PARSE - An AI Planning Assistant for Refactoring SEquences

Abstract

We investigate an open issue in refactoring, namely, the ordering of a sequence of refactorings (conflicts and dependencies amongst the refactorings), and propose a novel solution to the problem via the usage of a partial order planner from the field of Artificial Intelligence. We formulate the problem as an AI planning problem and use AI planning algorithms to come up with a suitable plan i.e. a suitable ordering among the different refactorings chosen by the developer. The tool that we develop is called PARSE - An AI Planning Assistant for Refactoring SEquences.

1 Introduction

At its heart, Software Refactoring [Fow99] simply refers to the modification of the code, to improve its internal structure whilst preserving the external functionality. This *restructuring* essentially helps in the onerous task of software maintenance. However, the actual task of refactoring is not easy and in the absence of good tools can quickly become tedious, error prone and time consuming. Since refactoring largely remains an additional activity carried out by the developer, he may not be willing to do it, unless it can be done easily and the resulting code is correct. Thus tool assisted refactoring techniques has seen fairly active research in recent times. In this proposal we present an extant open problem in this area and a novel solution to it.

Currently, we believe that the following are some of the open problems in the area of tool assisted refactoring techniques.

1. **Correctness of refactored code:** The code produced due to refactoring by tools like Eclipse [IBM05] is sometimes incorrect. The problem is particularly insidious as the code produced is *correct* in a majority of the cases, which may lull the developer into a false sense of security. The problem occurs because the tools do not perform sophisticated data flow analysis when doing refactoring.
2. **Ordering among a set of refactorings** When confronted with a set of refactorings which can be applied to the code, the developer is currently not given any assistance in deciding upon the ordering among them. It could be that some refactorings are dependent on other refactorings. Conversely, some refactorings may preclude the application of other refactorings. This is the open problem that we address in this paper.
3. **Propagating refactoring changes to other software artifacts:** Code changes due to refactoring. However, this change is not reflected in other software artifacts like design specifications. Thus these documents are rendered obsolete, hampering maintenance.
4. **Refactoring test cases:** After refactoring, the test cases might have to be updated. Some test cases would no longer be relevant, while additional test cases may need to be added. Current tools do not handle this scenario.
5. **Refactoring libraries:** The repercussions of refactoring of library code, especially the interface parts is not handled well by the current tools. Currently, the library developer must take a very conservative approach to refactoring the interface related code, since the tools do not inform which refactorings might actually break the client code.

Thus refactoring provides a fertile area for research and the research would be immensely beneficial to the software development industry, allowing savings of millions of dollars due to the decreased maintenance efforts.

1.1 Overview of the goals of our research

In this proposal, we will address one of the open issues stated previously i.e. the ordering among a set of refactorings. Currently, the developer will have to manually determine the order among a set of possible refactorings taking into account conflicts among them and sequential dependencies. As with any other manual process it can be cumbersome, prone to errors and time consuming. Thus an intelligent tool that helps in determining this order is the need of the hour and the aim of our research is to develop such a tool. We will use AI

Planning for this and our tool is named, PARSE (Planning Assistant for Refactoring SEquences).

2 Background and state of the art

Software Refactoring [Fow99] is code modification, to improve its internal structure whilst preserving the external functionality. The primary aim in refactoring is *not* to fix bugs or *add* new functionality, but to *improve* the code. When juxtaposed with the aphorism, ”If it ain't broke, don't fix it,”, one may wonder at the need for refactoring at all. It is a well documented fact that software maintenance i.e. making corrections to fix bugs and making changes to incorporate new features, is that phase of software development which consumes the maximum resources, in terms of money and man hours. Refactoring helps *tremendously* in software maintenance. It can be viewed as a sophisticated cleaning up of code. The main purpose of refactoring is to make the code easier to understand and modify. Understanding existing code and modifying it, is the leitmotif in software maintenance.

Opdyke [Opd92] provided the original definition of refactoring behavior preservation. According to this, for the same set of input values, the resulting set of output values should be the same before and after the refactoring.

We can augment the behavior preservation to include the following aspects as well: The performance of the program before and after the refactoring should remain the same i.e. it should take the same amount of execution time. The memory requirements of the program must *not* increase drastically.

Refactoring is different from both performance optimization and compiler optimizations. Performance optimization tries to improve performance at *any* cost i.e. even at the cost of making the source code more difficult to understand. Compiler optimization occurs at the back-end i.e. intermediate and machine code and is not concerned with the front end i.e. source code.

Refactoring does not also mean a *blind* replacement of all complicated constructs in a program with simpler constructs. For example, replacing a binary search algorithm with the simpler linear search is not refactoring. (The performance in the worst case has degenerated). However, replacing a recursive version of binary search with the iterative version may qualify as refactoring, if the developers are more comfortable with non-recursive implementations.

Refactoring opportunities in code are typically identified by ”code smells”. We will provide a synopsis of the major ”bad smells”. For more details please refer [Fow99]. The major ”smells” are:

1. **Duplicated Code** - the same piece of code being repeated at different sections in the program. This can occur due to the ubiquity of copy-paste.
2. **A very long method** - a method that does a lot of things and hence is long (say it has greater than 50 statements)
3. **A very large class** - a so called ”God” class that represents too many things.
4. **A function with a very long parameter list** - This indicates that the method is perhaps too complicated and hence must be refactored.
5. **Speculative Generality** - an over-engineered design, which has too much polymorphic behavior for future use, that may never arise.

2.1 Classification of refactorings

We present a classification (catalog) of refactorings as described in [Fow99], this will help in understanding the example that is presented in the next section. Again, please refer to [Fow99], for the complete set.

1. **ExtractMethod** - Here a piece of code that can grouped together is extracted and made into a method of its own, thus promoting reusability.
2. **Pull Up Method** - This typically occurs in an inheritance hierarchy due to copy-paste. Functions in the sibling derived classes have the same code and hence can be ”pulled up” to the base class.
3. **Rename method** - a method is named in an obscure manner using abbreviations and acronyms, this can be ”renamed” to something that clearly indicates its functionality.
4. **Replace magic number with symbolic constant** - A magic number like 12 is used instead of a more clear symbolic constant like NumberOfMonths.
5. **Form template method** - Introduce the template design pattern [GHJV95], by letting derived classes provide specific variants of an algorithm.

2.2 Benefits of refactoring

The following describes the many benefits of refactoring [Fow99].

1. **Software design improvement:** A primary casualty of bug fixes and feature enhancements in software, is its design. By refactoring, we can ensure that the design does not decay.
2. **Making software easier to understand:** As has been stated previously, this is the primary aim and benefit of refactoring.
3. **Finding bugs:** As we refactor code, we would be looking at it deeply to find refactoring opportunities. During this process we could also uncover dormant bugs in the code and fix them.

2.3 State of the art

In this section, we focus on the research that has taken place to address the refactoring challenges described previously. We focus primarily on the research problem that is addressed by us i.e. issues in a sequence of refactorings. We also briefly describe other research in the field of refactoring.

Mens et al. [MTR05, MTR06] address the following problem which is closely related to the problem discussed in this paper. During refactoring, a developer may be confronted with a number of refactoring options (these options may be found manually by him or suggested by a tool). The refactorings may have dependencies between them and it could also be the case that a particular refactoring precludes the application of another refactoring (conflict). Further a particular refactoring may be feasible only after the application of another refactoring (Sequential dependency). If the number of refactoring options are large, then it may be difficult for the developer to choose among these options and also to choose an optimal set of refactorings.

For this problem, they use critical pair analysis to detect the conflicts among refactorings and sequential dependency analysis of graph transformations. They represent the program as a type graph and the refactorings upon it as graph transformation rules. Refactoring pre-conditions are also included. The main drawback of their method is as follows: Critical pair analysis can be very time consuming. Sequential dependency analysis is *not* handled by their tool and must be done manually.

The same authors in an earlier work [MEDJ05], provide the formal theoretical foundation for representing refactorings via graph transformation rules. They conclude that programs can be represented by graphs and in such a scenario, refactorings are essentially graph transformation rules and can be defined by graph rewriting rules.

In the following paragraphs, we present some other research done in different areas of refactoring: Automatic identification of "bad smells" in code to detect candidates for refactoring. Higo et al. [HKKI04] use code clone analysis to detect duplicated code fragments (clones), which as described earlier are one of the first smells to look for while refactoring. But their tool is restricted to clones and even within that it does not identify all clones. Another interesting work in this area is by Kataoka et al, [KEGN01] who use the concept of program invariants to identify candidates for refactoring. In the absence of explicit invariants, their tool DAIKON infers the invariants. They can identify smells like unused parameters in a method which suggests the need for RemoveParameter refactoring. A drawback is that they use dynamic program analysis, and we know the traditional shortcomings of dynamic program analysis i.e. it may not be representative and is biased towards particular runs.

A different area of research deals with the identification of refactorings after they are done. Given source code differences between any two versions of a program, there could be a number of changes due to bug fixes/feature addition/refactoring. Which of these can be classified as actual refactorings? Weissgerber and Diehl [WD06], use signature based analysis and clone detection (token based) to detect and rank refactorings. (via code clone analysis). Demeyer et al. [DDN00] also address the issue by using a set of heuristics (4) based on change metrics (The Chidamber and Kemerer [CK94], Lorenz and Kidd [LK94]) to detect which of the changed code can be classified as refactoring. But their excessive reliance on metrics alone lead to a low success rate of identification of actual refactorings.

An issue in tool assisted automated refactoring is the correctness of the code produced after refactoring. Verbaere et al. [VEdM06] demonstrate some bugs in the refactored code [EV05]in the current IDEs like Microsoft Visual Studio [Mic05], Eclipse [IBM05] and propose methods to overcome it. They define a new scripting language JUNGL in which refactorings can be expressed and utilize data flow analysis and concepts from compiler optimization. Although their method seems to overcome the bugs in refactored code as compared with tools like Eclipse, it is very complex.

The problem of applying refactorings in libraries and evolving APIs has been addressed by Henkel and Diwan [HD05] with their tool CatchUp! that captures the refactorings done by a library developer. The client code developer then can use the same tool to replay the refactorings and make required changes in his code to "catchup" with the evolved library. The main drawback is that not every one may use the same IDE to do the refactorings and so the synchronization between library developer and library user becomes the conflicting point. Further, developers may forget to turn on the recordings when doing refactorings.

3 Challenges and goals

In this section we describe the open issue for which we propose a novel solution, in the next section. The open issue is as follows: Consider a developer who has decided that he wants to refactor his code. He would open the project in an IDE like Eclipse [IBM05]and try to detect some smells in the code which make it amenable for refactoring. Detecting *which* particular smell exists in code and hence which refactoring must be applied, is *not* our focus here. We assume that either this is done manually by using human intuition or by any one of the existing tools to help detect smells in code [HKKI04, KEGN01, SSL01].

Once the developer has decided upon a set of refactorings to apply to the code, the question that arises is: In what order must he apply the different refactorings? Or, does the order matter at all? i.e. applying the set of refactorings in any order is the same. This is the crux of the problem that we address in this paper. It turns out that the order does indeed

matter. More importantly, application of a certain refactoring may preclude the application of another refactoring. This is called conflict among refactorings. Further, a refactoring may be possible only after the application of another refactoring before it (sequential dependency).

Thus it would be helpful to the developer if an automatic ordering amongst the refactorings that he has chosen to incorporate, was provided. In the absence of this, he will have to either find the suitable order himself, or apply it in a random order. In the former case, it involves considerable additional time to be spent to find the suitable order. In the latter case, of applying the refactorings randomly, he may miss a potentially more beneficial refactoring because of the fact that it has been precluded due to an earlier refactoring.

3.1 Example

In this section, we describe a simple example to illustrate the problem that we address i.e. refactoring sequence more clearly. The figure 1 shows an inheritance hierarchy modeling an instructor at a typical American University. Specifically, we have an Instructor base class, with derived classes for an AssistantProfessor, AssociateProfessor and Professor. We are interested in the virtual function ComptSal, which computes the salary. The unrefactored code is shown in figure 2.

There are several ”smells” in this piece of code which makes it amenable to different refactorings. Firstly, the method name does not clearly reflect its functionality and hence it can be renamed to something more clear like ComputeSalary via the RenameMethod refactoring. The piece of code within this method that does the tax calculation can be extracted into another helper method, ComputeTaxDeduction via the ExtractMethod refactoring. Then we have magic numbers denoting the tax rates and these can be turned into symbolic constants via the Replace magic number with symbolic constant refactoring. Finally, the method is almost the same in all the three derived classes and hence can be moved up into the base class, via the Pull Up Method refactoring. The slightly specific portions related to the derived classes i.e. the basic salary for different types of instructors can be handled by introducing the Template design pattern [GHJV95].

Thus the applicable refactorings are: RenameMethod, ReplaceMagicNumber With SymbolicConstant, ExtractMethod, IntroduceTemplateMethod and PullUpMethod. Having decided upon this set of refactorings, the next question the developer would have is the order in which the refactorings must be applied.

Order 1: Suppose he chooses the following order for one of the derived classes, say Professor::ComptSal

1. RenameMethod
2. ExtractMethod
3. ReplaceMagicNumber

Now, with this set of changes, he can no longer apply the PullUpMethod from the derived classes, since the method name and body would not be the same across all the derived classes. Thus the RenameMethod and ExtractMethod *precluded* the application of PullUpMethod refactoring.

Order 2: Consider applying the refactorings in the following order:

1. PullUpMethod
2. RenameMethod
3. ReplaceMagicNumber
4. ExtractMethod
5. Form TemplateMethod.

This order ensures that there is no conflict between any elements amongst the set of refactorings and we can apply all of them. The final refactored code is shown in figure 3.

Comparison of the different orders: It also turns out that *subjectively* the resulting code due to order 2 is better than the resulting code due to order 1. This is because, the code for the salary computation and tax deduction is consolidated at one place, which makes the code easier to understand and modify. Suppose the tax rates change (a typical scenario) and we have to implement the ComputeSalary according to the new tax rate, then the change needs to be done only at a single place. Notice also that we have also introduced the template design pattern [GHJV95], for the basic salary got by each type of an instructor. Thus the refactoring is not merely cosmetic but has substantially cleaned up the code and made it use well known design patterns, which is always a good practice.

Response Time Any solution that solves this ordering problem must have a fast response time since it should be remembered that currently, developers do not have a lot of time to do refactoring and hence would like automated, quick and accurate solutions

4 Proposed research

4.1 Planning

Since our solution to the problem entails the use of AI planning, we describe it briefly here. The interested reader can refer to the [RN95] for more details. As we describe the concept of planning, please notice how *remarkably similar* it is to the problem of finding the ordering among a sequence of refactorings and hence lends itself naturally as a solution to the problem.

Planning: The task of coming up with a sequence of actions that will achieve a goal is called planning. Planning is represented by states, actions and goals. A planning problem consists of the following:

1. A description of the goal (in a formal language)

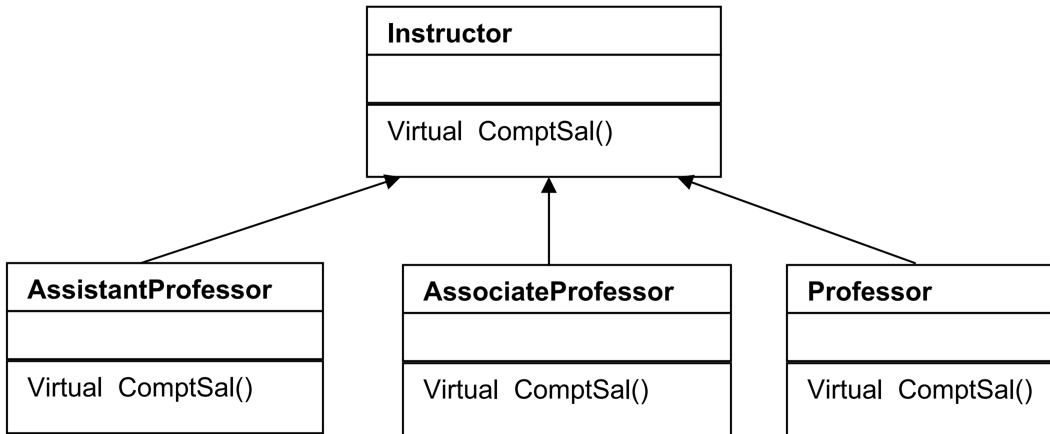

Figure 1: Class Hierarchy for the refactoring example.

```
virtual float Professor::ComptSal()
{
 // find the net pay by deducting the national tax, state tax
 // from the basic pay of $120,000. The national tax rate is 30.25%, the state
 // tax is 5.50%.
 float netPay = 120000- ((0.3025*120000) - (.0550*120000));
 return netPay;
}

virtual float AssociateProfessor::ComptSal()
{
 // find the net pay by deducting the national tax, state tax
 // from the basic pay of $100,000. The national tax rate is 30.25%, the state
 // tax is 5.50%.
 float netPay = 100000- ((0.3025*100000) - (.0550*100000));
 return netPay;
}

virtual float AssistantProfessor::ComptSal()
{
 // find the net pay by deducting the national tax, state tax
 // from the basic pay of $80,000. The national tax rate is 30.25%, the state
 // tax is 5.50%.
 float netPay = 80000- ((0.3025*80000) - (.0550*80000));
 return netPay;
}
```

Refactor – Rename to ComputeSalary

Refactor – Replace magic numbers

Refactor – Can be Pulled Up to base class, since it is the same in all the sibling derived classes

Refactor – Based on the comments, ExtractMethod to perform the TaxDeduction

Figure 2: Refactoring example - Before Refactoring

```
float Instructor::ComputeSalary()
{
 // the GetBasicSalary returns different values for different types of
 // instructors like assistant professor, professor. It is a protected virtual
 // method, thus it is an instance of the template design pattern.
 float basicSalary = GetBasicSalary();
 float taxDeduction = ComputeTaxDeduction(basicSalary);
 float netPay = basicSalary - taxDeduction;
 return netPay;
}

float Instructor::
ComputeTaxDeduction(float basicSalary)
{
 static const float NATIONAL_TAX_RATE = 0.3025;
 static const float STATE_TAX_RATE = 0.0550;
 float taxDeduction =
NATIONAL_TAX_RATE*basicSalary +
STATE_TAX_RATE*basicSalary;
 return taxDeduction;
}

virtual float Professor::GetBasicSalary()
{
 static const float
PROFESSOR_BASIC_SALARY = 120000;
 return PROFESSOR_BASIC_SALARY;
}

virtual float
AssociateProfessor::GetBasicSalary()
{
 static const float
ASSOC_PROFESSOR_BASIC_SALARY = 100000;
 return
ASSOC_PROFESSOR_BASIC_SALARY;
}

virtual float
AssistantProfessor::GetBasicSalary()
{
 static const float
ASSISTANT_PROFESSOR_BASIC_SALARY = 80000;
 return
ASSISTANT_PROFESSOR_BASIC_SALARY;
}
```

Figure 3: Refactoring example - After Refactoring.

2. A description of the set of actions (operators) available to achieve the goal, in a formal language;
3. A description of the starting state (also described in a formal language)

An operator is described not only by its action, but also the pre-conditions required by it and its effects i.e. the precondition states what must hold in a state before the action can be applied and the effect states the changes to the state as a result of applying the action. When given the set of initial state, actions and goal, the planning algorithm will come up with a sequence of actions such that beginning with the initial state, the goal can be achieved using only the set of available operators. The planning algorithm outputs a plan i.e. a sequence of actions.

A partial order planner exploits any decomposition available in the problem and attempts to find a solution independently to the sub problems. A combination of the independent sub goal plans to achieve the overall goal is called a linearization of the partial order plan. Partial order plans have the following features:

1. A set of ordering constraints like $A < B$, which means operator A must be executed before B (but not necessarily immediately before).
2. A set of causal links of the form $< A\ c\ B >$, where operator A achieves a pre-condition c for operator B to b executed. The causal link also asserts that no operator should execute in between that negates the effect that operator A achieves for operator B.

Algorithms like GRAPHPLAN [BF95] and IPP [KNHD97] exist which implement the partial order planning. We intend to use them in our system.

4.2 Proposed Novel Approach

Having described the basics of planning, we now describe our solution to the problem of finding an ordering among a set of refactorings. If we treat the different refactorings like Rename Method as operators, then we see that we can formulate the entire thing as a planning problem. Informally, the different parts of the problem are as follows:

- **Initial State:** The unrefactored program.
- **Operators:** The applicable refactorings like Rename Method, Add Class
- **Goal State:** The refactored program.

To describe the operators we will use the notation used by Roberts [Rob99] and specify the refactoring operators with pre-conditions and effects. We will specify the pre-conditions and effects for only a small catalog of refactorings to illustrate the general idea. Further, strictly for use with a real world planning algorithm, we will have to specify these using a formal language like First Order Logic, but

for ease of understanding, (and writing this paper) we will denote it in a natural language, English.

- **Rename Method (oldName, newName, class)** : Renames the method in *class* from *oldName* to *newName*.
  - **Pre-condition**: There should not be a method *newName* in the *class* already. Method *oldName* should belong to the *class*.[1]
  - **Effect**: Method *oldName* no longer exists in *class*. Method *newName* exists in *class*. All calls to *oldName* are changed to *newName*.
- **Add Class (newClass)** : Adds a new class *newClass*
  - **Pre-condition**: There should not be a class *newClass* in the program already.[2]
  - **Effect**: Class *newClass* added to the program.
- **PullUp Method (methodName, derivedClass, parentClass)** : Pulls up the method *methodName* in *derivedClass* to a parent class*parentClass*.
  - **Pre-condition**: There should not be a method *methodName* in the *parentClass* already. Method *methodName* should belong to the *derivedClass*. *All* classes that are siblings of *derivedClass* i.e derived classes at the same level, *must* have the method *methodName*
  - **Effect**: Method *methodName* no longer exists in *derivedClass*. It also does not exist in all classes that are siblings of *derivedClass*. Method *methodName* exists in *parentClass*.
- **Move Method (methodName, sourceClass, destinationClass)** : Moves the method *methodName* in *sourceClass* to *destinationClass*
  - **Pre-condition**: There should not be a method *methodName* in the *destinationClass* already. Method *methodName* should belong to the *sourceClass*.
  - **Effect**: Method *methodName* no longer exists in *sourceClass*. Method *methodName* exists in *destinationClass*.

How do planning algorithms help? Here we describe with our *instructor* example, how a planning algorithm can detect a conflict and arrive at a suitable plan. Having formulated the above set of refactoring operators; we have to instantiate it for specific cases. In our example, we can instantiate the Rename Method refactoring as RenameMethod (ComptSal, ComputeSalary, Professor) and PullUpMethod as PullUpMethod as PullUpMethod (ComptSal, AssociateProfessor, Instructor). Suppose the RenameMethod is done before the PullUpMethod, then, its effect negates one of the pre-conditions for PullUpMethod namely, all the sibling derived classes should have the same name is violated, since the class Professor, would not have a method named ComptSal at all. Thus in this instantiation, RenameMethod cannot be done before PullUpMethod. A planning algorithm will be able to detect such conflicts. It will also realize that in this instantiation, Extract Method and Replace Magic Number do not have any dependencies and hence can be ordered in any way. Thus the planner will be able to produce a suitable plan.

Tools such as Eclipse which allow refactoring, already have an underlying representation for the program and use this in the support that they provide for refactorings. We would also use this programatically to instantiate a refactoring operator with the proper pre-conditions and effects. Once this is done, this can be fed to a planning algorithm, which would yield the plan. Please note that the developer is only required to choose the different set of refactorings that he wants to apply, the code over which this should be applied and the input parameters to the refactoring operator like the new method name for RenameMethod.

4.3 Formulation of the planning problem for refactoring sequences

Here we formulate the planning problem for the *instructor* example. Note that we present a simplified version of the operators and predicates.

- Initial State: NOT RenamedMethod(ComptSal) *AND* NOT Pulled Up Method (ComptSal) *AND* NOT Replaced Magic Number *AND* NOT Extracted Method *AND* NOT Introduced Template Method
- Goal State: The same as above, with all the NOTs removed.
- Operators: RenameMethod, PullUpMethod, ExtractMethod, ReplaceMagicNumber, Introduce Template Method

With these as the input, the partial order planner will come up with a partial order like the one shown below:

1. PullUpMethod < FormTemplateMethod;
2. PullUpMethod < RenameMethod

One possible linearization of the plan is shown below:

1. PullUpMethod
2. ExtractMethod
3. ReplaceMagicNumber
4. FormTemplateMethod
5. RenameMethod

4.4 Evaluation Plans

We intend to do the following evaluation to test the efficacy of our solution. Since planning has been traditionally expensive in terms of memory and time requirements (P-SPACE

[1] We will ignore method overloading for the moment.

[2] With C++ namespaces, the same class name can exist in different namespaces, but we will ignore this for simplicity.

complete [RN95]), we will pay special attention to this. We intend to perform both user studies and automated runs to test our solution.

In the user study, we would assign developers (say 10 in number) having varying degree of programming experience, say 1-5 years, a reasonably sized programming project (LOC $>$ 5K) and the task of refactoring some portions of it (similar to our *instructor* example). We will compare the effort (time) it takes for the human subjects to come up with an order among the sequence of refactorings and the time taken by our system, using the planner. We will also compare the actual order arrived at by the developers and our system, to check for inconsistencies. This should give us a basic feel of the utility of our system.

In the automated runs to test our system, we will write scripts to automatically perform a set of refactorings (in a random order) on a program. We will then feed the same set of refactorings to our planning system and come up with the plan. The experiment will continue, with an automated application of the refactorings to the program, in the same order as suggested by the planner. We will then apply the object oriented metrics like Chidamber and Kemerer [CK94], Lorenz and Kidd [LK94] on the random order refactored program and the program refactored using the order specified by the planner. A comparison of the metrics in the two cases, should again give an insight into the utility of the system. Note that some subjectivity is involved in this, since metrics is on the whole, a slightly controversial area. For example, we cannot say that a program with 10 classes and 50 methods is necessarily better than a program with 5 classes and 30 methods or vice versa. The metrics that we use will be the ones suggested by NASA [NAS98] and will include the following:

- Cyclomatic complexity
- Lines of Code
- Comment percentage
- Weighted methods per class
- Response for a class
- Lack of cohesion of methods
- Coupling between objects
- Depth of inheritance tree
- Number of children

Further, we will use refactoring sequences of different lengths like a set of 5, 10, 15 etc to test the usefulness of the planning approach. In all the test cases, time and memory usage will be monitored.

We plan to implement our idea as an Eclipse Plug-in, PARSE (Planning Assistant for Refactoring Sequences). We will also compare our tool with the AGG tool built by Mens et al [MTR05, MTR06], since it too attempts to find conflicts among different refactorings using another approach (critical pair analysis).

5 Summary of foreseen contributions

As described earlier, software maintenance is the most expensive phase of the software life cycle. Refactoring helps a lot in alleviating the problem associated with maintenance. For this, refactoring itself should be easy to apply, error free and not very time consuming, else the developer would shy away from doing the refactorings as he normally is under a great time pressure. Discovering the ordering among a set of applicable refactorings can be tedious, error prone and time intensive. This is where our tool steps in and helps by automatically finding a suitable plan of action. The developer now has to merely follow the suggested plan and apply the refactorings.

References

[BF95] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[DDN00] Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–177, New York, NY, USA, 2000. ACM Press.

[EV05] Ran Ettinger and Mathieu Verbaere. Refactoring bugs in eclipse, intellij idea and visual studio., 2005.

[Fow99] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[HD05] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, 2005.

[HKKI04] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Refac-

toring support based on code clone analysis. In *PROFES*, pages 220–233, 2004.

[IBM05] IBM. Eclipse ide, 2005. http://eclipse.org.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743, 2001.

[KNHD97] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. Technical Report report00088, 1, 1997.

[LK94] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[MEDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 2005.

[Mic05] Microsoft. Microsoft visual studio ide, 2005. http://www.microsoft.com.

[MTR05] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, April 2005.

[MTR06] Tom Mens, Gabi Taentzer, and Olga Runge. Analysis refactoring dependencies using graph transformation. *Software Systems Modeling (SoSyM)*, 2006.

[NAS98] NASA. Applying and interpreting object oriented metrics, 1998.

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA, 1992.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[Rob99] Donald Bradley Roberts. *Practical analysis for refactoring*. PhD thesis, 1999. Adviser-Ralph Johnson.

[SSL01] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics based refactoring. In *CSMR*, pages 30–38, 2001.

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. In Dieter Rombach and Mary Lou Soffa, editors, *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.

[WD06] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.