Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems

Olaf Seng, Johannes Stammel and David Burkhart
FZI Forschungszentrum Informatik
Haid-und-Neu-Strasse 10-14
Karlsruhe, Germany
{seng,stammel,burkhart}@fzi.de

ABSTRACT

A software system's structure degrades over time, a phenomenon that is known as software decay or design drift. Since the quality of the structure has major impact on the maintainability of a system, the structure has to be reconditioned from time to time. Even if recent advances in the fields of automated detection of bad smells and refactorings have made life easier for software engineers, this is still a very complex and resource consuming task.

Search-based approaches have turned out to be helpful in aiding a software engineer to improve the subsystem structure of a software system. In this paper we show that such techniques are also applicable when reconditioning the class structure of a system. We describe a novel search-based approach that assists a software engineer who has to perform this task by suggesting a list of refactorings. Our approach uses an evolutionary algorithm and simulated refactorings that do not change the system's externally visible behavior. The approach is evaluated using the open-source case study *JHotDraw*.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design

Keywords

Refactoring, Evolutionary Algorithms, Software Metrics, Design Heuristics

GECCO'06, July 8–12, 2006, Seattle, Washington,USA.

1. INTRODUCTION

According to Lehman's first law a software system that reflects some external reality undergoes continuous change or becomes progressively less useful [9]. Since a system's original design is rarely prepared for every new requirement and the changes have to be made quickly by different people without properly adjusting the system's structure, it gets harder and harder to maintain the system. This phenomenon is known as software decay [8].

As a consequence, a software system's structure has to be constantly reconditioned. Using known heuristics and metrics [11], one can assess the quality of this structure as well as identify the spots having a negative impact on its maintainability. These spots are instances of so called bad smells. Typical examples for these bad smells are high coupling between subsystems or god class (a large class providing more than one abstraction) [13].

The hard question is how to remove these bad smells. Refactorings [8] provide a powerful means to get rid of them, because their application does not change the externally visible behavior. But deciding which refactoring to apply and where to apply it is a tough question, since refactorings can have undesired side effects on the system's structure. Removing one bad smell might result in creating a new one at another place. Just imagine that you try to get rid off a god class by moving several of the methods belonging to it to other classes. This could result in the creation of a new god class, because the other classes already contain several methods.

For improving the subsystem decomposition, methodologies based on search algorithms have already been proven to be quite successful [16][6]. In this paper we show how to use an evolutionary algorithm for optimizing the class structure of a system. This task is much more difficult because it is much harder to specify behavior preserving class level refactorings.

We designed a methodology for object-oriented systems that helps the user to determine refactorings to improve the class structure of a system with respect to the values of several metrics and the number of violations of object-oriented design principles. User input is only necessary for setting initial parameters. Our approach suggests a list of behavior preserving refactorings. Of course, the final decision whether to apply or not a proposed refactoring is left to the user, since his understanding of good design might contradict the metrics used as part of the fitness function.

The main contributions of this paper are

- describing a novel search-based approach for refactoring a software system's class structure and
- evaluating the proposed approach using an open-source case study

The rest of this paper is structured as follows: section 2 describes our approach in detail, with focus on the model, the representation, the fitness function and the refactoring operations we use. In section 3 we present the results of an experiment conducted on a well-known open source case study. Section 4 contains a brief summary of related work and we conclude in section 5.

2. APPROACH

One of the biggest challenges when optimizing class structures using random refactorings is to ensure behavior preservation. One has to take special care of the pre- and postconditions of the refactorings [15], e.g. it is not possible to move methods that override a method of a superclass without moving the overridden method as well. These constraints are much harder to fulfil compared to the constraints that apply to optimizing subsystem decomposition, where name clashes and visibility violations are the only problems that can occur and which are easy to solve by comparison.

Figure 1 gives a short overview of the workflow of our approach. The source code needs to be transformed into a suitable model - our phenotype - using standard fact extraction techniques. It allows simulating the source code refactorings and calculating the impact of these refactorings on the fitness function. Our genotype consists of the already executed refactorings.

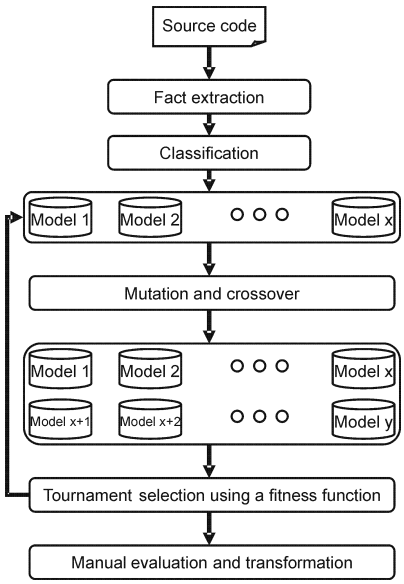

Figure 1: Workflow of the proposed approach

It is necessary to differentiate the model elements according to the role they play in the system's design before trying to improve the structure. We call this step classification. Not all elements can be treated equally, because e.g. design patterns sometimes deliberately violate existing design heuristics. This means that we cannot assess them properly using a generalized fitness function. Currently we restrict our approach to those elements that respect general design guidelines. Elements that deliberately do not respect them are left untouched in order to preserve the developers conscious design decisions.

We consider multiple models which form our population at a time. The initial population is created by copying the model extracted from the source code x times. During one evolution step, elements of the current population are modified using a model refactoring, or a new model is created by combining the genomes of two parents using a crossover operator. In order to reduce our population to the initial size of x elements we select the most promising elements using our fitness function and a tournament selection strategy [1]. Since the already applied model refactorings are stored as our genotype a user can easily find out later which refactorings he needs to carry out on the source code in order to improve the structure.

The optimization stops after a predefined number of evolution steps. This number is chosen based on the number of model elements and the number of refactorings being used. The user needs to inspect the resulting structure manually, because he has to decide which refactorings to apply to the source code.

2.1 Phenotype

As already mentioned in the section above, our phenotype consists of the abstract source code model and several model refactorings These model refactorings simulate the actual source code refactorings.

2.1.1 Source code model

Our source code model is designed to accommodate several object oriented languages. The basic model elements are classes, methods, attributes, parameters and local variables. In addition to these elements, we need a special element called *access chain*, which will be described below and the following access types:

- class access: A class can access another class directly, e.g. if it inherits from the other class.
- method access: A method access models that a method of another class is called.
- attribute access: An attribute access is part of an access chain and models that an attribute is referenced in the method body of a method.
- local variable access: A local variable access occurs if a local variable is referenced inside a method.
- parameter access: A parameter access models that the parameter is referenced inside a method.
- operator access: An operator access models the use of operators like *++*.
- this access: Models explicit references to the *this* object, like *this.getMovie()*.
- super access: Models explicit accesses to the super class, like *super.getMovie()*.
- literal access: Models references to literals like '7'.

An *access chain* models the accesses inside a method body, because we need to adapt these references during the optimization. If a method is moved, we need to change the call sites. An access chain therefore consists of a list of accesses of the above types. In general, each statement is mapped to a separate access chain. If the statement is an assignment, it is modeled by two access chains, which represent its left and right side. Access chains are hierarchical, because each method argument at a call site is modeled as a separate access chain, that could possibly contain further access chains. During the fact extraction we additionally compute whether an access chain is a read or a write access.

An example model representing the source code shown in Figure 2 is depicted in Figure 3. The class *C* contains the method *calcArea(Rectangle r1)* that has one read access chain consisting of two parameter accesses and two method accesses. Class *C* also contains the method *foo(Rectangle r2)* that has one read access chain consisting of one method access and one parameter access.

```
class C {
...
int calcArea(Rectangle r1) {
    return r1.height()*r1.width();
}
...
void foo(Rectangle r2) {
    calcArea(r2);
}
...
}
```

Figure 2: Example code

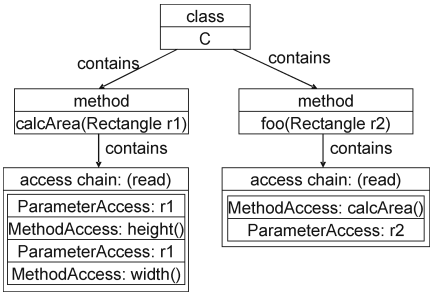

Figure 3: Example model

Things we currently do not take into account and therefore leave unchanged are initialisers, inner classes and synchronisation points. Additionally, we assume that the system being processed does not make use of reflection.

2.1.2 Model Refactorings

Our model refactorings are based on the source code refactorings that can be found in [8]. The model we proposed in section 2.1.1 allows to simulate most of the important refactorings for changing the class structure of a system, which are *Extract Class*, *Inline Class*, *Move Attribute*, *Push Down Attribute*, *Pull Up Attribute*, *Push Down Method*, *Pull Up Method*, *Extract Superclass* and *Collapse Class Hierarchy*.

In general these refactorings cannot be applied automatically and require user information in order to be carried out completely. For example, it is not possible in general to decide, whether an attribute can be moved or not, and it is required that a user specifies the new object on which a moved method should be called.

But since we are proposing a search-based approach, we cannot require user input for every refactoring we simulate. Therefore, we have to restrict these general refactorings to special refactorings, which are independent from user input.

Currently, we have successfully defined and implemented special refactorings for *Move Method*, *Pull Up Attribute*, *Push Down Attribute*, *Pull Up Method*, *Push Down Method*. For the remainder of this paper we focus on the *Move Method* refactoring.

2.1.3 Special refactoring for moving methods

For moving methods we consider the following two strict preconditions: At first, target and source class are not part of the same inheritance hierarchy. Otherwise you would use *Pull Up Method* or *Push Down Method* . At second, the method to be moved must not implement a method of an interface or overwrite a method of a superclass.

We do not take into account further preconditions concerning visibilities and names, since a violation can be resolved easily after the optimization, by a simple *Rename* refactoring or by calculating the necessary visibility of model elements and adjusting it.

In order to successfully move a method in the general case it is necessary to manually specify for all accesses an object with the type of the target class on which the method should be called. A user of course can decide whether to use an already existing object, or to create a new one. In order to allow automatic processing, we need to make sure that suitable objects already exist at the call sites. This is achieved by considering the following three special cases of the move method refactoring which allow to reuse already existing objects at the call sites:

- O1: Move a method to one of its parameter types. The type must not be a primitive type and its type definition must be available as source code.
- O2: Move a method to a type of a randomly chosen attribute of the class the method belongs to. The type must not be a primitive type and its type definition must be available as source code.
- O3: Move a static method

Besides allowing an automatic transformation, the first two special cases of *Move Method* constitute a good heuristic for speeding up the search process. It does not seem to make sense to move methods to classes, which are not already related to the original class, because this would introduce new dependencies. Checking these preconditions is done before the optimization starts and leads for every type of refactoring to a list of model elements that could be processed. These checks could be carried out during the optimization as well, but doing it beforehand and storing the information as part of the representation saves time and therefore speeds up the optimization.

To apply a refactoring to the model, we have two possibilities. Either we provide the complete source object as an additional parameter (A1) or we provide only the elements of the source object which are read only by the method as

additional parameters (A2). The second alternative reduces dependencies between source and target classes, but is not applicable as often as the first one due to its more restrictive preconditions. After changing the method's signature, we additionally have to adapt the call sites and the access chains inside the methods. At the call sites, we need to replace the old object references used to call the method with references to an object of the type of the target class. Inside the method, we need to replace all accesses to surrounding class members with an access using the newly introduced method parameter.

To give a short example we show in Figure 4 how the method *calcArea(Rectangle r1)* that has been introduced in Figure 2 could be moved the class *Rectangle*, which is its parameter type.

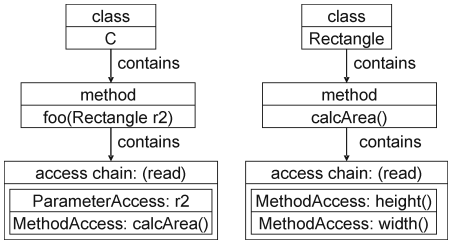

Figure 4: Example model refactored

The parameter *Rectangle* can be deleted, since we moved the method to this class. Inside the method the accesses to the parameter object *r1* can be removed. At the call site, we have to modify the existing access chain such that the parameter *r2* is now used to call the method *calcArea()* instead of being used as a parameter.

Figure 5 shows how the source code would look like, if the proposed refactoring would be applied to it.

```
class C {
...
void foo(Rectangle r2) {
    r2.calcArea();
}
...
class Rectangle {
...
int calcArea() {
    return height()*.width();
}
...
}
```

Figure 5: Refactored example source code

To allow a free movement in the search space, it should be possible to move the methods back to their original place. For methods that have been moved using O3 or O1/A1, this is automatically possible. For the other mutations, special inverse transformations have been designed, avoiding the creation of unnecessary backreferences to the original classes.

Further special refactorings have already been specified but not implemented yet. These include *Move Attribute*. It is only allowed to move attributes between classes that have been previously split using extract class, because this assures that there is a one to one relationship between corresponding objects at runtime.

2.2 Genotype

Our genotype consists of an ordered list of executed model refactorings including necessary parameters. The current phenotype is created by applying these model refactorings in the order that is given by the genotype to the initial source code model. Therefore the order of the model refactorings is important, since one model refactoring might create the necessary preconditions for some of the following ones.

2.2.1 Mutation operator

Our mutation operator extends the current genome by an additional model refactoring. This means that our genome grows during the optimization and the length of the genome is unlimited. For performance reasons we store the current phenotype together with the current genotype, because otherwise we would have to execute all previous model refactorings before being able to choose a new one.

2.2.2 Crossover operator

Our crossover operator combines two genomes by selecting the first n model refactorings from parent one and adding the model refactorings of parent two to the genome. N is randomly chosen. We can be sure that the n model refactorings of parent one can be executed safely, but not all model refactorings of parent two might be applicable. Therefore we apply the model refactorings to the initial source code model. If we encounter a model refactoring that cannot be executed due to unsatisfied preconditions, we drop it.

An example crossover is depicted in Figure 6. At first, three elements of P1's genome are combined with P2's genome resulting in an intermediate genome PI. While applying PI's model refactorings to the original source code model it turns out, that the model refactorings M6 and M7 cannot be executed.

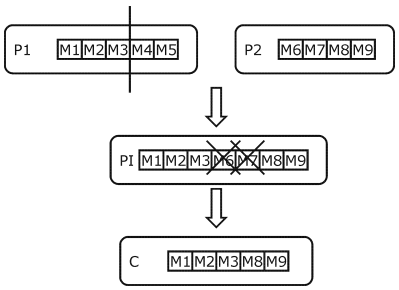

Figure 6: Example application of our crossover operator

The advantage of this crossover operator is that we can guarantee that the externally visible behavior is not changed. The drawback is that it takes some time to perform the crossover, since the refactorings need to be simulated again.

2.3 Fitness Function

Our fitness function is a weighted sum of several metric values and is designed to be maximized. The properties we would like to capture with it are coupling, cohesion, complexity and stability. Usually it is desirable to

have low coupling values between classes, in order to keep classes independent of each other. If classes are not cohesive, it is likely that a class captures more than one concept and therefore some methods should be moved to other classes. For coupling and cohesion, we rely on existing metrics, which are taken from Briand's catalogues [4]. Coupling is assessed by computing the *Response for class* (RFC) and the *Information-flow-based-coupling* (ICP) metrics. Cohesion is assessed using the *Tight class cohesion* (TCC), the *Information-flow-based-cohesion* (ICH) and the *Lack of cohesion* (LCOM5) metrics. These metrics are computed globally for the whole system by simply adding up the values for the individual classes.

The motivation for using complexity metrics as part of our fitness function is that software systems very often contain so called god classes [13]. They contain a lot of methods that should be moved to so called data classes which are associated to the god classes and provide almost no functionality. It is desirable to distribute a system's complexity equally between classes. To assess complexity we use two variants of the *Weighted method count* (WMC) metric. One with the cyclomatic complexity and one with the complexity value one. In the second case the WMC metric would result in the *Number of methods* (NOM) metric. In order to normalize these metrics we defined a trapezoid shaped function that maps the complexity values to a value between zero and one. Complexity is considered to be optimal inside an interval a_{min} and a_{max}. Outside an interval b_{min} and b_{max} it is considered to be zero. Between b_{min} and a_{min} and b_{max} and a_{max} it is linearly interpolated. The advantage of such a fitness function is its fuzzy shape, which does not punish complexity values too hard, if they are only slightly outside the specified optimal values. Currently we are using the following values for the NOM metric: $a_{min} = 10$, $a_{max} = 40$, $b_{min} = 3$ and $b_{max} = 50$. Of course these values can be customized by the user.

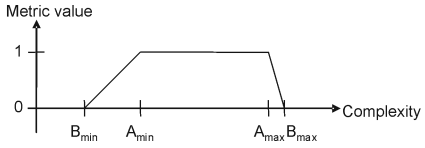

Figure 7: Complexity metric

The metric for stability has been adapted from the reconditioning of subsystem structures [12]. Using this metric one tries to achieve a layered class structure, so that each class only depends on classes that are more stable than itself. At the same time, this metric punishes cyclic dependencies between classes.

The formula for calculating the instability of a class k is:

$$I(k) = \frac{C_e}{C_e + C_a}$$

C_e is the number of other classes whose elements k uses. C_a is the number of classes using elements of k.

Since not all of our metrics are normalized it is not possible to just add up all values in order to obtain the fitness value for a software system. The fitness value of a system S is calculated by comparing the actual metric values $M(S)$ with the initial value $M_{init}(S)$ and maximum values $M_{max}(S)$ obtained by a calibration run optimizing each metric alone beforehand.

The metric values of the software system are kept as an array $M(S)$ = [WMC(S), $RFC(S)$, $ICP(S)$, $TCC(S)$, $ICH(S)$, $LCOM5(S)$, $ST(S)$]. The current fitness value $fitness(S)$ is calculated with the help of $M(S)$, $M_{init}(S)$ and $M_{max}(S)$.

$$fitness(S) = \sum_{i=1}^{n} w_i * \frac{M_i(S) - M_{init_i}(S)}{M_{max_i}(S) - M_{init_i}(S)}$$

Using the weights w_i a user can decide to focus on certain aspects of the fitness function. The sum of all weights has to be one.

2.4 Classification

Before optimizing the structure, we classify the model elements according to the roles they play in the system's design, e.g. whether they are part of a design pattern. We need to do this for two reasons. On the one hand, there are many design patterns, which deliberately violate existing design guidelines, e.g. the methods belonging to a facade are usually not cohesive, since they only forward incoming requests to other classes that can fulfill these. Currently such model elements cannot be assessed correctly by our fitness function and therefore are left unchanged in order to preserve the original design decisions. Of course our approach could be extended by other special refactorings which could try to split up a huge facade, but this is out of the scope of this paper.

On the other hand, there are a lot of elements, which are not worth to be considered separately, because they only provide helper functionality for other elements. Getter methods for example are usually small methods that simply expose the state of an object to the outside of a class. They could possibly be moved, but it does not really make sense to move them without the attributes they encapsulate and the corresponding setter-methods. Using this information can speed up the optimization, since it is not necessary to try to move the getter- and setter methods alone.

In order to classify the existing model elements we are using an approach similar to the one described in [2]. The main idea to identify the role of an element is to describe its static structure and additionally make use of naming conventions. As an example, static factory methods are described as being static, having a non-void return type, and containing a call to a constructor that creates objects whose type is equal to the return type of the method. The specification and detection of such roles is implemented as a Java-Library.

At the moment we focus on roles of methods, because we restricted ourselves to the move method refactoring for the remainder of this paper. But of course we are able to identify candidates for design patterns which involve classes as well, e.g. facades, factories, proxies, observers and visitors.

We identify the following types of helper methods:

- getter and setter methods: These methods are strongly coupled to the attributes they get or set. They should not be moved to another class alone.

- collection accessors: Collection accessors are simple methods that do not contain a lot of statements. They simply add some new objects to the surrounding class. They are tightly coupled to some attribute and therefore do not need to be moved around without the attribute.

- state methods: State methods perform simple checks that describe the state of an object. They are tightly coupled to the class and its attributes.

Additionally, we identify factory methods and delegation methods. Several factory methods are often bundled in a class of their own. They create an instance of a certain type but should not necessarily be moved to that type, even if they use no attributes of the surrounding factory class and only methods and attributes of the types instantiated. Factory method is a design pattern itself. Delegation methods do not provide much functionality. They simply forward requests to classes capable of fulfilling them. Therefore using standard coupling and cohesion metrics, delegation methods tend to be moved to the classes doing the real work. Delegation method is not a design pattern itself, but used as a part of many other design patterns like e.g. facade.

3. EVALUATION

In this section we present a first evaluation of our approach. Currently only the move method refactoring and the crossover operator are considered. We used the open source software JHotDraw as a case study, which is a framework for creating technical drawings. It is well-known as a good example for the use of design patterns.

Table 1 gives an overview of the system's size:

Subsystems	11
Classes (inner)	275(64)
Methods	2571
Attributes	489
LOC	28776

Table 1: JHotDraw statistic

Our experiments were carried out on a PC at 3 GHz with 1 GB of RAM. For each run we used a population size of 30, 4000 generations, a mutation probability of 80 % and a crossover probability of 20 %. One run took about 30 minutes. In order to judge how stable our results are, we carried out the optimization 10 times. The first goal of our evaluation was to to find out whether our approach could find meaningful refactorings in the original system structure. Our second goal was to to show, that our approach was able to move manually misplaced methods back to their original positions.

First we examined how restrictive our preconditions for moving methods are and how many methods are not considered during the optimization because of their classification as special elements. Table 2 shows the results of our classification. 724 methods out of the 2571 overall methods are classified as special methods, which means that they are not processed by our approach. The results of the classification have been manually inspected by us. We did not find any false positives among the getter/setter/collection/delegation and state methods. But we found eleven false positives among the factory methods. It is not crucial to identify all false positives, because our approach is still able to process a case study. Having false positives just means, that some methods, that could be moved in theory are left untouched and therefore the result of an optimization could have been even better. False negatives are not that important either, because if our approach fails to identify some design pattern methods during the classification we would discover them later on during the manual inspection of the proposed refactorings. On the whole, our classification approach turned out to be successful.

Getter methods	205
Setter methods	176
Collection methods	49
State methods	56
Factory methods	135
Delegation methods	103
Pattern methods	724
Movable methods	718
Movable non pattern methods	260
Overall methods	2571

Table 2: Method statistic

Table 2 additionally gives an overview of the number of methods that can be considered during our optimization. The precondition check results in 718 movable methods. After combining these results with our classification information, we are still able to move 260 methods around. This means that the classification significantly reduced our search space, but there are still too many methods to examine them manually without being assisted by a tool and therefore a relevant number of methods can be moved.

Figure 8 shows the results of our series of optimizations. The lowest line represents the minimum fitness value, the middle line the median fitness value and the highest line the maximum fitness value for each generation of all 10 runs. Convergence is pretty good and statistical spread is low. You can observe that after approximately 2000 generations the fitness value does not significantly change any more.

As expected our approach is able to find refactorings that improve the fitness value of JHotDraw's system structure. Table 3 shows the average relative improvement of the individual metrics, obtained by comparing the final values to the maximum values $M_{max_i}(S)$ calculated during the initial calibration runs:

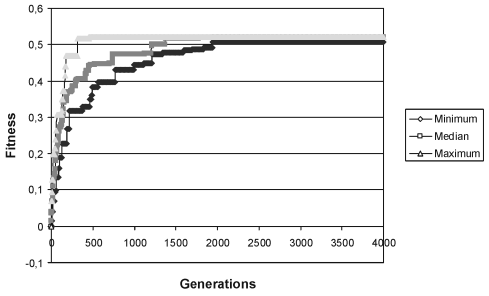

Figure 8: Optimisation results

$$\text{relative improvement} = \frac{M_{final_i}(S) - M_{init_i}(S)}{M_{max_i}(S) - M_{init_i}(S)}$$

A value of 0 % means that this metric could not be improved during the optimization. A value of 100 % means that our approach proposed refactorings that resulted in a final metric value equal to the one obtained during the calibration.

The results just presented show that at least with respect to our fitness function JHotDraw offers some potential to be

WMC	NOM	LCOM	TCC	Sta	ICH	ICP	RFC
3%	5%	33%	31%	66%	81%	87%	3%

Table 3: Relative improvement

refactored. In order to judge whether the refactorings make sense to a developer, we manually inspected the proposed refactorings. This is of course not easy, since we were not involved in the original design of JHotDraw. The names of the methods that were proposed to be moved are shown in the first column of Table 4. The suggested target class is shown in the second column. The third column contains the number of runs this method has been proposed to be moved. 10 out of the 11 proposed refactorings have been found in every run, which indicates that our results are pretty stable.

Method	**Target class**	**#**
PolygonFigure.chop	Geom	10
AttributeFigure.initializeAttributes	FigureAttributes	10
FigureAttributes.read	StoreableInput	10
FigureAttributes.write	StoreableOutput	10
FigureAttributes.writeColor	StoreableOutput	10
ShortestDistanceConnector.findPoint	Geom	10
TextTool.fieldBounds	FloatingTextField	10
PertFigure.readTasks	StoreableInput	10
PertFigure.writeTasks	StoreableOutput	10
ColorMap.color	ColorEntry	2
ColorMap.colorIndex	ColorEntry	10

Table 4: Methods proposed to be refactored

From our perspective all proposed refactorings can be justified. Consider for example the static method *Point chop(Polygon poly, Point p)* of class *PolygonFigure* that has been suggested to be moved to the class *Geom*. This method chops a polygon using a specified point. As you can observe from its source code shown partially in Figure 9 it makes heavy use of the methods *length2() and intersect()* of class *Geom* but does not use any attributes or methods of the class it belongs to. The purpose of this method is to provide utility functionality that is best kept in the class *Geom* which is part of the utility package.

```
for (int i = 0; i < poly.npoints; ++i) {
int nxt = (i + 1) % poly.npoints;
 Point chop =
   Geom.intersect(poly.xpoints[i],
                  poly.ypoints[i], ...);
if (chop != null) {
   long cl =
   Geom.length2(chop.x, chop.y, p.x, p.y);
 }
...
```

Figure 9: Example code

But for a few of the proposed refactorings we could also think of design decisions why the suggested refactorings have not been already carried out by the original designers. The class *AttributeFigure* for example contains a method called *initializeAttributes* which our approach proposed to be moved to the class *FigureAttributes* because it operates only on the attribute *fAttributes* which is of type *FigureAttributes*. By moving this method the knowledge to create default attributes is transferred to the class that represents the attributes, which is reasonable from our perspective. From a metrics perspective moving this method reduces the coupling between *AttributeFigure* and *FigureAttributes*, because the calls to the set method are now performed in the class *FigureAttributes*. But on the other hand, the original designers might have preferred to leave the class *FigureAttributes* ignorant of any default combinations that are meaningful to the class *AttributeFigure*.

As a second goal, we wanted to show that if we modified the original system by randomly selecting 10 methods and misplacing them, our approach would be able to suggest refactorings moving them back their original position. Since JHotDraw is generally considered to be well designed, we assumed that its original design could serve as a reference design. Therefore, we did another series of 10 runs using the modified system.

The names of the manually misplaced methods are listed in the first column of Table 5. The second column contains the name of the class the methods were manually moved to. The third column shows the number of runs, the specific method has been moved back to its original position.

Method	**Misplaced to class**	**#**
MDIDesktopManager.resizeDesktop	MDIDesktopPane	10
Geom.distanceFromLine	PolygonFigure	1
Geom.ovalAngelToPoint	ChopEllipseConnector	10
ImageFigure.drawGhost	FigureAttributes	10
PolygonFigure.findSegment	FigureAttributes	10
DiamondFigure.getPolygon	FigureAttributes	10
AttributeFigure.writeObject	FigureAttributes	10
CompositeFigure.bringToFront	QuadTree	10
CompositeFigure.findFigure	QuadTree	10
PertFigure.layout	Quadtree	10

Table 5: Manually misplaced methods

Our approach successfully moved back each method at least once, meaning that it was able to restore the original reference design. Nine out of the ten methods even have been moved back during each run, which prove that the results are very stable.

Our evaluation revealed that our preconditions for our special refactorings still allow to move a relevant number of methods and that our approach is able to suggest meaningful refactorings. Furthermore, it turned out that our approach is able to move manually misplaced methods back to their original position. As part of a first evaluation, these results are very convincing and promising.

4. RELATED WORK

The general idea to treat software engineering as a search problem is described in [5]. Search-based algorithms have already been successfully used in order to recover and improve the decomposition of software systems into subsystems [6].

Existing approaches for improving class structures can be roughly classified into three different categories. Approaches of the first category consider optimizing class structures as a manual analytical process. The main idea is to find one problem and to fix it without considering other problems. T. Dudzikan and J. Wlodka present an integrated approach to restructure programs written in Java [7]. Starting from a catalog of bad smells, potential solutions are proposed to

the user. L. Tahvildari and K. Kontogiannis [17] present an approach for aiding the user in deciding which refactoring he should apply. Their main idea is to estimate the impact refactorings have on some elementary metrics. How the refactorings can be really carried out is not considered by this approach. Another approach that analyzes the impact refactorings have on metrics is described in [3]. They point out that one refactoring can have a different impact depending on the context in which it is applied. How these refactorings can be used in order to improve the structure is not examined. Compared to our approach all of these approaches do not take side effects of refactorings into account. They focus on one part of the system and one structural problem at at time.

The approaches that fall into the second category improve a system automatically but with respect to only one special design guideline. Lieberherr [10] for example presents an approach to transform systems into systems that conform to the *Law of Demeter*. Approaches of this category do take side effects of refactorings into account, but only for the special problem they try to tackle. The scope of these approaches is more limited than the scope of our approach, since we address a number of refactorings and bad smells at a time.

Methodologies that fall into the last group are similar to our approach and can be described as approaches that try to improve a system's structure using a search-based algorithm and random transformations. One such approach is described in [14]. The authors' only goal is to improve the structure of inheritance hierarchies, which means that our approach is more general.

5. CONCLUSION

In this paper we presented an approach that proposes a list of refactorings that help a software engineer to improve the class structure of a software system. The application of these refactorings leads to a behaviorally equivalent system structure with better metric values and fewer violations of object-oriented design heuristics.

Our major contribution to the existing state of the art is describing a novel search-based approach for refactoring a software system's class structure and evaluating the proposed approach using an open-source case study.

Our approach uses a special model which allows us to simulate refactorings with all necessary pre- and postconditions, a mechanism for classifying structural elements according to their role, mutation and crossover operators and a fitness function. We demonstrated the potential of our approach by applying it to the open-source case study JHotDraw and believe that it forms a convincing foundation for further research in this area.

In order to further improve and extend our approach we plan to implement further refactorings, to exploit more heuristics on good class design and to validate our approach using further case studies.

6. REFERENCES

[1] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic Programming - An Introduction*. Morgan Kaufmann Publishers, Inc., 1998.

[2] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of oo systems. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 3–14. IEEE Computer Society, 2004.

[3] B. D. Bois and T. Mens. Describing the impact of refactoring on internal program quality. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications*, pages 37–48. Vrije Universiteit Brussel, 2003.

[4] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, 1999.

[5] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *Journal of IEE Proceedings - Software*, pages 161–175, 2003.

[6] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *IEEE Proceedings of the 1999 Int. Conf. on Software Tools and Engineering Practice (STEP'99)*, 1999.

[7] T. Dudzikan and J. Wlodka. Tool-supported discovery and refactoring of structural weaknesses. Master's thesis, TU Berlin, 2002.

[8] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999.

[9] M. Lehman and L. Belady. *Program Evolution, - Processes of Software Change*. Academic Press Professional, Inc, 1985.

[10] K. Lieberherr. *Adaptive Object-Oriented Software - The Demeter Method - With Propagation Patterns*. PWS Publishing Company, 1995.

[11] R. Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of Technology of Object-Oriented Languages and Systems - Tools 39*, 2001.

[12] R. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[13] H. McCormick and R. Malveau. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1998.

[14] M. O'Keeffe and M. O. Cinnèide. Towards automated design improvement through combinatorial optimisation. In *Proceedings of the Workshop on Directions in Software Engineering Environments*, 2004.

[15] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[16] O. Seng, M. Bauer, M. Biehl, and G. Pache. Search-based Improvement of Subsystem Decompositions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2005.

[17] L. Tahvildari and K. Kontogiannis. A metric based approach to enhance design quality through meta-pattern transformations. In *Proceedings of the seventh European Conference on Software Maintenance and Reengineering*, 2003.