involves a variety of trade-offs; precise domain abstraction can result in a proliferation of classes, sharing responsibilities between high-level classes may increase coupling, and so on.

Our thesis is that, by treating design as a combinatorial optimisation problem, where the goal is maximisation of a set of object-oriented design metrics, we can effectively apply established computational techniques to the problem of automated design improvement. In other words, we take a weighted sum of metric values as a quality function and search for designs that give a higher result for this function in order to increase overall design quality. The principal challenge in this approach is the discovery of a set of metrics and associated weight values that are sufficiently accurate in measuring the extent of a program’s adherence to design heuristics, and hence the quality of its design. Efforts to date on the theoretical side of this project have therefore

been focussed on the development of a systematic method for assigning coherent weights to a set of metrics for object-oriented design in order to produce a quality function that accurately reflects the chosen heuristics.

3. Object-Oriented Design Quality

When seeking to increase design quality it is of course necessary to have a clear conception of what constitutes good (object-oriented) design. A survey of the object-oriented metrics literature revealed that there is no established metric suite fully validated as capturing design quality. Where there is partial validation, such as in the case of the Chidamber-Kemerer metrics [5] this takes the form of a proven link between metric values and properties such as fault-proneness of classes [21] or maintenance effort as measured by number of lines changed [16]. While this shows that the metrics can be used to give some idea of the correctness or maintainability of a design, it does not prove that they are good indicators of overall quality, which includes other facets such as reusability and extensibility. In other words, while established metrics suites may be useful for their intended purpose of assisting in project management they would seem to be ill-suited to applications that require more detailed information.

It was therefore decided to adopt a slightly different model of design quality in this work; that of compliance with established design heuristics. Sets of heuristics such as Arthur Riel’s ‘Object-Oriented Design Heuristics’ [20] provide comprehensive and coherent catalogues of accepted design goals and the means to achieve them. Design quality is therefore defined here as the level of adherence to a set of design heuristics; up to now a subset of Riel’s [20] has been used. It is intended that the combinatorial optimisation method developed in this work be flexible enough that any suitable set of heuristics could be substituted without undue difficulty, in order to facilitate expansion and adaptation.

4. Dearthóir

4.1. Overview

Dearthóir is a prototype design improvement tool capable of restructuring a class hierarchy and moving methods within it in order to minimise method rejection¹, eliminate code duplication and ensure superclasses are abstract where appropriate. Automated behaviour-preserving transformations (*refactorings*) are applied stochastically to an object-oriented program to produce alternative designs, which are

¹Java does not feature explicit method rejection; however, methods may be inherited by a class but never called on an instance of that class, thus needlessly increasing its complexity. Therefore we define a rejected method in Java as one which is inherited but never used.

evaluated using a set of metrics designed specifically for this purpose. In this approach the task of design improvement becomes a search through the space of alternative designs for those that are superior to the original, judging by the metric values.

It is of course necessary to transform the set of metric values for any design to a single quality value in order to rank alternative designs. The evaluation function used by *Dearthóir* to achieve this is a weighted sum of metric values:

$$q_d = \sum_{m=1}^n w_m \text{metric}_m(d)$$

where q_d is the quality value for a design d , n is the number of metrics, w_m is the weight on metric m , and $\text{metric}_m(d)$ is the value for d of metric m . The metrics used are described in section 5.3.2, while the set of refactorings is described in 4.3.

4.2. Simulated Annealing

Dearthóir uses the simulated annealing (SA) technique to find close-to-optimum solutions to the combinatorial optimisation problem described above. Although any method of solving such problems could be used SA was selected because stochastic techniques have been shown to be effective in solving hard combinatorial optimisation problems such as the timetabling problem, where conventional search techniques require an infeasible run-time [6]. The disadvantage of simulated annealing is that solutions are not provably optimum, but since our goal is to discover a solution of high quality in a reasonable time, rather than find the very best solution, this is not an issue.

A simulated annealing search essentially involves making series of tentative changes to some solution of a combinatorial optimisation problem. Changes which increase the quality of the solution are accepted, and the changed solution becomes the starting point for the next series of tentative changes. In addition, some changes which reduce the quality of the solution are accepted to allow the search to escape from local minima. Such (negative) changes are accepted with a probability that decreases steadily during the annealing process. The main features of a simulated annealing system are an evaluation function such as that described above, and a means of changing solutions — which in this case is the set of automated refactorings. A *cooling schedule* is also required that determines how quickly the annealing runs, and hence how likely the solution is to be of high quality.

4.3. Refactorings

The refactorings used in *Dearthóir* are behaviour-preserving transformations on Java code. Recent advances in the automation of refactorings [19] [9] [15] have encouraged the development of several tools which can carry out refactorings under human direction, however only the subset that can be fully automated are suitable here. In order to limit the scope of design changes made by *Dearthóir* until the work is more mature, the refactorings currently employed are limited to those that have an effect on the positioning of methods within an inheritance hierarchy. It should be noted that in order for the simulated annealing search to move freely through the search space every change to the design must be reversible; to ensure this we have chosen pairs of refactorings that complement each other:

1. Move a method up or down in the class hierarchy
 - `pullUpMethod` - Moves a method to the superclass of its containing class.
 - `pushDownMethod` - Moves a method to a subset of the subclasses of its defining class. The subset consists of those classes that require the method themselves, and those that have a descendant that requires the method.
2. Extract or collapse a subclass
 - `extractSubclassFromAbstractClass` - Creates a new subclass of an abstract class. A random subset of the subclasses of that (abstract) class are made to inherit from the new class.
 - `collapseFeaturelessClass` - Removes a class that has no constructors, methods or attributes from the hierarchy.
3. Make a class abstract or concrete
 - `makeClassAbstract` - Changes a non-abstract class to an abstract class. Where the class in question has no constructors this is trivial; where it does, it is necessary to create a new concrete subclass.
 - `makeClassConcrete` - Changes an abstract class to a non-abstract class. If a concrete subclass of this class was created by the above refactoring this is deleted, and its features moved to its superclass.
4. Change superclass link of a class
 - `changeSuperclassDown` - Changes the superclass of a class so that the class in question is repositioned at a lower point in the inheritance hierarchy.

- `changeSuperclassUp` - Changes the superclass of a class so that the class in question is repositioned at a higher point in the inheritance hierarchy.

5. A Method for Developing Heuristic-Compliance Metric Sets

5.1. Overview

The goal of this technique is the transformation of a set of quite specific heuristics into an evaluation function suitable for use in a combinatorial optimisation program. While some heuristics will obviously not be suitable for transformation to metrics, e.g. “Ensure code is well commented”, some heuristics will initially appear to be suitable but prove otherwise. The following method is intended to filter out heuristics that cannot easily be transformed into valid metrics because they are vague, unsuitable for the programming language in use or dependent on semantics.

5.2. The Method

1. For each heuristic:
 - (a) Define property to be maximised or minimised in the heuristic
 - (b) Determine whether the property can be accurately measured. If a metric for this property does not exist and cannot be readily identified the heuristic may be too general to be of use in this approach - consider rejecting it.
 - (c) Note whether the metric should be maximised or minimised.
2. Identify dependencies between metrics — are there cases where a change in the value of one metric must result in a change in the value of another?
3. Establish precedence between dependent metrics, and thresholds where necessary. This step involves the prioritisation of some heuristics over others and requires an understanding of the goals of each heuristic and the trade-off between them. As this is a subjective process it may be useful to record assertions made; these are effectively additions to the model of design quality suggested by the chosen heuristics.
4. Check that the graph of precedence between metrics is acyclic — any cycle within the graph indicates that the set of metrics is unworkable and must be modified.
5. Weights should now be assigned to each of the metrics according to the precedences and thresholds established.

5.3. Application in Dearthóir

5.3.1 Heuristics

The heuristics chosen for the *Dearthóir* prototype are those judged as being particularly relevant to the positioning of methods and presence of abstract classes in an inheritance hierarchy. In other words, heuristics were selected that would lead to metrics that could be affected by the set of refactorings currently employed. All heuristics are taken from Arthur Riel’s ‘Object-Oriented Design Heuristics’ [20], but are renumbered below for convenience. The heuristics selected were the following:

1. Minimize the number of messages in the protocol of a class ([20], p.17).
2. Eliminate irrelevant classes from your design ([20], p.42).
3. If two or more classes have common data and behaviour (i.e. methods), then those classes should each inherit from a common base class which captures those data and methods ([20], p.97).
4. All base classes should be abstract classes ([20], p.89).

We have attempted to capture the above heuristics precisely in metric form to quantify design quality, with the exception of the *abstract superclass rule* or ASR (heuristic 4 above), which states that all classes that are not leaves on the inheritance tree should be abstract. In [14] Hürsh discusses the advantages and disadvantages of compliance with this rule in various situations, as well as describing a refactoring that can be applied to any concrete superclass in order to comply with this rule. Our interpretation of these findings is that compliance with the ASR is desirable except where it results in an unnecessary proliferation of classes. We therefore consider a weakened version of the abstract superclass rule, that we name the *abstract superclass guideline* (ASG), beneficial in object-oriented design:

Abstract Superclass Guideline 1 *All classes that are not leaf classes of the inheritance tree should be made abstract, unless doing so results in the addition of a featureless concrete class.*

5.3.2 Metrics

The following are the results of applying step one from 5.2 to the heuristics above.

1. Rejected Methods (RM) - minimise
From heuristic 1; the total number of ‘inherited but not used’ methods for all instantiable classes in a design.

Rejected methods will increase the size of class interfaces unnecessarily and therefore make maintenance and class reuse more difficult. In Java methods are not explicitly rejected; in our current system we consider a method rejected where it is inherited, but no instance of the inheriting class has the method called on it within the system. It follows that only concrete classes can reject methods.

2. Unused Methods (UM) - minimise

Also from heuristic 1; the number of methods in the design that are defined by a class but never called on an instance of that class, and not called by the class internally. Such methods increase the size of the class interface unnecessarily.

3. Featureless Classes (FC) - minimise

From heuristic 2; the number of classes in a design that have no constructors, methods or fields. These should be avoided where possible since they have no responsibilities and increase design complexity.

4. Duplicate Methods (DM) - minimise

From heuristic 3; the number of methods that are duplicated within an inheritance hierarchy. These should be eliminated since code duplication hampers maintenance.

5. Abstract Superclasses (AS) - maximise

From the Abstract Superclass Guideline; the number of abstract classes in a design. Any class that can be declared abstract should be, since it is effectively abstract. In addition, most superclasses should be abstract - see 5.3.1.

5.3.3 Interaction Between Metrics

As mentioned in section 2, metrics for object-oriented design will often conflict; for example, the goal of minimising public interfaces conflicts with the goal of total encapsulation since adding accessor methods increases the size of the interface. This must be taken into account in treating design as a combinatorial optimisation problem, since each metric in the evaluation function must be weighted to ensure correct prioritisation of design goals.

The following dependencies were identified in carrying out step 2 of the method above:

1. Abstract Classes must be of a lower priority than Featureless Classes; otherwise a design with more abstract classes would be favoured even if they were all empty.

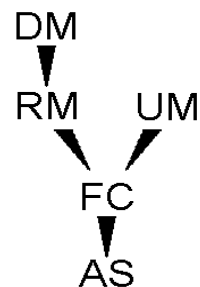


Figure 1. Relationships between metrics.

2. Rejected Methods and Unused Methods must be of higher priority than Featureless Classes; otherwise methods may be moved into classes where they do not belong in order to make those classes non-featureless.
3. Duplicated Methods must be of a higher priority than Rejected Methods; otherwise identical methods could be placed in sibling classes in order to prevent method rejection - we consider code duplication even less desirable.

Figure 1 illustrates the dependencies among the *Dearthóir* metrics; each metric must be more heavily weighted than its children in the graph. It can be seen that the graph of dependencies between metric weights is an acyclic graph; if this were not the case the set of metrics would be unusable, as noted in section 5.2.

Step 3 of the method for assigning coherent weights to heuristic-compliance metrics involves establishing precedence and determining thresholds. In this case, precedence has already been established since each metric is connected to the precedence graph (figure 1). If a metric were not connected it would be necessary at this point to decide where it should fit into the weighting scheme.

Most of the dependencies in the graph do not require thresholds; for example, Abstract Classes must be weighted less than Featureless Classes as a design with an empty abstract class would otherwise be preferred to a design without one, but the relationship is one-to-one — there will either be one abstract class and therefore one abstract featureless class or there will not. In contrast, a duplicate method is avoided by pulling the method up into its superclass, which could result in the method being rejected by any number of classes. We must therefore establish a threshold value for this dependency — it could be said that it is more important to avoid code duplication than any amount of method rejection; therefore the threshold can be an arbitrarily high number.

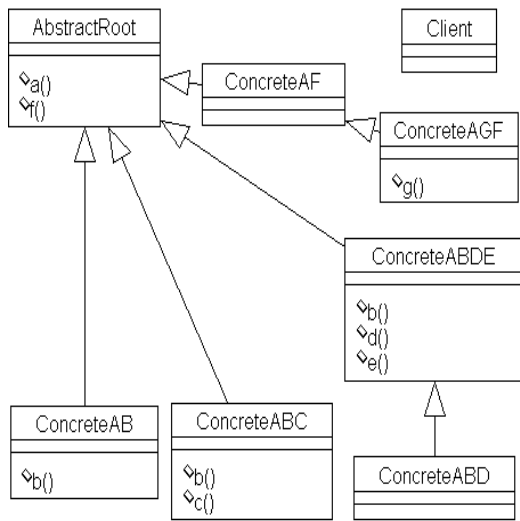


Figure 2. Input design.

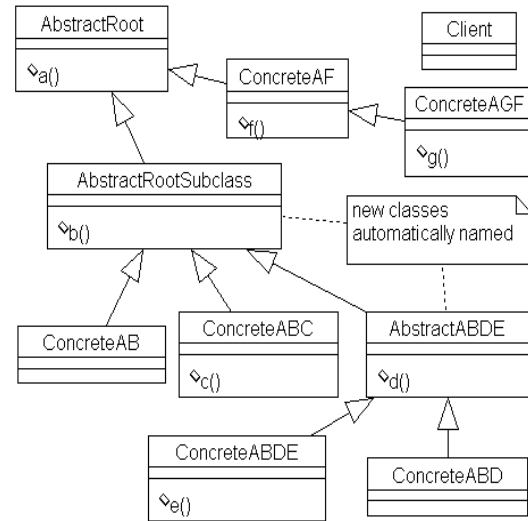


Figure 3. Output design.

6. Case Study

6.1. Overview

In this case study the prototype design improvement tool was applied to a small inheritance hierarchy. As mentioned above, *Dearthóir* works by repeatedly applying a randomly chosen refactoring to the current design as part of a simulated annealing process. The quality of the designs produced is measured by a weighted sum of metric values, so several potentially conflicting design goals can be simultaneously pursued.

6.2. Input

The source code of a simple program consisting of the eight Java classes shown in figure 2. Seven of the classes form an inheritance hierarchy, while the other is a client class that instantiates classes within the hierarchy and calls methods on them. Methods required by a class are indicated by the capital letters at the end of the name of that class. Methods called by the client are considered necessary for the class called on and therefore not rejected. The input design is flawed in that it contains method duplication (method `b()` in three classes), method rejection (e.g. method `e()` rejected by `ConcreteABD`), and does not comply with the abstract superclass guideline (`ConcreteABDE` is not abstract).

6.3. Results

The design of the output classes is shown in figure 3. Although the new design appears more complex at first glance,

upon further inspection it can be seen that three of the five metric values have been improved with the addition of only two classes. The design has been improved in the following ways:²

- The method `f()` has been moved to the class `ConcreteAF` so that it is no longer rejected by `ConcreteAB`, `ConcreteABC`, `ConcreteABDE`, and `ConcreteABD`.
- A new subclass of `AbstractRoot` has been extracted and the method `b()` moved to it so that can be available to `ConcreteAB`, `ConcreteABC`, `ConcreteABDE`, and `ConcreteABD` without being duplicated or rejected by other classes.
- The class `ConcreteABDE` has had the Abstract Superclass Transform performed on it, so the method `e()` is available to it, without being rejected by the class `ConcreteABD` which now inherits from `AbstractABDE`. Note that the class `ConcreteAF` has not been made abstract despite being a concrete superclass. This is because it contains no methods that are not used by its subclass, and therefore a new concrete subclass of `ConcreteAF` would be a featureless class.

6.4. Interpretation

The metric values for the input and output designs are shown in table 1. It can be seen from figures 2 and 3, and

²Although many intermediate designs are passed through during the annealing process, we discuss results in terms of the accumulated differences between input and output.

Table 1. Metric values for input and output

	RM-	UM-	FC-	DM-	AS+
Input	5	2	2	3	1
Output	0	2	2	0	2

from the values in table 1, that the output design is superior to the input design in terms of the metrics used. The output contains two more abstract classes, three less duplicate methods and five less rejected methods than the input, while the numbers of featureless classes and unused methods remained constant.

Given that *Dearthóir* has succeeded in changing the design to improve the metric values, we now consider whether the new design is an improvement from a human perspective. On inspection of the input and output designs the most obvious difference is the introduction of a subclass of `AbstractRoot`, which contains the method `b()` in the output design. This alone represents a significant improvement, since `b()` was previously duplicated across three classes; a situation which would have made any change to the method much more difficult to implement. Without adding the subclass the only solution to the code duplication would have been to pull the method `b()` up to the `AbstractRoot` class, but this would have meant making it available to two classes that do not require it. So, in the output design it is easier to make methods available to those classes that require them, without adding to the interfaces of classes that do not. This is a significant gain in terms of maintainability of the design.

Another difference between the input and output designs that stands out lies in the general positioning of methods, and the associated decrease in method rejection. In the input design several classes contain clumps of methods, whereas in the output design methods are spread quite evenly between the various classes. This indicates that responsibilities are being distributed more evenly among the classes, which means that components of the design are more modular and therefore more likely to be reusable. This indicates that adherence to low-level heuristics can lead to gains in terms of higher-level goals.

7. Conclusion

In this paper we have presented a novel approach in the direction of automated tool support for software engineering, aimed at reducing the cost of software development by automated design improvement. By treating the task of improving an object-oriented design as a combinatorial optimisation of metrics it is possible to search for designs that are superior in terms of the metrics and associated weights selected. If this technique can be extended to encompass

a greater number of design goals it may be possible for future software engineering environments to automatically improve the maintainability and extensibility of prototype designs, and so reduce the cost of development.

The principal benefit of this approach is that it facilitates automated refactoring towards a design that can be considered a compromise between conflicting goals — an automated design improvement tool embodying this property is likely to be applicable in a considerably wider range of situations than a tool capable of maximising one design property to the detriment of others. We have demonstrated here the utility of the approach in restructuring a small inheritance hierarchy and repositioning methods within it in order to achieve the potentially conflicting goals of minimal code duplication and method rejection while avoiding a proliferation of classes.

While the work presented here can be considered a proof-of-concept for automated design improvement through combinatorial optimisation, some issues remain to be addressed in order to establish whether the technique will be effective in practice. Apart from practical concerns such as runtime and communicating design changes to the user, the main issue concerns the difficulty of establishing coherent weights for a set of metrics sufficient to give an accurate measure of overall design quality — and there is also a question over whether a definitive set of metrics can be found. As a first step towards addressing these issues we have developed the method for deriving metrics and associated weights from design heuristics described above. Given that the number of refactorings that can be implemented automatically is limited, not all heuristics will be useful in the automated approach, so the task we face in defining a suitable set of metrics is somewhat simpler than discovering the ideal metrics suite for object-oriented design.

Future work will include tests on larger (real world) designs and the expansion of the refactoring capabilities of *Dearthóir*. There will be a concurrent need to examine further heuristics, and continued development of the method for deriving metrics from heuristics.

8. Acknowledgements

The authors would like to acknowledge the financial contributions of Enterprise Ireland and the Faculty of Science, University College Dublin to this project.

References

- [1] IntelliJ, <http://www.jetbrains.com/idea/features/refactoring.html> - 5 march 2004.
- [2] Jrefactory, <http://jrefactory.sourceforge.net/> - 5 march 2004.

- [3] Refactorit, [http:// www.refactorit.com/ ?id=1370](http://www.refactorit.com/?id=1370) - 5 march 2004.
- [4] E. Casais. An incremental class reorganization approach. In O. L. Madsen, editor, *Proceedings of the European Conference on Object-Oriented Programming*, pages 114–131, Utrecht, June 1992. LNCS.
- [5] S. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20:476–493, June 1994.
- [6] A. Colorni, M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini, and M. Trubian. Heuristics from nature for hard combinatorial optimization problems. In *International Transactions in Operational Research*, pages 1–21, March 1996.
- [7] J. O. Coplien. Software design patterns: Common questions & answers. In *Proceedings of Object Expo New York*, pages 39–42, New York, June 1994. SIGS Publications.
- [8] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. In *Conference on Object-Oriented Design*, pages 251–267, 1996.
- [9] S. Ducasse, M. Rieger, and G. Golomingi. Tool support for refactoring duplicated oo code. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology: ECOOP’99 Workshop Reader*, number 1743 in LNCS, Lisbon, June 1999. Springer Verlag.
- [10] F. B. e Abreu and R. Carapuca. Candidate metrics for object-oriented software within a taxonomy framework. *The Journal of Systems and Software*, 26(1):87–96, July 1994.
- [11] A. Eerola. A disciplined approach to the maintenance of the class hierarchy. In *Technical Report Raport A/1999/2, University of Kuopio, Department of Computer Science and Applied Mathematics*, 1999.
- [12] M. Fowler. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley Longman, Reading, Massachusetts, 1999.
- [13] J. Hogan. An analysis of OO software metrics. Technical Report CS-RR-324, Coventry, UK, 1997.
- [14] W. L. Hürsch. Should Superclasses be Abstract? In *European Conference on Object Oriented Programming*, pages 12–31. e, 1994.
- [15] A. Kupin. Design and development of program transformation tool. Master’s thesis, University of Munich, Department of Computer Science, Aug. 2000.
- [16] W. Li and S. Henry. Object oriented metrics that predict maintainability. *Journal of Systems and Software*, 23, 1993.
- [17] I. R. Moore. Guru - a tool for automatic restructuring of self inheritance hierarchies. In *TOOLS USA*, pages 267–275. Prentice-Hall, 1995.
- [18] I. R. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 235–50, San José, Oct. 1996. ACM.
- [19] M. Ó Cinnéide. *Automated Application of Design Patterns: a Refactoring Approach*. PhD dissertation, University of Dublin, Trinity College, Department of Computer Science, 2000. Available from: <http://www.cs.ucd.ie/staff/meloc/home/papers/thesis>.
- [20] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, Reading, Massachusetts, first edition, 1996.
- [21] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An empirical study on object-oriented metrics, Nov. 1999.
- [22] J. van Gurp and J. Bosch. Design erosion: problems and causes. *J. Syst. Softw.*, 61(2):105–119, 2002.