Interactive and Guided Architectural Refactoring with Search-Based Recommendation

Yun Lin\textsuperscript{1,2}, Xin Peng\textsuperscript{1,2}, Yuanfang Cai\textsuperscript{3}, Danny Dig\textsuperscript{4}, Diwen Zheng\textsuperscript{1,2}, Wenyun Zhao\textsuperscript{1,2}

\textsuperscript{1}School of Computer Science, Fudan University, China
\textsuperscript{2}Shanghai Key Laboratory of Data Science, Fudan University, China
\textsuperscript{3}Department of Computer Science, Drexel University, USA
\textsuperscript{4}School of EECS, Oregon State University, USA

ABSTRACT

Architectural refactorings can contain hundreds of steps and experienced developers could carry them out over several weeks. Moreover, developers need to explore a correct sequence of refactorings steps among many more incorrect alternatives. Thus, carrying out architectural refactorings is costly, risky, and challenging. In this paper, we present Refactoring Navigator: a tool-supported and interactive recommendation approach for aiding architectural refactoring. Our approach takes a given implementation as the starting point, a desired high-level design as the target, and iteratively recommends a series of refactoring steps. Moreover, our approach allows the user to accept, reject, or ignore a recommended refactoring step, and uses the user’s feedback in further refactoring recommendations. We evaluated the effectiveness of our approach and tool using a controlled experiment and an industrial case study. The controlled experiment shows that the participants who used Refactoring Navigator accomplished their tasks in 77.4\% less time and manually edited 98.3\% fewer lines than the control group. The industrial case study suggests that Refactoring Navigator has the potential to help with architectural refactorings in practice.

CCS Concepts

•Software and its engineering → Maintaining software; Search-based software engineering;

Keywords

automatic refactoring, reflexion model, high-level design, interactive, user feedback

1. INTRODUCTION

The maintenance of industrial projects is often hindered by technical debt, which is a metaphor of sacrificing long-term quality for short-term value. A recent empirical study \cite{11} on industrial projects shows that architectural flaws are the most important source of technical debt. Although architectural flaws could be addressed by refactoring, developers often perceive architectural refactoring as substantially costly and risky \cite{12, 23}.

Examples of architectural refactorings include decoupling a large code base into smaller modules, retrofitting a design pattern, etc. In these cases, the developer has a desired high-level design in mind as the target of refactoring. However, the developer needs to conduct a series of low-level refactorings to achieve the desired design. Without explicit guidance about which path and/or which refactorings to take, such refactoring tasks can be demanding. For example, Tokuda and Batory \cite{44} presented two case studies where architectural refactoring involved more than 800 refactoring steps, estimated to take more than 2 weeks. From these examples, we see a need for automation of the process. Several researchers proposed to automate various stages of the process for applying small-scale refactorings, such as identifying refactoring opportunities \cite{7–10, 41, 45, 46}, scheduling code smell resolution sequences \cite{27}, automatically completing refactoring operations \cite{14, 16}, and searching for optimal refactoring solutions \cite{19, 36, 37, 40}.

Our approach is guided by several observations. Fully automatic refactoring usually does not lead to the desired architecture. Research and experience \cite{35} show that even semi-automated tools for lower-level refactorings have been underutilized. Moreover, human designers understand the problem domain better and their feedback should be included in tool-supported refactoring.

In this paper, we present a tool-supported, interactive, and guided recommendation approach for architectural refactoring. We implemented our approach as a tool, Refactoring Navigator (RN for short)\textsuperscript{1}, based on the metaphor of the GPS route navigation. We consider the current implementation of a software system as the starting point, the desired high-level structure as the target, and calculates a reflexion model \cite{32} to reveal the discrepancies between the source and desired design. The bigger the discrepancies are, the larger the distance between the starting point and the target is.

RN calculates and recommends refactoring “paths” from the starting point to the target, each path being a sequence of stepwise atomic refactorings, such as moving a method from one class to another, decomposing a class, or pulling up a field/method. The user can examine the recommended steps, and accept, reject, or ignore them interactively. RN records these decisions as user feedback, and considers them when calculating the next recommendation.

RN applies the accepted recommendations to the source code automatically and updates the reflexion model accordingly. Based on the updated reflexion model and user’s feedback, RN launches a new iteration with updated refactoring recommendations. This

\textsuperscript{1}RN demo: https://www.youtube.com/watch?v=YVT5UU7xIgCQ
way, the user can complete the refactoring process in an iterative and interactive manner. In each iteration, RN uses a hill-climbing algorithm to recommend refactorings that increase the consistency between the source and the target, optimize modularity metrics, and incorporate user feedback on preferred refactorings.

We used two complementary methods to evaluate RN: a controlled experiment, and a case study on an industrial project. In the controlled experiment, two groups of students performed the same architectural refactoring task. Both groups used RN to generate and examine reflexion models, while only the experimental group used the recommendation function of RN. The control group was additionally equipped with code smell detection tools to find and automate refactoring opportunities. We compared the refactoring results and analyzed the behaviors of the two groups. The results indicate that RN significantly improves the efficiency of refactoring: on average, the experimental group participants spent only 13.3 minutes to finish refactoring, and manually edited 6 lines of code. By contrast, the control group participants spent 59 minutes and had to manually edit 354 LOC. The savings on both time and effort were significant.

To further validate our approach, we conducted an industrial case study by applying RN to refactor the original source code from an industrial partner. This real-world industrial project has over 40K LOC, and our industrial partner had previously encountered significant challenges in architectural refactoring, spending several weeks on the task. Using RN, we accomplished the refactoring in less than a day, showing the feasibility of applying RN in real projects.

This paper makes the following contributions:

**Approach:** Our approach uses a hill-climbing algorithm to search and recommend refactoring “paths” that revises source code step by step toward a target design, and completes atomic refactoring steps automatically. Instead of a push-button technique that may produce results that users do not want, our approach uses feedback interactively to recommend refactoring steps.

**Implementation:** We implemented our approach in the RN tool\(^2\) developed as an Eclipse plugin so that the user can benefit from the interactive features of an IDE, as shown by the results.

Evaluation: We conducted a two-pronged empirical evaluation. A controlled experiment shows that the RN users can accomplish their refactoring task in 77.4% less time and edited 98.3% fewer lines of code. A case study on an industrial project suggests that RN has the potential to help with architectural refactoring in practice.

2. AN ILLUSTRATIVE EXAMPLE

To illustrate our approach, we use a bus monitoring software system that has evolved over time. As shown in Figure 1a, the latest implementation contains: a BusMonitor class containing data and associated methods reflecting the dynamic information of buses such as schedules and location, a VehicleUI class that accepts user queries about bus information, and three views—MapView, TableView and CurveView classes—to display dynamic operational information of buses in different formats.

\(^2\)RN at GitHub: https://github.com/llmhyy/Refactoring-Navigator

At the early stages, only MapView was required, and the BusMonitor class created and maintained an instance of MapView. Later on, when the project required a table view and curve view, the developer extended BusMonitor to maintain two more instances, and its `updateViews()` method enumerated and updated all three views.

As the project required more views, it is obvious that BusMonitor quickly grew into a god class, and the proper refactoring strategy is to apply the Observer pattern, as shown in Figure 1b. In this target design, the bus information data should be isolated into a BusData module that takes the role of Subject. The Concrete View module should contain classes taking the role of Concrete Observers, such as the MapView class. An Abstract View observer interface is needed to decouple the Subject from Concrete Observers.

Existing work on reflexion model [32] can be used to show the discrepancies between the implementation and the target design, but it leaves the refactoring task to the developer. Although this small example appears to be straightforward, in the real project where we extracted this example, the BusMonitor was already a God class that contained many functions related to bus data manipulation and was tangled with many other classes. In order to extract Bus Data module and decouple BusMonitor from all the views, the developer must read through thousands lines of code, deal with compilation errors, and preserve the refactored program behaviors. In large systems, such refactoring can be very time consuming.

Thus we ask the question, can state-of-the-art automatic refactoring tools help alleviate the maintenance costs? We applied two representative refactoring tools, JDeodorant [13, 46] (a code smell detector) and CodeImp [37] (a search-based refactoring tool) to the bus monitoring system for exploration.

Given the source code, JDeodorant reports 16 feature-envy and 9 God-class smells. However, after removing these code smells, the refactored implementation and the target design still differ significantly. As an example, JDeodorant suggests to move the `init()` method from the BusMonitor class to the VehicleUI class to reduce coupling. Yet this move does not help eliminate the inconsistencies between the implementation and the target design. Because JDeodorant provides an unordered list of basic refactoring suggestions for various code smells, and provides multiple ways to fix a smell, it presents no clear steps or order leading to the target design.

CodeImp provides consecutive steps guiding developers to perform stepwise refactorings in order to improve design metrics, such as LCOM5, RFC, CBO [20]. The most recommended steps suggest redistributing the fields and methods between BusMonitor and its super class Monitor. For example, the first three steps aim to pull up the `updateMap()`, `updateTable()`, and `updateCurve()` methods from BusMonitor to Monitor to improve the LCOM5 metrics. The fourth step aims to pull down the `activatedViewNum` field from Monitor to BusMonitor to improve the CBO metrics. Obviously, these refactorings do not lead the implementation to the target design.

Clearly, existing state-of-the-art tools were not designed for architectural level refactoring. In this paper, we present RN, a novel, interactive, tool-supported technique to address these problems. For example, given a source model depicted in Figure 1a and a target design in Figure 1b, RN will first recommend a “path” with 4
atomic refactoring steps:

1. Step Method: move `updateMapView()` method from `BusMonitor` class to `MapView` class;
2. Step Method: move `updateTableView()` method from `BusMonitor` class to `TableView` class;
3. Pull Up Method: pull up `updateMapView()` method in `MapView` class and `updateTableView()` method in `TableView` class to a newly created abstract class in `AbstractView` module.
4. Extract Class: split the large `BusMonitor` class by extracting out a `refresh()` method that invokes view-updating methods and several bus data relevant fields.

For each recommended step, the user can accept, ignore, or reject it. If the user accepts a recommended step, `RN` will automatically execute it and refactor the source code. If the user rejects a recommended step, `RN` will recalculate a new path, taking user feedbacks (i.e., the previous decisions) into consideration. For example, if the user rejects Step 3 after the first two steps are accepted and executed, `RN` will recalculate and recommend the following path:

1. Step Method: move `updateCurveView()` method from `BusMonitor` class to `CurveView` class;
2. Pull Up Method: pull up the three view-updating methods to a newly created abstract class in `AbstractView` module.
3. Extract Class: split the large `BusMonitor` by extracting `refresh()` and data fields.

If the user accepts all these steps, `RN` will complete these atomic refactorings automatically, and the Bus Monitor source code will be refactored into an Observer pattern, without the user writing a single line of code.

3. APPROACH

Figure 2 presents an overview of `RN` that consists of six steps. The grey rectangles represent steps requiring human intervention. We briefly explain and then elaborate each step.

![Figure 2: Approach Overview](image)

1. Target Design Construction: The user starts `RN` by first constructing a Target Model that reflects the desired structure. Similar with other work on reflexion models [32] [38], the Target Model is simply a graph with nodes (modules) and edges (relations among modules). The user can also select the part of the source code s/he thinks should conform to the target model.

2. Source Model Extraction: From the selected source code, `RN` automatically generates a Source Model, which is another graph reflecting the relation among program elements.

3. Reflexion Model Generation: Given the Target Model and Source Model as input, `RN` calculates a Reflexion Model to reveal discrepancies between the two graphs. These recommendations form a path (i.e., refactoring steps) that will refactor the source code towards the target design.

4. Refactoring Path Calculation: Using the Reflexion Model as input, `RN` computes an ordered list of refactoring Recommendations. These recommendations form a path (i.e., refactoring steps) that will refactor the source code towards the target design.

5. User Examination: Given each recommended step, the user can decide whether to accept, reject, or ignore it. `RN` will take these User Feedbacks into consideration when calculating the next steps to recommend.

6. Recommended Step Execution: Each accepted step will be executed, updating both source code and Source Model automatically. After that, `RN` enters step 3 to generate an updated Reflexion Model and starts a new iteration consisting of steps 3, 4, 5, and 6.

The refactoring process ends when no discrepancies exist in the reflexion model, or the tool cannot produce further refactoring recommendations to eliminate the discrepancies. The user can manually change the source code or adjust the target model at any time to restart the iteration. Next we elaborate on these 6 steps.

3.1 Target Model Construction

Similar to traditional reflexion models, `RN` allows the user to draw a simple box-and-line graph modeling the desired design. Each box has a name, representing a module, i.e., a collection of program elements. The user can enter a set of descriptive keywords that are used to find matching source code elements.

Different from existing work on reflexion model, `RN` distinguishes three types of dependencies between modules:

- **Inheritance**: module `M1` inherits `M2` if some class in `M1` inherits from, or implement, classes or interfaces in `M2`;
- **Creation**: module `M1` and module `M2` have creation relation if some class in `M1` creates instances of classes in `M2`;
- **Other**: classes in module `M1` and module `M2` have dependency relations other than inheritance and instantiation.

Two modules can have multiple relations. We distinguish these three types based on the rationale that abstraction through interface implementation is a common refactoring strategy, while an entry class, e.g. a UI class, usually has to create a lot of classes, but does not substantially depend on them.

Figure 1b depicts a target model, the application of Observer pattern to the bus monitoring system. The Abstract View should contain the Observer interface definition. The Concrete View module should contain classes taking the role of Observers. The Bus Data model should contain classes playing the role of Subjects. The `Monitor GUI` model contains entry classes whose only task is to instantiate Observer classes. The descriptions within each box are keywords `RN` uses to match source code elements. For example, classes with words “table”, “map”, “curve”, “view”, are likely to be observers.

3.2 Source Model Extraction

Given a designated piece of source code, `RN` extracts a source model representing program elements and their dependencies. A program element can be a class, interface, method, or field. Dependencies include inheritance, implementation, method call, field access, type reference, and instance creation. In a source model, we similarly categorize dependencies into Inheritance, Creation, and Others as defined in target model. Figure 1a depicts the source model extracted from the code of the bus monitoring system.

3.3 Reflexion Model Generation

Given the target and source models, `RN` generates a reflexion model by mapping source model elements to target model modules. A reflexion model reveals the difference between the source and the target by revealing conformance and discrepancies: 1) Convergence Dependency: a dependency exists in both the target and the source; 2) Absence Dependency: a dependency exists in the target but missing in the source; and 3) Divergence Dependency: a dependency exists in the source but not expected in the target.

Figure 1c shows a reflexion model generated from the target and source shown in Figure 1b and Figure 1a. In this case, `RN` maps `MapView`, `TableView`, and `CurveView` classes into the `ConcretView` module in the target, maps the other classes into the `MonitorGUI` module. This mapping results in four absence dependencies to `Bus Data`
and Abstract View modules since none of the source elements are mapped to them. After applying Observer Pattern, the Monitor GUI module should only be responsible for creating objects, but does not have any other dependencies to the views. In the current source code, however, the Bus Monitor depends on all three views, which explains the divergence dependency from Monitor GUI module to Concrete View module. Next we elaborate on the mapping algorithm.

The Mapping Algorithm.

The objective of the algorithm is to find the best one-to-many mappings between each module in the target, and program elements in the source, using two criteria: 1) lexical similarity: the topic/concern description of the module and program elements should match most, and 2) design conformance: the consistency between the target and the source should be maximized.

Similar to existing work (e.g., [28]), we measure lexical similarity based on the descriptions of modules provided by the user and the code text of program elements. We transform each model description or program element into a document by tokenization, filtering out stop words and word stemming, and encode it in a Term-Frequency/Inverse-Document-Frequency (TF/IDF) vector [5]. For a module $m$ with an $n$-dimensional TF/IDF vector $V_m$ and a class/interface $c$ with an $n$-dimensional TF/IDF vector $V_c$, we compute their lexical similarity as follows:

$$Sim_{lex}(m, c) = \frac{\sum_{i=1}^{n} V_m[i] \times V_c[i]}{\sqrt{\sum_{i=1}^{n} V_m[i]^2} \times \sqrt{\sum_{i=1}^{n} V_c[i]^2}}$$

(1)

To calculate design conformance, we quantify the two types of inconsistencies (i.e., absence and divergence) for each dependency type (Inheritance, Creation, or Other) as follows. Given a dependency type $d_{type}$, let the set of absence dependencies of $d_{type}$ be $D_{abs}(d_{type})$, the set of convergence dependencies of $d_{type}$ as $D_{div}(d_{type})$, the number of modules in target model as $N_m$, and calculate the design conformance of type $d_{type}$ as follows:

$$Sim_{str}(d_{type}) = 1 - \frac{|D_{abs}(d_{type})| + |D_{div}(d_{type})|}{N_m \times (N_m - 1)}$$

(2)

Generally, Equation 2 indicates that (1) the more absence (or divergence) dependencies of type $d_{type}$ are derived from the mapping, the less design conformance value is, and (2) the conformance value is normalized between 0 and 1.

Given these two measures, we use a genetic algorithm (GA) [17] to find an optimal mapping solution to generate reflexion model. It first generates a set of $individuals$ as the initial population, then evolves the population by creating new generations of mapping solution through an iterative process, until the algorithm reaches a predefined number of generations $N_{pop}$. An individual represents a set of mappings, consisting of a sequence of $genes$, each indicating which module a program element is mapped to. For example, an individual [1, 2, 1, 3] represents four classes/interfaces (an element index in the array) mapped to modules 1, 2, 1, and 3 (an element value in the array) respectively.

The algorithm randomly generates $N_{pop}$ individuals as the initial population (where $N_{pop}$ is a predefined even number indicating the size of the population). We represent each individual’s fitness value as the weighted sum of lexical similarity and design conformance:

$$F_{map} = w_1 \times \frac{\sum_{c \in CS} Sim_{lex}(Map(c), c)}{|CS|} + w_2 \times \frac{\sum_{t \in DT} Sim_{str}(t)}{|DT|}$$

(3)

In the above equation, $CS$ and $DT$ are the set of classes/interfaces in the source and the set of dependency types in the target respectively; $Map(c)$ is the target module that $c$ maps to, and $w_1$ and $w_2$ are two predefined weights satisfying $w_1 + w_2 = 1$.

We randomly pair all individuals in $N_{pop}$ into $N/2$ pairs. For two paired individuals $parent1$ and $parent2$, our approach generates two offsprings, $offspring1$ and $offspring2$, by exchanging their genes. For each gene, we pass the value from $parent1$ or $parent2$ to $offspring2/offspring1$ with a given probability $P_{crossover}$. We then mutate the genes of the offsprings. For each gene, our approach changes the value to indicate mapping to another randomly selected target module with a given probability $P_{mutation}$.

After the algorithm reproduces a new generation with $2N_{pop}$ individuals, we sort them by fitness in descending order and select the first set of $N_{pop}$ individuals as the candidate generation. To ensure diversity, the approach does not allow the same individual to appear multiple times in one generation. Accordingly, it will scan the first set of $N_{pop}$ individuals to identify those duplicated ones and replace them with ones in the second set of $N_{pop}$ individuals.

3.4 Refactoring Path Calculation

Similar to a GPS navigator that calculates routes given a start location and a destination, RN calculates refactoring “paths” given a source model and a target model. Each path consists of an ordered list of atomic refactoring steps. Following existing work on automatic refactoring [13, 21, 46], we support the following types of atomic refactoring: 1) Move Method—moving a method from one class to another; 2) Extract Class—decomposing a large class into smaller classes; and 3) Pull Up Field/Method—extracting and moving similar fields or methods from multiple classes to an existing or a newly created abstract class or interface. Different from previous work, our objective is to identify refactoring opportunities that lead to the target design. To ensure that each refactoring preserves the program behavior, our approach examines a set of preconditions for each atomic refactoring, following and extending on existing work [13,21,46].

We use a search-based algorithm to find an optimal refactoring path criteria: 1) minimizing the inconsistencies between the target and source; 2) improving OO design quality, such as coupling and cohesion; and 3) maximizing the lexical similarity between the target and source. We now introduce how we quantify these criteria as fitness value and how our search algorithm works.

Fitness Value of Reflexion Model.

Inconsistency. We measure this primary factor by the number of remaining inconsistencies after applying a series of refactorings:

$$N_{str} = \sum_{t \in DT} |D_{abs}(t)| + \sum_{d \in D_{div}(t)} |SrcD(d)|$$

(4)

In Equation 4, the general inconsistency consists of absence inconsistency and divergence inconsistency of all types of dependency. We use $DT$ as the set of dependency types, and for each dependency type $t \in DT$, $D_{abs}(t)$ and $D_{div}(t)$ are its absence dependency set and divergence dependency set of the resulted reflexion model respectively. As for absence dependency set of type $t$, we simply use its size to quantify the absence inconsistency. However, as for divergence dependency set, we quantify the divergence inconsistency in a finer way. Given a divergence dependency $d$, we use $SrcD(d)$ to denote the set of program dependencies (e.g., method invocation, field access, etc.) contributing to $d$. For example, a divergence dependency between modules $M_1$ and $M_2$ could be contributed by 5 method invocations, thus, any refactoring supposed to reduce the number of method invocation between $M_1$ and $M_2$ should decrease the inconsistency of the reflexion model.

For the example showed in Figure 1c, there exist three types of dependency, i.e., Inheritance, Creation, and Other. For Inheritance, $t_i$, $|D_{abs}(t_i)| = 1$ and $|D_{div}(t_i)| = 0$; For Creation, $t_c$, $|D_{abs}(t_c)| = 1$ and $|D_{div}(t_c)| = 0$; For Other, $t_o$, $|D_{abs}(t_o)| = 2$
and \( |D_{d(k)}(t_0)| = 1 \). In addition, there are 6 method invocations contributing to the divergence dependency from Monitor GUI module to Concrete View module. As a result,
\[
N_{str} = 1 \ (Inheritance_{abs}) + 1 \ (Creation_{abs}) + 2 \ (Other_{abs}) + 1 \ (Other_{div}) \times 6 = 10
\]

**Quality.** We measure design quality by CBO (Coupling Between Objects, a value between 0 and 1, indicating the overall coupling in the source) and LCOM5 (Lack of Cohesion Of Methods, a value between 0 and 1 indicating the lack of cohesion of methods of a class/interface \( c \) within all the classes/interfaces, \( CS \)) [20].

\[
D_{Q_{avg}} = \frac{CBO + \frac{\sum c \in CS \ LCOM5(c)}{|CS|}}{2}
\]

(5)

**Lexical Similarity.** This measures the average lexical similarity:

\[
Sim_{lex_{avg}} = \frac{\sum_{c \in CS} Sim_{lex}(Map(c), c)}{|CS|}
\]

(6)

Given these measures, we define a fitness function for a refactoring path as follows:

\[
F_{ref} = -N_{str} - D_{Q_{avg}} - (1 - Sim_{lex_{avg}})
\]

(7)

The fitness value is less than 0. The higher the value, the better the solution. In Equation 7, \( N_{str} \in [0, \infty) \), \( D_{Q_{avg}} \in [0, 1) \), and \( Sim_{lex_{avg}} \in [0, 1] \). It indicates that the consistency between the target and source is dominating, and OO design metrics and lexical similarity are used to break ties.

**Refactoring Path Generation Algorithm.**

We use a revised multiple ascent hill-climbing (HCM) algorithm to search for an optimal refactoring path (see Algorithm 1). This algorithm has three features: it iteratively attempts to reach highest fitness value, adapts hill-climbing algorithm to avoid being trapped in a local optima, and takes user feedback into consideration to adjust fitness value. The algorithm takes as input a reflexion model \( model \), recorded \( fb \), and a given descent number \( descNum \), and returns a refactoring path (i.e., a series of refactoring steps).

```
Input : reflexion model model, feedbacks fb, descent number descNum
Output: bestSolution
1 bestFit = evalFitness(model);
2 bestSolution = [];
3 solution = [];
4 for i = 0; i < descNum; i++ do
5    localBestFit = evalFitness(model);
6    while true do
7        ref = findBestRef(model, refCands, fb);
8        if refCands == [] then
9            return bestSolution;
10       end
11        ref = findBestRef(model, refCands, fb);
12        model = model.apply(ref);
13        solution.add(ref);
14        fit = evalFitness(model);
15        if fit > localBestFit then
16            localBestFit = fit;
17            if localBestFit > bestFit then
18                bestFit = localBestFit;
19                bestSolution = solution;
20            end
21        end
22    end
23 end
24 return bestSolution;
```

**Algorithm 1:** Search-Based Refactoring Recommendation

The algorithm first computes the fitness value (see Equation 7) of the given reflexion model, and initializes the value of \( bestFit \) (Line 1), along with two empty lists, \( bestSolution \) and \( solution \) (Line 2-3). It then repeats a procedure of searching a local optimum for \( descNum \) times as follows. It first initializes the value of local \( bestFit \) (Line 5), then finds a local optimum in an iterative process (Line 6-24).

In each iteration, the algorithm identifies a set of possible refactoring opportunities \( refCands \) (Line 7). If \( refCands \) is not empty, the algorithm selects a best candidate refactoring by combining the fitness function with user feedback (Line 11, see Equation 8). The selected refactoring \( ref \) is then applied to the current model (Line 12) and added to the current solution (Line 13). If the fitness value of the updated model is greater than local \( bestFit \), the algorithm updates the local optimum (Line 16). Otherwise, a local optimum has been reached. If the fitness value is greater than \( bestFit \), the best solution is updated to the current local optimum. Once a local optimum is reached, the algorithm escapes the local optimum to explore more optimal solution (Line 22).

The \( findBestRef \) function considers both the fitness function and user feedback by Equation 8 to select a best refactoring from \( refCands \). A candidate refactoring similar to an accepted refactoring (similarity is calculated by processing the difference between refactoring types and elements involved) is more likely to be accepted/rejected again. For a candidate refactoring similar to \( m \) approved and \( n \) rejected refactorings, we calculate its value as:

\[
Eva_{ref} = F_{ref} \times (1 - \alpha)^m \times (1 + \beta)^n
\]

(8)

where \( F_{ref} \) is the fitness value of the reflexion model after applying the candidate refactoring, and \( \alpha \) and \( \beta \) are predefined coefficients between 0 and 1.

**Example.** For the Bus Monitor example, \( RN \) first detected four refactoring opportunities from the source code: Extracting a class from BusMonitor class, or Moving one of the three update*View() method from BusMonitor to *View (for Map, Table, or Curve) class. Our algorithm chose one of the Move Method refactoring as the first recommended step for the following reason:

Each \( update*View() \) in BusMonitor class invokes two methods in *View class and it is invoked only once in BusMonitor class. Thus, the Move Method refactoring will introduce one invocation while removing two from *BusMonitor class to *View class. Hence, it will reduce the source contribution to the divergent dependency from Monitor GUI module to Concrete View module from 2 to 1. By contrast, despite that Extract Class refactoring can remove an absence dependency from Monitor GUI module to Bus Data module, it will introduce one divergence dependency (with two source contributions) from Bus Data module to Concrete View module, increasing the overall discrepancies.

In the second step, \( RN \) recommends another move refactoring for the same reason. When calculating the third step, \( RN \) found a new Pull Up Method refactoring opportunity, i.e., pulling up update*MapView() method in MapView class and updateTableView() method in TableView class to a newly created abstract class in AbstractView module. This step generates higher fitness value because the Move Method refactoring can decrease the inconsistency only by 1, while the Extract Class refactoring can decrease the inconsistency by 4 (2 for reduced absence dependency from Monitor GUI module to Bus Data module and Abstract View module; while 2 for reduced source contribution for divergence dependency from Monitor GUI module to Concrete View module). Therefore, it is recommended as the third step. After the fourth step (i.e., the Extract Class refactoring) is chosen, \( RN \) cannot reduce more discrepancies, and the path with 4 steps (listed in Section 2) is completed.
3.5 User Examination

Accepting user feedback is a key feature of RN. The user can examine every step in a recommended path, and decide to accept, ignore, or reject any steps. Consider the 4-step path calculated above. Even though RN recommended Pull Up Method refactoring as the third step, it is more intuitive to move all three update*View() methods before the Pull Up Method refactoring. In this case, the user may accept the first two steps, but reject the third one. The two accepted steps will be executed and RN will start another round of recommendation starting from step 3, referencing to user’s accepted and rejected refactorings by Algorithm 1 and Equation 8. In the Bus Monitor example, given the two acceptances and one rejection from the user, RN will reevaluate the path as follows. Assuming $\alpha$ and $\beta$ are both 0.5, and RN needs to reevaluate Pull Up Method versus Move Method. For the former, since a similar move was rejected previously, its marginal increase of inconsistency would be $\Delta N_{str} = -4 \times (1 - 0.5) \times (1 + 0.5) = -2$. For the latter, since it is similar to two accepted refactorings, then $\Delta N_{str} = -1 \times (1 - 0.5) \times (1 + 0.5)^2 = -2.25$. Therefore, the Move Method was recommended.

3.6 Recommended Step Execution

When the user accepts a recommended step, RN will update the source code and source model automatically, and generate a new reflexion model. Note that the user can accept some, but not all recommended steps, and ask RN to execute refactoring. In this case, the tool will recheck the precondition of an accepted step before applying it. If the precondition is not satisfied, the tool asks the user to regenerate the reflexion model.

4. EVALUATION

To evaluate the effectiveness of RN, we answer the following research questions:

RQ1 (Productivity): Does RN help developers perform architectural refactoring faster and more correctly?

RQ2 (Search-Contribution): What is the contribution of RN’s search-based refactoring path recommendation and auto-refactoring completion?

RQ3 (Feedback-Contribution): What is the contribution of user feedback loop?

RQ4 (Applicability): Is our approach applicable to real-world industrial projects?

To answer these questions, we use two complementary approaches: a controlled experiment and an industrial case study. We used the controlled experiment to quantitatively compare the effort needed to refactor towards a target design, and to analyze the respective contributions of refactoring path recommendation and user feedback. To determine applicability, we used the real industrial project to which we applied RN to reenact the architectural refactoring that our industrial partner previously performed manually.

The controlled experiment is the only way to quantitatively and precisely measure human effort and productivity, whereas the real industrial project evaluates whether RN has the potential to be used in real-world projects. As part of our significant effort of evaluation, we hoped to find large-scale, architectural refactorings in open source projects. However, after months of combing through the revision history of a large number of open source projects used in the corpus of an award-winning paper [34], we realized that large-scale refactorings in open source projects involving dozens of files are extremely rare. The number of files touched by most recorded refactorings in [34] is smaller than those in the project we used in our controlled experiment. Moreover, the major advantage of RN is allowing the user to describe a target design, which appears to be impossible to obtain from open source projects unless we are the developers. Next we present our evaluation procedure and results.

4.1 Controlled Experiment

The purpose of our controlled experiment was to quantitatively measure the effectiveness of RN, which consisted of two phases. The first phase of our experiment was only designed to show that the automatically generated reflexion model is good enough for the user to start with. That is, the mapping of the source code does not severely deviate from the target modular structure. The second phase aimed to evaluate the main contributions of RN: 1) search-based refactoring path recommendation with automatic completion of refactoring steps, and 2) automatic navigation adjustment based on user feedback.

Subject System. We chose the subject system from a class project used in a software design course taught in both Fudan University and Drexel University. Unlike most student projects, this one has medium size: it is a questionnaire management system. Questionnaires come in two types: either a survey or a test with grades. The project supports two types of users: the designer who creates a questionnaire and the respondent who completes the questionnaire.

A designer can use the system to create, modify, save, load, export, display, and print different types of questions, including multiple choice questions, matching questions, ranking questions, etc. For a test, the designer can also assign points and provide correct answers. A respondent can use the system to complete a given questionnaire, which the system can automatically grade and/or tabulate. The system design should support easy extension of new types of questions, and different ways to create, save, print, and display the questionnaire. Students should carry out the design through the proper application of multiple design patterns, and must implement the system in 10 weeks using Java.

As the creator of the student assignment, the third author of the paper is well-aware of the best possible design for the system, and provided an authoritative modular design as the target. We selected one of the student submissions as the subject to be refactored. This submission contains 21 classes, 2 interfaces, 203 methods, 44 fields, and 2,866 LOC. The source code of student submission can be found at [3]. We chose this particular submission because it represents typical design flaws, such as the violation of single responsibility principle and high coupling.

In our refactoring experiment, we chose to refactor the code implementation of saving and loading of a questionnaire to and from an XML file. This code involved 15 classes, one interface, 121 methods, 23 fields, and 1,010 LOC. As the fundamental part of the system, the highly coupled source code in this submission made it difficult to apply proper design patterns manually or support the required extensions easily, such as adding new types of questions.

Figure 3 depicts the target design, which consists of five modules: IO, Abstract Question, Concrete Question, Abstract Answer, and Concrete Answer. The classes in the Concrete Question module should inherit the classes in the Abstract Question module; the classes in the Concrete Answer module should inherit from classes in the Abstract Answer module; the IO module can depend on the two abstract modules, but not on the modules with concrete classes, other than creating their instances.

From Figure 3, we see that the given implementation is inconsistent with the desired design, as the IO module depends on several methods of concrete question classes and concrete answer classes (see the two divergence dependencies in dashed arrows). In this case, the student author of the code was new to OO concepts: although he employed abstract classes (e.g., Question) and interfaces (e.g., Answer), he did not fully grasp how to leverage polymorphism.
to decouple dependencies among classes. Therefore, the inheritance hierarchy in his code violated the Liskov Substitution Principle [25]. More specifically, the IO class manipulated the internal data structure of concrete question and answer classes, and took all the responsibility of creating, modifying, loading, and saving these classes, which created numerous divergence dependencies from the IO module to Concrete Question and Concrete Answer modules. Thus, the purpose of the refactoring is to decouple IO from concrete question/answer classes.

**Experiment Design** We recruited 18 participants (two PhD students, 12 master students, and four senior undergraduate students) from the school of CS at Fudan University. Before the study, we conducted a survey to investigate their backgrounds. Six of them were experienced in refactoring, the other 12 were familiar with OO design but had little experience in refactoring. Based on the survey, we divided the 18 participants into two skill-equivalent groups, an experimental group (EG) and a control group (CG).

In order to evaluate the recommendation function of RN, the experimental group used all the features of RN. The control group used a customized version without the recommendation function. Moreover, the control group used JDelloworld [13, 21], which we call JD for short, a state-of-the-art automatic refactoring tool to identify refactoring opportunities by code smell detection, and to complete atomic refactoring steps automatically so that the participants do not need to conduct the entire refactoring manually.

Before the experiment, we provided a 1.5-hour tutorial and another 1.5-hour of practice for both groups to learn the concept and function of refactoring model generation. In addition, the experimental group learned the refactoring recommendation function of RN, and the control group learned JD. We also explained both the subject source code to be refactored and the target design to all the participants. We demoted the functionalities of the system and explained the responsibility of each module in the target design.

Afterwards, we conducted the two phases of the controlled experiment as follows. In the first phase, to test the reflexion model generation function, all the participants were provided with the target design as shown in Figure 3. After that, each participant provided a module description and ran the automatic mapping function of RN to generate a reflexion model accordingly. We then collected the module descriptions and evaluated the accuracy of the automatically generated clusterings.

In the second phase, all the participants were given the same, correct mappings so that they have the same starting point to refactor. They were asked to refactor the subject system towards the target design, and we recorded the time spent to complete the refactoring. We designed a test suite consisting of 50 test cases to test behavior preservation after refactoring, so that we can evaluate the accuracy together with efficiency.

All the participants were asked to run a full-screen recorder during the experiment, which enabled us to analyze the behaviors of each participant after the experiment. We also added behavioral monitors to the tool, to monitor and record the participants’ behaviors (e.g., accepting/rejecting recommendations). Based on our initial analysis of their behaviors, we interviewed some of the participants to get more accurate explanations about their behaviors. A video of how a EG participant performed refactoring using RN is available at [4].

In the experiment, the GA-based mapping process of RN was configured as: \(w_1 = 0.5, w_2 = 0.5, N_{gen} = 500, N_{pop} = 50, P_c = 0.5, P_m = 0.01\). The search-based recommendation process was configured with the following parameters: \(descNum = 5, T_{ref\_sim} = 0.6, \alpha = 0.2, \beta = 0.2\).

**Phase 1: Reflexion Model Generation.** We evaluated the accuracy of the automatic mapping by comparing it with the authoritative mapping provided by the authors, which is calculated as \(|\text{Class}_{a}|/|\text{Class}_{t}|\). \(\text{Class}_{a}\) represents the set of classes/interfaces correctly mapped to corresponding modules and \(\text{Class}_{t}\) represents the set of all the classes/interfaces.

Of the 18 participants, 4 of them mapped all the classes/interfaces correctly to the target modules, and 12 participants only incorrectly mapped one or two classes/interfaces. The mean value of the accuracy is 0.913, the first quartile is 0.875, the median is 0.938, and the third quartile is 0.969, indicating most students can use RN to obtain a reasonably correct mapping. There was one outlier with a low accuracy of 0.625, caused by vague module descriptions specified by the participant. For the Concrete Question and Concrete Answer modules, she just described them as “question” and “answer” respectively.

**Phase 2: Refactoring Navigation.** The second phase of the experiment evaluates the main contributions of RN w.r.t. productivity, and the contribution of the refactoring path recommendation and feedback loop to overall success.

**RQ1: Productivity Results.** Table 1 shows the results of the two groups in terms of completion time in the second phase (Time), the number of remaining inconsistencies (|INC|), the number of passed test cases (|TC|), the number of times applying automatic refactoring (|AR|), and the lines of codes manually edited (ELOC). The table also shows all the participants eliminated all the inconsistencies between the target and original modular structure. More data can be found at [2].

| Par(EG) | Time(m) | |INC| | |AR| |ELOC| |
|----------|---------|-----------------|-----------------|-----------------|-----------------|
| P1       | 16.617  | 0               | 50              | 12              | 11              |
| P2       | 14.717  | 0               | 50              | 12              | 5               |
| P3       | 8.033   | 0               | 50              | 12              | 5               |
| P4       | 7.583   | 0               | 50              | 12              | 5               |
| P5       | 10.333  | 0               | 50              | 12              | 5               |
| P6       | 9.300   | 0               | 50              | 12              | 5               |
| P7       | 15.350  | 0               | 50              | 12              | 5               |
| P8       | 31.317  | 0               | 50              | 12              | 5               |
| P9       | 11.183  | 0               | 50              | 12              | 5               |
| Avg      | 13.270  | 0               | 50              | 12              | 6.0             |
| SD       | 6.895   | 0               | 0               | 1.9             | 354.1           |

The table clearly shows that, on average, the experimental group accomplished the refactoring task much faster—13.270m (EG) vs. 58.619m (CG), and more accurately—all 50 (EG) vs. 36.44 (CG) test cases were passed, and with much less effort—6.0 LOC (EG) vs. 35.41 LOC (CG). We applied Wilcoxon’s matched-pairs signed-rank test, which also shows that the difference is statistically significant both in terms of time (p=0.008) and the number of test cases (p=0.018).

To understand why only 2 out of 9 CG participants passed all the test cases, we analyzed their refactored programs, and observed various mistakes in different places. For example, some partici-
pents in the control group forcefully modified the code to remove structural inconsistency, but ignored the need to preserve code behaviors. Therefore, their submitted code failed many test cases, even though it reached the target design. Due to the lack of refactoring navigation, the participants in the control group were more likely to make incorrect or incomplete refactorings.

**RQ2: Contribution of Automatic Path Recommendation.**

To investigate the impact of this feature on performance improvement, in addition to the number of auto-refactoring applied (#AR) in Table 1, we also collected the following data (Table 2) from the screen captures:

<table>
<thead>
<tr>
<th>Group</th>
<th>#Ite</th>
<th>#CSC</th>
<th>#RLD</th>
<th>#CG</th>
<th>#CRM</th>
<th>#CSD</th>
<th>#IAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
<td>3</td>
<td>11</td>
<td>P10</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
<td>14</td>
<td>1</td>
<td>P11</td>
<td>38</td>
<td>2</td>
<td>144</td>
</tr>
<tr>
<td>P3</td>
<td>8</td>
<td>12</td>
<td>6</td>
<td>P12</td>
<td>14</td>
<td>8</td>
<td>26</td>
</tr>
<tr>
<td>P4</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>P13</td>
<td>4</td>
<td>3</td>
<td>62</td>
</tr>
<tr>
<td>P5</td>
<td>8</td>
<td>11</td>
<td>3</td>
<td>P14</td>
<td>5</td>
<td>15</td>
<td>43</td>
</tr>
<tr>
<td>P6</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>P15</td>
<td>5</td>
<td>5</td>
<td>57</td>
</tr>
<tr>
<td>P7</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>P16</td>
<td>15</td>
<td>3</td>
<td>77</td>
</tr>
<tr>
<td>P8</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>P17</td>
<td>4</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>P9</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>P18</td>
<td>3</td>
<td>3</td>
<td>90</td>
</tr>
<tr>
<td>Avg</td>
<td>5.89</td>
<td>6.56</td>
<td>4.78</td>
<td>Avg</td>
<td>12.78</td>
<td>5.00</td>
<td>49.89</td>
</tr>
<tr>
<td>SD</td>
<td>1.73</td>
<td>4.45</td>
<td>3.15</td>
<td>SD</td>
<td>10.86</td>
<td>4.03</td>
<td>41.88</td>
</tr>
</tbody>
</table>

| Table 2: Behavioral Statistics of the Two Groups |

From EG, we collected the number of iterations (#Ite)—the number of times the user provided feedback and #CSC recaptured a "path" (i.e., a series of refactoring steps), the number of times they checked the source code (#CSC) which indicates the RN did not provide sufficient information so the student had to check the source code — , and the number of times they used the checking dependency function of RN (#CkD).

From CG, we collected the number of times they checked the reflexion model (#CRM) to update the divergence/convergence after the code was changed, the number of times they used the code smell detection function of JD (#CSD), as well as the number of times they used the checking dependency function of RN (#CkD). We did not collect #CSC for CG because they worked with source code all the time.

These data reveals the following facts: The EG participants used the automatic refactoring function more often: each EG member applied 12 auto-refactoring of RN (i.e., accepted recommended refactoring steps 12 times), while 5 out of the 9 CG members never used the auto-refactoring provided by JD, and they refactored by manually going through hundreds of LOC. P14 is an exception, who executed 10 automatic refactorings and edited 390 lines of code. Her video record and program revealed that most of the automatic refactorings she executed are incorrect and she manually edited a lot of code to correct the refactorings.

More interestingly, we observed that in the last few steps of the refactoring, the participants in EG manually edited the code to eliminate the divergent dependency from the IO module to the Concrete Answer module. The recommended refactorings could eliminate the majority of code dependencies from IO to Concrete Answer, but there remained five type references to concrete answer classes, which could not be eliminated automatically. The only way to remove the discrepancy was to manually replace the remaining concrete class references with the abstract answer class. In this study, the participants in the experimental group check code with the support of the examining dependency functionality provided by RN, and slightly modified the code to complete the task. As we will demonstrate in our industrial case study, in some design-level refactoring, human intervention is inevitable. Nevertheless, this extra effort is minor: the EG members only edited (including moving) 5 to 11 lines of code to complement the auto-refactorings. By contrast, the CG group edited 354.1 LOC on average.

The participants in the experimental group used 3 to 8 iterations to accomplish all 12 refactorings. Some participants were more aggressive and accepted most of the recommended refactorings for fewer iterations, while others were more cautious and only accepted few recommended refactorings and expected improvements in accuracy after RN learned from them. They checked source dependencies behind a divergence much less frequently than CG (4.78 vs. 49.89 times on average). Surprisingly, none of them ever rejected recommended refactorings. They ignored more often than rejected, even when they thought a recommended refactoring was incorrect. Our post-study interviews indicate a tendency to provide positive rather than negative feedback. Thus, RN may not recommend a rejected refactoring anymore. Giving no feedback on those "unsure" refactoring steps would leave more choices for next iterations.

The participants in the control group checked updated reflexion models between 4 and 38 times to examine the consistency between the desired structure and the source model. They used code smell detection provided by JD from 1 to 15 times, but seldom applied corresponding automatic refactorings (see the #AR column in Table 1). In the post-study interviews, the CG participants mentioned that JD reported a lot of refactoring opportunities in terms of code smells (e.g., God Class and Feature Envy), but many of them did not lead towards the target design. As a result, they soon felt overwhelmed and believed that checking source code was more straightforward and effective, and they checked source dependencies frequently, 50 times on average.

This checking process was usually inefficient, as there could be many code dependencies corresponding to a convergence or divergence dependency. For example, there are more than 20 method calls from IO to Concrete Question in the original source code. Moreover, there is no automatic way to check the absence of dependencies, so they have to explore their own path to reach the target, and deal with various problems, such as compilation errors. This explains why they needed more time and had more mistakes. In summary, the auto-refactoring and recommendation functions of RN contributed significantly to improvements in productivity.

**RQ3: Contribution of User Feedback.**

To evaluate the impact of user feedback, we compared the results obtained with and without user feedback. The participants in the experimental group provided similar feedback. We chose the most representative one for comparison. We produce the refactoring results without user feedback by running RN on the subject system and simply accepting all the recommended refactorings in order.

In the final refactored code with user feedback, one divergence dependency remains, from IO to the Concrete Answer module. As mentioned above, the dependency is caused by five type references to concrete answer classes in the code, and is easy to eliminate.

In the code refactored without user feedback, there is one divergence dependency and one absent dependency, corresponding to 13 dependencies in the code, including 12 method invocations from the IO class to concrete question classes, and one absent creation dependency, suffering from code tangling and poor design.

The above analysis shows a significant improvement in refactoring recommendation of RN after learning from user feedback.

**4.2 An Industrial Case Study (RQ4)**

Our research was motivated by a real industrial project which had severe maintenance problems. Two years ago we proposed a refactoring strategy, which was executed, but not by the authors. The refactoring process was difficult and time consuming due to the lack of description of the target design and proper tool support. The process took several weeks and its difficulty motivated our RN.
In order to evaluate how RN could have helped the architectural refactoring activity, we managed to retrieve the original source code after obtaining the permission of our collaborator (whose name has to remain anonymous). After that we drew the target design using RN, and evaluated whether RN can automate the refactoring that was conducted manually. Concretely, we investigate whether it is possible to fully refactor the original source code and completely replicate the manual refactoring. If not, to what extent can RN automate the process, and how much effort does it take to apply RN compared to the original manual refactoring. Next we introduce the basic characteristics of the subject projects. After that, we describe our refactoring process and the results.

For proprietary reasons, we call it Project X and changed all the sensitive names used in the project into animal names. Project X evolved for 8 years, and contains over 40K LOC, 148 classes, and was maintained by 4 developers. The main function of the system is to monitor data variation collected continuously in real time.

Since the project experienced years of evolution, many functions were added on demand without a proper consideration of the overall architecture. By the time we visited the company last year, the system was impossible to maintain: adding any new feature or fixing a bug would incur unexpected changes to many files. Although the project does not appear to contain large number of classes, many files are fairly long with more than 1,500 lines of code, indicating the existence of many God classes.

The system needs to monitor multiple types of data, and display them in different ways and in different views. The operation on one view can influence the display on other views, making the system highly coupled. The scope for refactoring involves 8 classes albeit 5,823 LOC, which typically suffered from the aforementioned design flaws, i.e., God class (i.e., Main class) and entangling dependencies between views (i.e., TreeView, TableView, and MapView), which almost form a complete graph. The entry class, Main.java, was responsible for a large number of business logic, such as user login, report generation, etc. It took us 3 days to understand the different relations among the code logic, and separate them from this God Class. Retrospectively, this is a typical problem where a model-view-controller pattern should apply.

**Refactoring Replayed using RN.** After retrieving the original source code, we drew the target design, and generated the reflexion model using RN. As shown in Figure 4, each class is treated as a module in the target design.

The target design follows three principles: 1) Separated Concerns: Different business logics, such as user login and report generation, should be separated from Main class, and new classes should be created for each of them, i.e., the empty UserLogin and Report Generation modules shown in Figure 4. 2) Simplified Entry Class: the entry class, Main, should not depend on the DBOperations module in any form. 3) Abstraction and Modularization: different views should have no dependencies among themselves. The container class, MainFrame.java should only create concrete view objects, but does not have other dependencies on them. In order to achieve this, the commonality of views should be abstracted.

The reflexion model shown in Figure 4 reveals the severe discrepancies between the original source code and the target design: there are 3 missing modules, 7 absence dependencies (dotted lines), and 11 divergence dependencies (dashed lines).

Next we applied RN to navigate the architectural refactoring. At first, RN presented a path with 8 steps. The first step suggested pulling up the highlightAnimal() methods in TreeView, TableView, and MapView into a newly created interface. This step is generally correct, but we prefer using an abstract class for the sake of reuse. Therefore, we rejected this step and asked RN to make a second recommendation. The tool then suggested a new path, also containing 8 steps, with the first step of pulling up the highlightAnimal() methods to a newly created abstract class.

RN then suggested the following steps: move several methods (e.g., refreshData()) from MainFrame to TreeView, TableView, and MapView to alleviate the divergence dependencies from MainFrame to these views, pull up common methods (e.g., clearAnimals()) in different views to the newly created abstract view, and extract different business logic (e.g., user-login) into new classes. Each step aims at improving the consistency of the reflexion model showed in Figure 4. In this case, we accepted all suggested steps, and obtained a reflexion model showed in Figure 5.

Figure 5 shows that most divergence dependencies are removed and all the absence dependencies are satisfied, except 3 remaining divergence dependencies from MainFrame to the other three views. The cause of the remaining divergence is the constructor of MainFrame, shown in Listing 1. Originally, each view kept two attributes of the other two views, e.g., TreeView contains a TableView and a MapView, and MainFrame initializes them in its constructor.

After extracting this commonality among all the views into the abstract class, the AbstractView refers to the same view type. However, the field access dependencies from MainFrame to the other three views still remain. Such dependencies cannot be eliminated by either moving a method, pulling up a method, or extracting a class, thus, RN could not fully recover the refactoring. In this case, we manually created a list, containing a collection of views, and made the list an attribute of the AbstractView class. Then we removed the concrete object instantiation from the constructor of MainFrame. Thus, all the divergence dependencies were removed.

Listing 1: Code for Unsolved Divergence

```java
public MainView() {
    ... 
    treeView.tableView = tableView;
    treeView.mapView = mapView;
    tableView.treeView = treeView;
    tableView.mapView = mapView;
    mapView.treeView = treeView;
    mapView.tableView = tableView;
    ... 
}
```

This case study similarly reveals the limitations of push-button, fully automatic refactoring at the architecture/design level: given the needs of creating complex data structures or further abstraction, human intervention is inevitable. Nevertheless, a great deal of manual effort was saved due to RN’s automated processes. It saved
us from going through messy God Classes where we spent most of our time during the manual refactoring process.

5. THREATS TO VALIDITY

The major threat is that we only used two subject systems: one for the controlled experiment and one for the case study. Our approach might not generalize to larger systems from other domains.

In our controlled experiment, the subject is not large, and the participants are students, rather than professional developers. The result may change otherwise. Other threats include that the two “equivalent” groups may not be truly equivalent, and the test suites used may not cover all aspects of the program behaviors. Finally, we only conducted one experiment with a Java program as the subject system. It is not clear if the results apply to systems written in other programming languages, or if the results may differ if more experiments are conducted.

In our industrial case study, we only reenacted the refactoring with RN on the project which was refactored manually two years ago. We cannot claim that the same results can be achieved for other legacy systems. RN currently only supports three atomic refactorings. Even though they appear to be sufficient for this case study, other systems may require different atomic refactorings.

In both evaluations, we assume that the users of RN have sufficient domain knowledge and are able to draw a reasonable Target Model, just like the users of existing reflexion tools. As we mentioned before, the impact of target model quality to RN is similar to the impact of destination input quality to a GPS. Further evaluation of the impact of target model quality, as well as the impact of user experience are important for our future work.

6. LIMITATIONS AND FUTURE WORK

Currently, RN only supports three atomic refactorings: Move Method, Extract Class, and Pull Up Field/Method. This limits the capability of our refactoring recommendation. We designed RN to be extensible, and will add new types of atomic refactorings.

In addition, RN only supports three dependency types to describe the target design. Our future reflexion model will be more expressive, e.g., differentiating different types of modules in target design and extending new dependency types. For example, the user should be able to differentiate UI modules, business logic modules, and data modules in the target design. Correspondingly, more specific dependency types such as asynchronous communication will be defined. We consider extending the domain-specific module and dependency types according to the characteristics of different systems, such as Web systems and Android apps.

7. RELATED WORK

Refactoring opportunity identification. Researchers have proposed various automatic approaches for identifying refactoring opportunities to ease adaption and extension [29]. Simon et al. [41] used distance-based cohesion metric to identify code smells and propose refactoring actions such as Move Method/Attribute and Extract/Inline Class. Tourwé and Mens [45] proposed an approach that uses logic meta programming to formally specify and detect refactoring opportunities. Bavota et al. [10] proposed an approach for recommending Move Method refactorings via relational topic models. Sales et al. [39] proposed a technique to recommend Move Method refactoring via static dependencies. Tsantalis et al. [46] proposed an approach for identifying Move Method refactoring opportunities with the objective of solving Feature Envy code smells. Bavota et al. [7–9] proposed a series of techniques for identifying Extract Class opportunities.

Our approach differs in our aim to refactor towards a target structure at a higher-level, whereas these techniques can be integrated into RN to identify atomic candidate refactorings.

Reflexion models. Murphy et al. [32] introduced software reflexion models, which can be used to support a variety of software engineering tasks, such as design conformance checking [38], reengineering [33], and consolidating software variants into product lines [24]. Reflexion models in these approaches reveal inconsistencies between high-level model and implementation, but provide no explicit recommendations to help developers decide how to eliminate the inconsistencies. In contrast, our approach starts from reflexion models, then fills all the essential gaps in order to produce code that adheres to the desired design.

Design-level refactoring. Moghadam et al. [31] proposed an automatic technique to refactor the source code towards a target UML-based design with a design differentiating technique. Terra et al. [42, 43] proposed an architectural repair recommendation system that can recommend refactorings to remove architectural violations based on predefined rules. In contrast, our approach recommends refactorings using search-based algorithms and supports user interaction and feedback.

Search-based refactoring. Previous approaches [19, 30, 36, 37, 40] used search algorithms to find an optimal refactoring solution for a system, with the goal of maximizing the fitness function defined by a set of design metrics. Therefore, the improved design structure after refactoring may not conform to the desired structure the developer has in mind, or the required structure previously specified. In addition, these approaches do not support user interaction or take user feedback into consideration.

Recently, some researchers propose to incorporate user feedback in automatic refactoring approaches. Liu et al. [26] proposed a monitor-based instant refactoring framework, which can use feedback to optimize code smell detection algorithms. Hall et al. [18] proposed an approach that enables the user to refine the refactoring results produced by unsupervised learning approaches. Bavota et al. [6] proposed an approach that uses interactive genetic algorithms to integrate developers’ knowledge in refactoring tasks. Although these approaches involve interactive feedback, the refactoring is still driven by code smell detection or design metrics and, unlike RN, these approaches do not steer the solution towards the high-level design goals that a user has in mind.

8. CONCLUSIONS

We present RN, an interactive refactoring navigation system. It calculates a reflexion model to reveal the discrepancies between the target design and the given source code, recommends stepwise refactoring path toward the target design, and takes user feedback into consideration. We first evaluated the effectiveness of the approach with a controlled experiment involving 18 students. The results show that RN helps developers perform their refactoring tasks more effectively and efficiently. Our industrial case study further suggests its potential to help with architectural refactoring in practice. In the future, we will support other atomic refactorings and conduct more industrial case studies.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers and Sean McDonald for insightful comments on earlier drafts. This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000800, National Natural Science Foundation of China under Grant No. 61370079, and the U.S. NSF CCF under grants 1439957, 1553741, 1065189, 1514315, and 1514561.
10. REFERENCES


