USING REINFORCEMENT LEARNING IN ENHANCING SOFTWARE REFACTORING PRIORITIZATION

BY

ARMIN KOBILICA

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

DECEMBER 2017

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **ARMIN KOBILICA** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

[illegible]
Dr. Lahouari Ghouti (Adviser)

[illegible]
Dr. Mohammad Alshayeb (Member)

[illegible]
Dr. Sajjad Mahmood (Member)

[illegible]
Dr. Khalid Al-Jasser
Department Chairman

[illegible]
Dr. Salam A. Zummo
Dean of Graduate Studies

8/1/18
Date

عمادة الدراسات العليا
جامعة الملك فهد للبترول والمعادن
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
1963
DEANSHIP OF GRADUATE STUDIES

Dedicated to my family

ACKNOWLEDGMENTS

First, praise to Allah, the Almighty, on whom ultimately we depend for sustenance and guidance. I would like to thank my advisor, Dr. Ghouti, for his patience, understanding, valuable help and the idea for the thesis work. I would like to express my deepest appreciation to Dr. Alshayeb for his constructive comments, valuable feedback and the continuous involvement in this work. The comments and criticism of Dr. Sajjad are highly appreciated and have greatly improved the quality of this thesis work. Most importantly, I wish to thank my father, mother and sisters for their endless support. This work would not have been possible without my loving and supportive wife and my children.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME:	Armin Kobilica
TITLE OF STUDY:	Using Reinforcement Learning in Enhancing Software Refactoring Prioritization
MAJOR FIELD:	Computer Science
DATE OF DEGREE:	December 2017

Background: Sequencing refactoring tasks play an important role in maximizing the benefits of refactoring in software development. Several techniques are proposed for the refactoring automation based on different preferences, yet far from reaching the optimal solutions.

Objective: We employ reinforcement learning (RL) techniques to automate the refactoring sequencing process. Automation is carried out to optimize coupling and cohesion at the class-level.

Method: The proposed solutions are developed and integrated as an Eclipse plugin and validated through case studies with detailed analysis and discussions.

Results: Empirical evaluation shows that the proposed methods contribute to automating refactoring while achieving the targeted software quality measures.

Conclusion: RL techniques are shown to be fit to the dynamic nature of the problem of refactoring sequencing. Both planning- and learning-based RL techniques attained efficient results with acceptable execution time and space requirements.

ملخص الرسالة

الإسم الكامل: أرمين كوبيليتسا

عنوان السالة: استخدام التعلم بالتعزيز في تحسين تحديد الأولويات في إعادة هيكلة البرمجيات

التخصص: علوم الحاسب الآلي

تاريخ الدرجة العلمية: ديسمبر 2017 م

الخلفية: يمثل وضع تسلسل مهام إعادة هيكلة البرمجيات دورا هاما في تحسين فوائد إعادة الهيكلة في عملية تطوير البرمجيات. تم اقتراح العديد من التقنيات لأتمتة إعادة هيكلة البرمجيات على أساس تفضيلات مختلفة. لكن هذه التقنيات لم تحقق بعد حلول مثلى لهذه المسألة.

الهدف: نقترح توظيف تقنيات التعلم بالتعزيز (reinforcement learning) لأتمتة وضع تسلسل مهام إعادة هيكلة البرمجيات. يتم تنفيذ الأتمتة عبر مقاييس الإقتران و والتماسك على مستوى الفئة.

الطريقة: يتم تطوير الحلول المقترحة و دمجها كبرنامج مساعد في بيئة التطوير المتكاملة المعروفة بإسم إيكلبس (Eclipse). كما تم التحقق من صحتها من خلال دراسة حالات مع تحليل مفصل ومناقشات.

النتائج: يظهر التقييم التجريبي أن الحلول المقترحة تسهم في أتمتة إعادة هيكلة البرمجيات مع تحقيق مقاييس جودة البرمجيات المستهدفة.

الخلاصة: أثبتت هذه الرسالة ملائمة تقنيات التعلم بالتعزيز للطبيعة المتغيرة لمسألة وضع تسلسل إعادة هيكلة البرمجيات. حيث أن تقنيات التعلم بالتعزيز المعتمدة على التخطيط و التعلم حققت نتائج فعالة مع وقت التنفيذ مقبول و متطلبات تخزين في الذاكرة بسيطة.

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

Software projects have a tendency to become very complex in very short time frames. Complex projects are challenging to maintain and become rapidly obsolete with low reusability levels. This phenomenon, known as the *technical debt*, affects all software stakeholders. One of the effective ways to minimize this debt consist of restructuring existing projects on a regular basis. This restructuring is known as *refactoring process*. While the refactoring process restructures existing code without adding new functionalities, software practitioners strive to maximize this process's effect with minimum effort and time spent. The success of the refactoring phase relies heavily on the refactoring task prioritization. Prioritization can be done based on different goals (time, effort and quality) and usually requires feasible automation. Automation of refactoring prioritization is not a trivial task due to the many complex factors involved (infinite search space, human-written code, developers' practices, coding standards, software project's goals and natures). Despite different techniques proposed in the literature to automate prioritization, no practical implementation is available to assess the validity of these techniques. In addition, the prototypes based on optimal prioritization are lacking.

1.2 Motivation

It is a software engineering dream to have a codebase "coded once" and used many times with slight updates for different purposes and projects. This would mean less time for testers, less cost for project managers, more optimized code for developers. In fact, such codebase will be easy to understand, reuse and maintain. On the other hand, for clients, such codebase will perform better, work faster and cost less. But the reality of software engineering projects reality says otherwise; technical debt has to be paid sometime in the process of code reusability and trade-offs have to be made where not all software-related stakeholders will be satisfied at the same extent, Traditionally, refactoring is defined as changing the internal structure of code without changing its external behaviour [1]. These changes aim to improve software for easier maintenance, better readability, and more reusability. Favorably, the same changes will lead to "simpler" software, cost-effective maintenance and enthusiastic developer attitude towards software improvement and future functionality. In research and software community, refactoring is tackled, using different techniques where effects are analyzed via theoretic and empirical procedures. In these techniques, refactoring is performed in semi-automated and automated fashions. Initially, refactoring was solely performed at the code level, while later many efforts focused on design-based approaches.

Recently, the software refactoring problem is formulated using artificial intelligence (AI) and optimization techniques for improved process automation based on different objective functions.

AI and optimization-based refactoring approaches are still in their infancy and this work aims at contributing to AI-based refactoring research by exploring prioritization of refactoring tasks based on different software quality preferences using Markov decision processes (MDP) and reinforcement learning (RL) solutions.

Prioritization of refactoring tasks intends to efficiently utilize the developer time by classifying refactoring tasks according to the "goal-oriented" preferences of the developer. For programmers, according to Pinto and Kamei [2], the programmers expect "good" refactoring tools to provide "optimal" refactoring recommendations.

The maintenance and improvement of large-scale software projects are challenging tasks and optimizing their time requirements is highly desirable. Refactoring recommendation systems, based on specific quality preferences, would assist developers in managing efficiently their time and to ensure that the maintenance/improvement goals are met within the time constraints.

Refactoring recommendation systems, without prioritization, focused on improving certain aspects of system quality, more likely to produce a large number of general refactoring tasks whose overall effects might be difficult to assess.

Moreover, it might affect developers to decrease their desire for maintaining and improving software that seems to have "no ends". Therefore, once the refactoring tasks are prioritized, with estimated trade-offs between different maintenance/improvement goals, developers will be able to clearly identify where to focus their efforts to achieve their quality objectives.

1.3 Research Objectives

The main objectives of the research work, carried out in this thesis, are:

- Formulate the refactoring prioritization as an RL problem where refactoring automation can be achieved.
- Propose different RL-based algorithms to automatize refactoring prioritization.
- Analyze the effects of the proposed RL-based solution on the refactoring automation process.

1.4 Research Methodology

The refactoring process is characterized by, at least, two distinct phases: 1) detection of bad smells and 2) application of identified refactoring tasks. While the first phase has been investigated extensively in the literature [3], the phase related to identifying the appropriate refactoring tasks and their sequencing has not been the focus of thorough research and investigation. In this thesis, our main focus is geared towards developing intelligent and optimized methods for automated refactoring processes. To ensure equal foot comparison between the refactoring processes, all empirical studies, reported in this thesis, are based on Java open source projects. The source code of these projects will form the basis for the proposed modeling where this source code is used as an input to the proposed refactoring model as transitioning states in a dynamic environment. More specifically, this dynamic environment is formulated using an MDP framework and RL algorithms. Along with the source code, the proposed models are fed with software quality preferences to produce optimal sequences of refactoring tasks to achieve the software developers' goals in terms of maintenance time and software quality measures.

1.5 Research Contributions

The main contributions of this thesis are listed below:

- A novel formulation of the automated refactoring sequencing problem using RL-based representation.
- An automated tool developed based on methods proposed.
- Empirical evaluation of proposed methods based on cases studies and different experimental settings.

1.6 Thesis Outline

The remaining of the thesis is organized as follows: The second chapter discusses the required background and theoretical aspects of the thesis work where software bad smells, refactoring and MDP/RL concepts are introduced with basic definitions and examples. Then, the literature review is laid out in Chapter 3 where we highlight existing research work. The proposed methods to automate and prioritize software refactoring are discussed in Chapter 4 where we show the formulation of the refactoring problem as an MDP process. The rationale behind such formulation is provided therein. Chapter 5 provides a detailed description of the software tools used to implement the proposed solutions where we introduce, *RefMark*, the plugin implementation of these solutions for the *Eclipse* integrated development environment (IDE). The experimental design for the empirical evaluation of the proposed solutions and the *RefMark* tool is outlined in Chapter 6 along with the standard procedures adopted in the empirical evaluation. This chapter concludes with the hypothesis testing and research questions addressed in this thesis. Detailed analysis and discussions of the results attained by the proposed MDP/RL-based solutions are given in Chapter 7. Cautionary notes on the recommendations, formulated in this thesis, are provided at the end of this chapter. Finally, Chapter 8 concludes this thesis where conclusions are drawn, a brief summary of the work is given and possible extensions to the work are summarized.

CHAPTER 2

BACKGROUND

Over the last decades, software practitioners have gained more insights on maintaining and improving large scale software projects where patterns of "*proper*" software source code organization and design emerged. For example, the object-oriented programming paradigms (OOPP) include the *substitution principle*, the responsibility-driven design, the law of diameter concept, and the open-closed principle. Most of these principles and concepts led a generation of software engineers to create design pattern-based test cases and regularly refactor their software projects [4].

During their evolution, software projects, especially the large ones, undergo changes and improvements to become more efficient and optimized. The same time, more software developers, embarking on the development teams of these projects, need to communicate and understand code written by others. As indicated by Lehman and Belady, the continuous software evolution is tightly connected to its success [5]. This evolution emphasizes the importance of flexible, maintainable and easy-to-communicate software design. These desirable features of successful software projects lay the floor for the following inevitable questions:

1. How can these attributes be detected and identified in software design?

2. What are the main properties of such software design?

It is must be admitted that the answers to these questions are not an easy matter since they cannot be merely based on simple quantitative attributes as they require software projects to adapt to design patterns, be modularized, include a detailed documentation, follow the adopted architecture.

Researchers, in the field of software engineering, have exerted commensurate efforts to detect obsolete and inadequate components in software systems. To successfully detect these components, different measurements and software metrics are proposed to capture the inherent software design properties. In fact, these efforts resulted in a plethora of software metrics and standardized design patterns where design violations are quantified using the concept of *bad smells*.

The widely accepted software metrics, typical bad smells, and recovery from design pattern violations (i.e., *refactoring*) are introduced and discussed later in this chapter. Following the discussion on manual refactoring, *automated refactoring* is introduced along with justifications for its usefulness.

Finally, the work presented in this thesis consists of applying emerging decision-based machine learning algorithms to the automated software refactoring problem, machine learning concepts related to Markov decision processes (MDPs) and reinforcement learning (RL), necessary to our work, are outlined in the remaining sections of this chapter.

2.1 Bad Smells

Thanks to the bad smells, software developers quantify *code states* that are tightly connected to code anomalies and bad practices during the software development cycle. As such, these bad smells represent design flaws and programming practices that do not strictly follow the adopted development paradigm. For instance, mixing between different programming approaches such as procedural thinking in an OOPP-based code constitutes a deviation from the adopted development paradigm. In the literature, bad smells have been also known as anomalies [1], code smells, design flaws [6] and anti-patterns [7]. Design flaws and code smells are related to the design and code, respectively. As indicated by their name, bad smells are only "*smells*" that usually do not cause direct program failures. However, they are strong indicators that software bugs are likely to happen in the future which may cause heavy burdens on software maintenance and reusability.

Despite, bad smells were initially specified using other programming paradigms, most of those reported in the literature are related to the OOPP one [8]. Fowler et al., among the first to work on bad smells, identified 22 sets of symptoms related to code smells and listed possible solutions to improve each of the identified smells [1]. Following the pioneering work of Fowler et al., novel methods to detect bad smells are attributed to different authors [9, 10, 11, 11, 12]. In addition to the proposed detection methods, the effects of bad smells on software defects [13, 14] and effort [15] were empirically investigated.

In general, bad smells are often addressed by restructuring and reorganizing the source code following a formal software refactoring process as discussed in the next section.

2.2 Software Refactoring

Software refactoring aims at mitigating the negative effects of bad smells by changing the internal structure of the software source code without changing its external behavior [1].

Pioneered by Opdyke, in his 1992 Ph.D. thesis, refactoring is proposed as a notion of reorganizing the source code for possible future improvements and restructuring [16]. These code improvements and restructuring tend to extend the software expected lifetime by increasing the software quality in terms of reusability, maintainability, and portability.

In its initial formulation, software refactoring is executed using manual, semi-automated and automated approaches. Prior to applying refactoring, code components, eligible for refactoring, must be detected/identified first. Then, a suitable set of refactoring tasks must be proposed to improve the identified code components. These two common refactoring steps are graphically depicted in Figure 2.1. However, finding the appropriate set of refactoring tasks represents a major challenge that most software engineering practitioners must undertake given the astronomical number of the possible refactoring combinations to be tried.

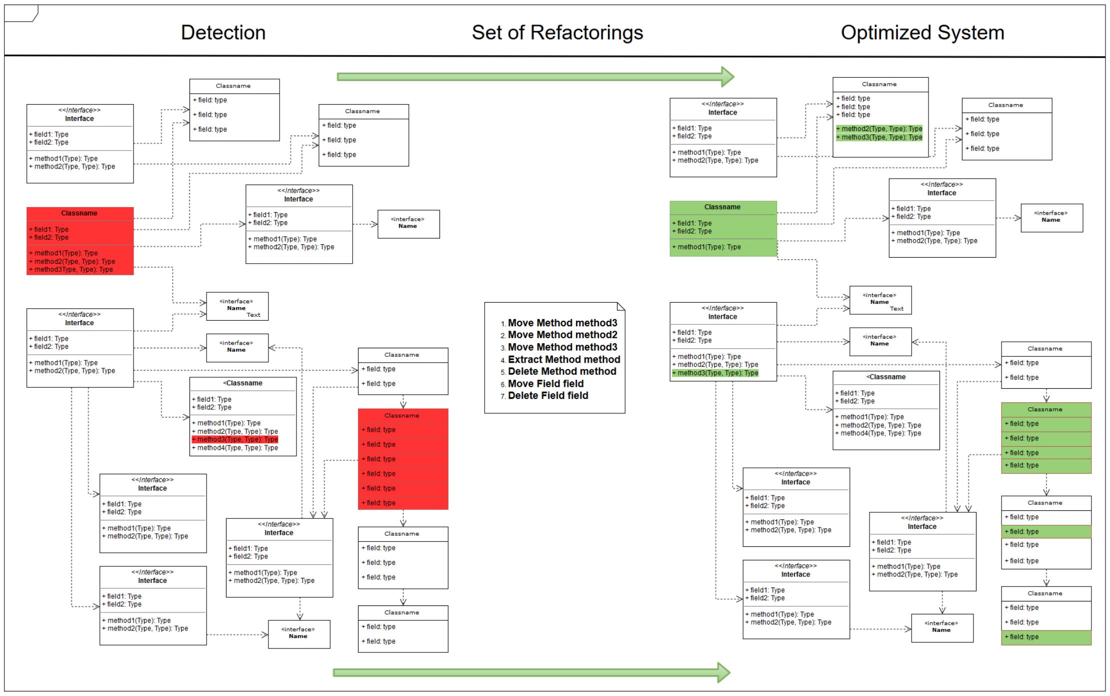


Figure 2.1: Refactoring Process.

In this thesis, we propose a new approach for automating software refactoring using machine learning formulations based on MDP and RL concepts. These concepts, encompassed by the ML field, are introduced below along with a technical background on the required ML models and frameworks.

2.3 Machine Learning

Machine learning enables computational machines to learn from the surrounding environments using the following paradigms:

1. Supervised learning.
2. Unsupervised (or semi-supervised) learning.
3. Reinforcement learning.

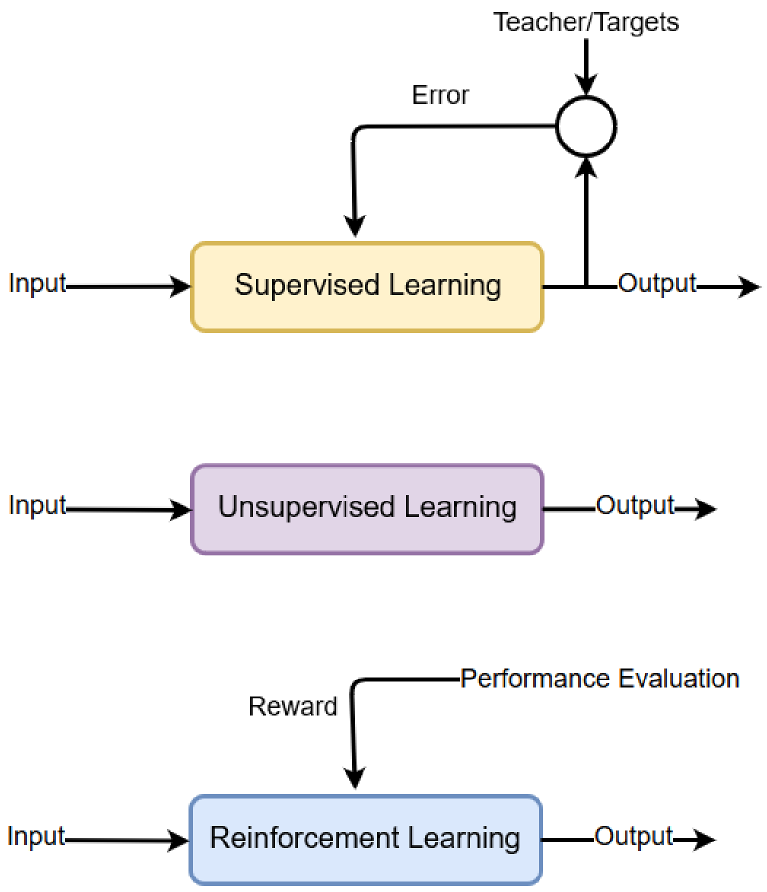


Figure 2.2: Machine Learning Paradigms.

Figure 2.2 depicts the main types of ML algorithms given above. It is clear that the behavior and implementation of a specific ML rely heavily on the type of "*target*" attributes that the ML algorithm tries to achieve or replicate.

An ML process or pipeline is sketched in 2.3 where the main pipeline steps are given. Data selection and modeling are crucial to a successful deployment of the ML pipeline. In this thesis, this step is crucial in the MDP formulation of the refactoring process.

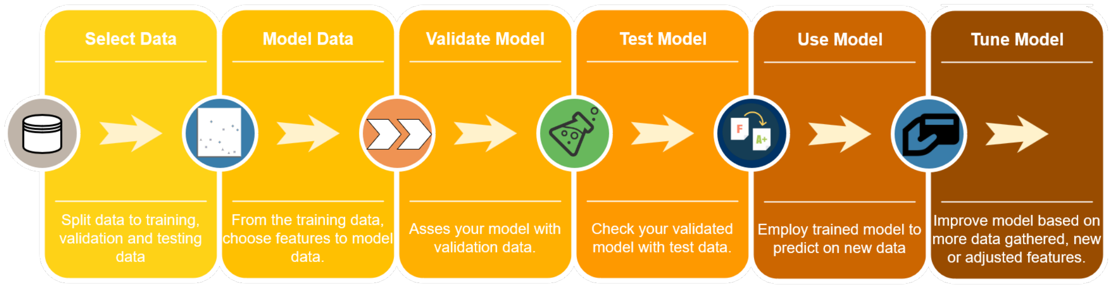

Select Data
Split data to training, validation and testing data
Model Data
From the training data, choose features to model data.
Validate Model
Asses your model with validation data.
Test Model
Check your validated model with test data.
Use Model
Employ trained model to predict on new data
Tune Model
Improve model based on more data gathered, new or adjusted features.

Figure 2.3: ML Process/Pipeline Summary.

2.4 Reinforcement Learning

RL is a category of ML algorithms that is inspired by physiological theory of human reactions to immediate stimuli. It consists of a family of algorithms that are based on similar concepts of immediate response (either positive or negative) from an environment on actions taken.

2.4.1 Fundamentals

Learning from Interaction

Instead of exposing an agent (the common name for the learning entity) to correct relations in data to be able to predict from it in the next possible occurrence, the RL agent is learning from direct interaction with environment following its immediate reward and cost. An agent can be a robot, player or any entity aims to plan or learn from the domain while environment can represent a real world, a game or any domain an agent operates in. Getting an immediate answer from the environment, in terms of reward or cost, the goal of the agent is to maximize its cumulative reward while interacting with the environment. For example, after a time step t, and n actions, agent obtained reward:

$$R_t = r_{t+1} + r_{t+2} + r_{t+2} +_{...} + r_{t+n} \tag{2.1}$$

This is the most basic case of agent's cumulative reward calculation where knowing the number of steps and its immediate reward we simply calculate the sum of rewards gained. More often, the reality of environment requires more detailed and more complex representation with more information to consider. In next lines, we will introduce more core ingredients of RL techniques.

Markov Property

The success of RL is based on good interaction with the environment and quick absorption of knowledge it gives us at each step. To maximize its learning experience, from each step, an agent wants to be able to use all of its gain from previous steps taken to decide the best possible action for next step. This requires that agent memorize all of its history passed in a certain environment. From this history, reward and next state reached is of the particular interest to quickly learn environment and its response. Having all history of agent, letting him decide, based on it, before each step taken, may be wonderful and perfect way of learning experience, but usually it is not win-win solution when it comes to reality and efficiency of the same agent. Markov property allows the agent to retain all past experience in a single, current state in very convenient and compact form and let him decide for next step by a simple consideration of its current state.

Markov Decision Processes (MDPs)

After understanding RL basic philosophy and concepts of agent's interactive learning from the environment, its desire to retain all gained experience in a current state, we need a solid formal framework to represent our RL technique. A Markov Decision Process (MDP) is defined as a formal representation of RL that meets Markov Property.

Commonly, MDP is defined as a tuple [17]:

$$MDP = \left\{ S, A, P(s, s^{'}, a), R(s, s^{'}, a) \right\} \tag{2.2}$$

where:

S: State space to "*encode*" the agent world or environment.

A: Set of all possible actions that the agent can take at any state to end up in the same or different state.

$P(s, s^{'}, a)$: State transition probability function is given by:

$$P\left(s, s^{'}, a\right) = P\left(s_{t+1} = s^{'} |\ s_t = s, a_t = a\right) \tag{2.3}$$

Eq. 2.3 defines the probability of ending up in state $s^{'}$ while being at state s and taking action a.

$R\left(s, s^{'}, a)\right)$: Reward obtained while being at state s, taking action a and landing in state $s^{'}$. In simple MDP formulations, the reward function, R, is independent of the action taken a.

γ: A discount factor to enable the mathematical tractability of the MDP formulation and influence the MDP valuation of the future rewards.

Various approaches in each of these MDP parts generate different algorithms. How specific problem is addressed and translated into these categories is crucial for its successful RL story.

Finite and Infinite Horizon

Looking closer at the definition of an MDP, we found that MDP has the state probability function. The logical question is why do we need probability distribution to calculate the sum of reward gained in n steps of agent interaction with environment?! The truth is that RL instance of any real problem to be solved rarely has a perfect environment with all possible information available including possible actions, their expected outcome, states, number of steps, etc.

If the number of steps is known in advance for certain environment, we have a case of the so-called *finite horizon*, where we know that after certain steps, no change will happen by any action and life of the agent ends. Furthermore, if the time is considered, environment with limited timestep is also an environment with a finite horizon. If the number of steps is not known in advance, or no time limit is set for the life of the agent in a certain environment, we have so-called *infinite horizon* RL model. Infinite horizon instances of RL gained a lot of attention in practice and variety of algorithms and improvements were proposed in the literature. Our particular interest, in this thesis, is in the MDP infinite horizon environment with discounted reward. Formally defined, discounted reward function represents:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} \ + \cdots \tag{2.4}$$

where γ is a discount factor with values between 0 and 1 inclusive to favor immediate over future rewards. Setting γ to zero will result in a "*future-myopic*" preferences that ignore any future reward regardless of its value. On the other hand, $\gamma = 1$ ensures that the resulting preference does not differentiate between current and future rewards that are deemed equally preferable regardless when they are achieved. Usually, γ is set to 0.9 in most instances such that near rewards gained are preferred over rewards gained in the remote future. In addition, such values allow achieving "*natural preferences*" of human beings whose attitudes tend to value current rewards more than those achieved in the future [17].

The state probability function also addresses state-action pair by its likeness of occurrence and suitable probability for that pair.

Under probability function is also counted stochastic nature of the environment where the agent possibly not always executes "the right" action or not always reaches the same state from its current state using the same action. If that is the case, we have non-deterministic MDP environment where, with some percentage, the agent is considered a not reliably predictable entity. If the agent always executes intended actions, without failures, or next state can be reliably predicted given the current state and action taken, we have the deterministic environment.

Once discounted factor entered reward function and having stochastic nature, our reward function is defined as expected discounted cumulative reward:

$$E(R_t) = E\ (\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}) \tag{2.5}$$

A careful look at the formula, we may note that the number of actions goes to infinity, but with discount factor infinite horizon has finite value where discount factor diminishes "a far" rewards taken basically making them near-to-zero values. Fortunately, discount factor allows infinite horizon being treated as a finite.

Discounted infinite horizon environment is of interest in this thesis and the finite horizon is out of its scope.

The expected cumulative reward for a certain state in the environment gave as its value function or simply the desirability of a certain state. Next, to the state-value function, we have also action-value function, denoted by $Q(s, a)$ where the utility is defined for the state considering it with the action taken from that state. Formally, state-value function:

$$V(s) = E\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \mid s_t = s\} \tag{2.6}$$

Action-value function:

$$Q(s,a) = E\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \mid s_t = s,\ a_t = a\} \tag{2.7}$$

The complete solution of an MDP is in a finding for the agent the correct action for every possible state of the environment. This complete solution or a map of action-state pairs represents *policy*, π. Formally: $\pi : S \rightarrow \Pi(A)$ policy maps between all states and its probability distributions over the set of actions. $\pi(s,a)$ - probability that the agent selects action a in state s [18]. Policy, in our case with Markov Property preserved, is a stationary as an action solely depends on a current state. Non-stationary policies define different actions for the same state depending on a time step as an environment considered is with finite horizon.

The agent objective is to maximize its reward on each state. The policy that contains maximum expected reward in every state is named *optimal policy*, π^*. Consecutively, the policy contains optimal value function, V^* for each state, and satisfies:

$$V^{\pi^*}(s) \geq V^{\pi}(s) \qquad \forall\pi\forall s\epsilon S \tag{2.8}$$

Optimal policy based on action-value functions, Q^*: $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ Optimal action-value function is the function that maximizes action-value function for each state-action pair. Finding the optimal policy requires finding optimal values for state-value function (or action-value function) for each state, means, that we have traversed all possible combinations. The core of solution for an

MDP laid down in a *Bellman Equation* that uses a recursive approach to update value function till its convergence, the optimal value. *Bellman Equation* for value function is represented as:

$$V^{\pi}(s) = \sum_{a \epsilon A} \pi(s,a) \sum_{s' \epsilon S} P(s,a,s')\ [R^{a}_{ss'} + \gamma V^{\pi}(s')] \tag{2.9}$$

Where MDP is defined as tuple $(S, A, P(), R)$, s is current state, S is set of all states, π is a policy and $P(s,a,s')$ is a probability that agent, starting from state s and taking action a will land in the next state s'.

As mentioned earlier, *Bellman Equation* is a recursive approach where the current state is updated based on its next state, as can be seen from the equation. The formula also gives us an insight of possible expensive and computationally high requirement of dependency on the calculation of each state value on all next state values till the last state. Also, $P(s,s',a)$, in the equation tells us that model of the environment and transition probabilities are known before we want to find the optimal solution, that is rarely the case. In reality, very few instances have clearly defined knowledge of dynamics of the environment and if the model of the environment is not completely known, finding optimal values may not be accurate with the *Bellman Equation.* If the accurate model of an environment is not available, dynamics of the environment depends on an estimation hence we have estimated value functions. Having all estimations learned by an agent interacting with an environment is one of the leading force behind the development of different RL techniques. All of these techniques have a goal, based on estimations, to found an optimal policy. In the next paragraphs, we summarize different approaches and algorithms for each approach that tries to solve an MDP.

2.4.2 Solving an MDP

Depending on the model availability, the majority of the algorithms for solving an MDP instance fall into two broad categories, planning-based and learning-based algorithms. Each of the categories has several algorithms with different variations and improvements considered.

Planning-based Methods

Planning-based methods for solving an MDP require that dynamics (a model) of the environment is known in advance. Once the model is available, dynamic programming approach is used to recursively calculate the optimal value. Previously discussed the solution of an MDP, *Bellman Equation*, is in the heart of dynamic programming techniques. The same drawbacks discussed are encountered in all other planning-methods since all of them are based on dynamic programming approach and its *Bellman Equation*. Admitting its disadvantages and fails in addressing real world problems, its well-defined theoretical background, soundness and concepts are crucial for grasping and comprehension of all RL-based techniques (planning and learning-based). Since the *Bellman Equation* calculating the value based on updates calculated from the all next states, varying the object of update we have different algorithms generated.

Policy Iteration Policy iteration algorithm uses an iterative approach to improve a complete policy, at each step, based on updating a current policy. Arbitrary policy is defined at the beginning as a start point and with each update, all policy improvements are noted and its actions are updated accordingly.

$$\pi(s) = arg\max_{a\epsilon A}(R_s^a + \sum_{s^{'}\epsilon S} P(s,a,s^{'})\gamma V(s^{'})) \tag{2.10}$$

For even a simple problems, this is a very expensive approach where every single action requires a full policy to be updated and saved for the next action.

Value Iteration Addressing a policy iteration expensive update step, Value Iteration algorithm updates state-value function of the current state based on the next state improvement only. This sounds familiar? Our first solution for an MDP where Bellman Equation was introduced, in the previous action, was Value Iteration algorithm. Now, we will add a formal representation of its update rule:

$$V_{i+1}(s) = \max_{a\epsilon A}(R_s^a + \sum_{s^{'}\epsilon S} P(s,a,s^{'})\gamma V_i(s^{'})) \tag{2.11}$$

Value Iteration algorithm needs a termination criterion to stop its further update of the state-value function. Usually, it is the improvement rate or difference between current and updated value function that can be used for the decision to stop an algorithm. For example, if the difference between two consecutive value functions is not greater than ϵ than algorithm stops.

$$\max_{s \epsilon S} |V_i(s) - V_{i+1}(s)| \leq \epsilon \tag{2.12}$$

ϵ can be chosen to be a small value, and Value Iteration algorithm usually produces desirable results that are near the optimal solution.

Algorithm 1 Value Iteration: Learn Value Function $V : S \to R$

```
Require:
    States $S$
    Actions $A$
    Reward Function: $R() : S \times A \to R$
    Transition Probabilities: $P(s, a, s^{'})$
    Discount Factor: $\gamma \in (0, 1)$
 1: procedure VALUE ITERATION($S, A, R, P, \gamma$)
 2:     V(s) = 0
 3:     repeat
 4:         $\delta = 0$
 5:         for $s \in S$ do
 6:             $v = V(s)$
 7:             $V(s) = \max_{a \in A}(R_s^a + \sum_{s^{'} \in S} P(s, a, s^{'})\gamma V_i(s^{'}))$
 8:             $\delta = \max(\delta, \mid v - V(s) \mid)$
 9:         end for
10:     until converged or $\delta \leq \epsilon$
11:     for $s \in S$ do
12:         $\pi(s) = arg\max_{a \in A}(R_s^a + \sum_{s^{'} \epsilon S} P(s, a, s^{'})\gamma V(s^{'}))$
13:     end for
14:     return V, $\pi$
15: end procedure
```

Learning-based Methods

Planning-based methods require a full model of an environment to calculate a policy for an MDP instance. On the other hand, we have a set of algorithms that learn a model or its estimate based on the interaction with the environment. An agent is allowed, for the certain number of steps, to interact with the environment and based on its interaction to decide the best action to take in a particular state. Based on the number of steps taken for each estimation, based on different preferences for an agent in choosing the best action, we have different learning-based algorithms for solving an MDP instance. The common dilemma for all learning-based RL algorithms is in the relation between agent's usage of current knowledge and exploration of a new knowledge of an environment because both cannot be done at the same time. This central dilemma is known as Exploration vs. Exploitation dilemma. If the agent in his common learning path choose always the maximum expected action-value assuming that it will lead to the optimal value in the long term. This approach of only exploitation is called a greedy approach and, usually, it doesn't yield optimal policy. In the long term, every time exploiting the knowledge without exploration of possible higher values hidden behind current low values will not guarantee a good or near-optimal solution. A better approach to learning algorithms is in a choosing appropriate ratio between the time of exploitation and exploration, and different methods exist to address this crucial part of successful RL learning algorithm. ϵ-greedy and *Softmax* are one of the most famous selection approaches for exploitation and exploration ratio.

ϵ-greedy As already mentioned that allowing an agent to only exploit what already it knows from the environment by taking only the highest values will not lead to the optimal solution and this all-time greedy approach has fast convergence rates with poor overall performance. ϵ-greedy approach alters all-time greedy approach by allowing an agent to acts greedily most of the time, but on occasion to choose randomly between non-greedy actions. This occasion is defined by an ϵ. More precisely, an agent acts greedily with the probability 1 - ϵ where ϵ is defined in the range ($0 < \epsilon < 1$), while acts non-greedily with the probability of ϵ. When the agent acts non-greedily, it chooses randomly from the set of all actions (including the greedy one).

Softmax ϵ-greedy method approaches non-greedily actions by randomly choosing between all actions not looking in their estimations. If the environment generates estimations of the wide range of values, this may lead to sub-optimal policies far from the optimal. *Softmax* method is addressing this possible ϵ-greedy weakness by weighting the non-greedy actions based on their current estimations of action-value functions (Q-values). Furthermore, the weighting of non-greedy actions can be done on many ways, by arranging them based on some distribution (for example Gibbs distribution), by sorting them simply by their values assigning to each probability based on its Q-value estimation, by employing search algorithms [19], etc. Selecting appropriate method (in literature called action-selection methods) for Exploitation and Exploration is a crucial in RL learning-based methods (it is obvious that RL planning-based methods contain a model and no estimation is required, hence no exploration) and not rarely suitable method depends on the nature of the problem and no best action-selection method exists until it meets certain problem and proves its efficiency.

Monte Carlo

The Monte Carlo (MC) learning-based approach on RL instance is a group of related algorithms that are inspired by Monte Carlo randomized concepts. In general, MC approach divides agent's learning experience into episodes, and, at the end of each episode, averages gained and employs different techniques for updating current values (state-value or action-value (Q) functions) gained within the episode. By changing the time of MC step (averaging) we create an additional category of MC algorithms. For example, *every-visit MC* averages the discounted rewards across all steps within the episode. On the other hand, *first-visit MC* averages only discounted rewards of the first steps within the episode. The famous MC inspired algorithms for solving an MDP are MC Policy Evaluation and MC Control.

Temporal Difference

Combining the benefits of both dynamic programming and Monte Carlo approaches, the temporal difference (TD) algorithm creates another family of RL learning-based method for solving MDP problems. Taking the advantage of dynamic programming, its update function is based on the all next states while allowing the agent to learn from immediate interaction with the environment. The TD algorithm generates improved algorithms that were successfully applied in many real-world problems [17]. Based on the update phase where the current MDP policy is refined, different TD algorithms are possible. In this thesis, our attention is restricted to TD algorithms where the update takes place after each step taken.

T(0) is a TD-based algorithm where the MC method is combined with policy evaluation. *T(0)* updates the value function based on successor states while employing gains (rewards) with a learning rate using the following update rule:

$$V(s_t) := V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.13}$$

where α is a learning rate with values between 0 and 1. Faster learning is possible using values of α close to 1 at the expense of higher residual error. On the other hand, α, with values close to 0, allows reaching lower residual errors but at the cost of longer training times. In most practical implementations, α is decreased gradually throughout the learning phase as a trade-off between learning speed and accuracy.

Q-Learning Unlike *TD(0)* that incorporates the state-value function for one-step update, the Q-learning algorithm updates the maximum expected action-values (called *q-values*) of the next state. The Q-learning updates its q-values using:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a' \epsilon A} Q(s_{t+1}, a') - Q(s_t, a_t) \right] \tag{2.14}$$

Pseudo-code for the q-learning is listed in Algorithm 2.

Algorithm 2 Q-Learning Algorithm: Learn Q Function $Q : S \rightarrow \{R, A\}$

```
Require:
    States $S$
    Actions $A$
    Reward Function: $R : S \times A \rightarrow R$
    Discount Factor: $\gamma \in (0, 1)$
    Learning Rate: $\alpha \in (0, 1)$
    Transition Probabilities learned by agent
 1: procedure Q-LEARNING($S, A, R, \gamma, \alpha$)
 2:     Q(s) = 0 (initialization)
 3:     repeatafter each episode
 4:         for each step in episode do
 5:             choose action $a$
 6:             execute action $a$, observe interaction, reward $r$, $s^{'}$
 7:             $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a^{'} \in A} Q(s^{'}, a^{'}) - Q(s, a)]$
 8:             $s = s^{'}$
 9:         end for
10:     until Q converged or certain number of iterations achieved
11:     for $s \in S$ do
12:         $\pi(s) = arg \max_{a \in A} Q(s, a)$
13:     end for
14:     return V, $\pi$
15: end procedure
```

Sarsa The state-actionrewardstateaction algorithm (aka Sarsa) is another TD one-step algorithm created from the modified version of the q-learning algorithm using an approach of *TD(0)*. Sarsa updates the a-values as a q-learning but does not take the maximum expected value of the improvement of next state, it rather directly updates on any improvement on next state, as it is the case in *TD(0)*. The update of the q-values is given by:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.15}$$

Algorithm 3 Sarsa: Learn Q Function $Q : S \rightarrow R, A$

```
Require:
    States $S$
    Actions $A$
    Reward Function: $R() : S \times A \rightarrow R$
    Discount Factor: $\gamma \in (0, 1)$
    Learning Rate: $\alpha \in [0, 1]$
    Transition Probabilities learned by agent
 1: procedure Q-LEARNING($S, A, R, \gamma, \alpha$)
 2:     Q(s) = 0 (initialization of Q arbitrarily)
 3:     repeatafter each episode
 4:         for each step in episode do
    choose action $a$
    execute action $a$, observe interaction, reward $r$, $s^{'}$
    $Q(s, a) = Q(s, a) + \alpha[r + (s^{'}, a^{'}) - Q(s, a)]$
    $s = s^{'}$
 5:         end for
 6:     until Q converged or certain number of iterations achieved
 7:     for $s \in S$ do
 8:         $\pi(s) = arg\max_{a \in A} Q(s, a)$
 9:     end for
10:     return V, $\pi$
11: end procedure
```

Pseudo-code for the Sarsa is listed in Algorithm 3. It is worth to mention that TD methods are one of the most common due to their simple and yet powerful generation of model-free experience from the interactive actions. From the TD methods, the discovery of Q learning is recognized as the most valuable for success and respect of RL-based methods they enjoy now. There exist other different approaches for solving RL instances (for example, Dyna-Q and Q-Planning), but our main concentration, in this thesis, will be on the algorithms presented till now, particularly the VI, Q-Learning and Sarsa algorithms.

CHAPTER 3

LITERATURE SURVEY

3.1 Refactoring

Refactoring related-topics have taken extensive research effort in the last decade from the different perspectives. For illustration, last three years we found several systematic literature surveys and reviews on refactoring from the different aspects [20, 21, 22, 23, 24, 25, 26].

In general, one of the first literature surveys on refactoring was conducted a more than a decade ago [27]. It was considered one of the main resource for related work of refactoring. Authors perform the state-of-the-art survey of research work done till 2004 and discussed their impact based on different criteria. For instance, activities supported, techniques and patterns used in these activities, parts of software being refactored, etc. The survey was also followed by open discussion and advice, to take into consideration, for refactoring tools and the process itself. Rather more general with lack of more detailed concentration of each work presented found in standard systematic literature reviews (the term of SLR in software engineering came long after (2009) by work of Kitchenham et al. [28]), this survey covered the wide range of topics related with refactoring.

With motivation that review of research about code smells and refactoring will help in better understanding of effects of refactoring on software quality, Wangberg's [29] revised published sources till 2009 with the wide range of topics discussed. The main findings were related to the facts that very small portion of the work included empirical studies(24 %). Most of the work (22 out 28) was concern about detection of the code smells and reviewers admitted a significant increase in the number of publication on code smells and refactoring since 2005 [29].

Published sources till 2009 were again analyzed by Zhang et al. [30] with the aim to examine empirically the usage of code smells and refactoring in improving software quality. They found that Duplicated Code was the most analyzed bad smell among researchers and that most of the contributions of research society were in developing techniques to detect bad smells and only a few authors were concern about quantifying its relation with software quality on empirical base. Although in overall this review suggests that there is little evidence currently available to justify using code-smells [30], their detailed discussion tries to confirm that software engineers may believe that bad smells and its philosophy behind software quality improvements are common-sense and do not require any empirical evidence.

Al Dallal [21] reviewed object-oriented code-related refactoring opportunities in published sources ignoring all 'non-code' refactoring (for example, model-based or refactoring to service or aspect-oriented code refactoring). Furthermore, only studies reporting empirical evaluations were considered. The author found that Extract Class, Move Method, and Extract method were refactoring tasks used in most approaches. According to this SLR, around 40 % of the work reviewed did not define their refactorings. Moreover, this SLR presents empirical proof that most common practice in evaluation of refactoring success was intuition or it was closely related to human involvement (with 48.9% of all evaluation approaches). Although published in 2015, this SLR includes published sources till 2013 and we found it failing to recognize and classify AI-based approaches as a separate and important shift in empirical refactoring approach.

While Al Dallal's SLR concentrate on code-based refactoring only, model-based (UML) refactoring was the main interest of SLR conducted by Misbhauddin and Alshayeb [22]. SLR analyzed 94s publication sources with different criteria and its effect on model quality.

Recently, Rasool et al. [20] conducted SLR on code-smells mining techniques on research published till March 2015. Results include that Feature Envy, Data Class, Large Class, Long Method and Long Parameter List code smells acquired the maximum attention of researchers. According to authors, lack of formal definitions of code-smells and lack of standardized benchmark system for evaluation of detection/refactoring techniques are gaps that should be addressed by code-smells' research community. Also, most of the tools (92 -95 %) for code-smells detection/refactoring are for Java programming language-based code and most of the datasets used for evaluation are open-sourced. Authors recognized AI-based refactoring works (namely, search-based and Machine Learning-based) listing its resourced and commenting on its dependency on search-space and training data. Additionally, for search-based techniques, authors mentioned additional effort in tuning parameters and possible drawback with the generality of the approach.

As we can see, refactoring and concept of bad-smells, in general, are part of active research and in the next section, we present, in more detail, what has been done on its prioritization.

3.2 Prioritization of Refactoring Tasks

Liu et al [31] addressed the problem of prioritizing refactoring based on conflicts. Different conflicts among refactorings are detected and prioritizing model is proposed. The main objective of the proposed model is to improve software quality while resolving the problem of scheduling conflicted refactorings.

Work in [32] is one of the first attempts to formally (mathematically) define prioritizing refactoring problem as a Multiobjective Entity Refactoring Set Selection Problem (MOERSSP). While the formulation includes in its name multiobjective, the two objective functions were combined into single optimization problem by aggregation with weighted coefficients. As one of the early attempts to use evolutionary algorithms into prioritizing the set of refactorings, the work lacks automation and it is manually tuned based on 'trial-and-error' fashion. Also, empirical evaluation is done on a single project without further evaluation.

Authors in [33] manually analyzed (in a pairwise manner) different smells and their priority of refactoring. Results of pair-wise analysis between smells were represented in graph-manner with priority denoted by an arrow between each smell. This graph is later used as a base for topological sorting to find the optimal refactoring sequence of bad smells. This work takes into account only developers' time effort as an objective to minimize. It is a semi-automated method with high dependence on human-interaction for analysis and evaluation. Refactoring tasks are performed on source-code and 'quantity' of the code (code smell) was one of the main reason for prioritizing its severity ('longer' code smell more severe).

Meananeatra [34] focused on prioritizing refactoring with respect to maintainability. For authors, maintainability is achieved by its four criteria (analyzability, changeability, stability, testability). The prototype developed requires from developer manual preferences of these four criteria. After developers' preferences, the second optimization objective is to reduce the number of smells that will achieve developers goal chosen. The main algorithm for presenting refactoring process is graph-based where root node represents the source code and its children are the different output of refactoring tasks. Using predefined criteria and preferable outcome, graph transformations are used to find the optimal sequence of refactorings. As a result, formulation of final selection can be seen as maximizing the number of removed bad smells and maintainability while minimizing the size of refactorings in refactoring sequence and the number of program elements to modify. Authors proposed the solution but no evaluation of real projects is found. Since, as authors stated (2012), it was prototype and development was in progress, till now (2016) no trace for its application and evaluation.

Mining documented defect examples, authors in [35] created detection rules based on defects regularities. Genetic Programming (GP) is used to automatically extract these rules. After detection, these rules were used to denote the minimized number of detected defects that maximize code quality in refactoring process. Maximizing code quality, as a first objective, is combined with minimizing the effort needed to apply refactoring operations [35] creating multi-objective approach. Non-dominated Sorting Genetic Algorithm (NSGA-II) [36] was employed to solve a multi-objective problem with finding a set of solutions (Pareto Front).

Mkaouer et al. [37] designed a tool (DINAR) that requires developers interaction to dynamically prioritize refactoring tasks. NSGA-II optimization algorithm is used to optimize three objectives: improve software quality, reduce the number of refactorings and increase semantic coherence. After developers' feedback, local search is performed to update the refactoring recommendations. The novelty of this approach, next to proposed tool, is an analysis and exploration of Pareto Front in the interactive manner that includes direct developers' feedback. This work is built on previous work of the same group where they used defect repositories to ”mine regularities about defect manifestations that can be translated into detection rules" [38].

Moreover, the same group of authors [39] taken in account dynamic environment of software development to model refactoring based on the uncertainties related to the class importance and code smell severity. Robust optimization [40] is used to design the code refactoring problem and NSGA-II multiple-objective approach algorithm is employed to find trade-offs between software quality (minimizing the number of code smells) and the robustness of the refactoring solutions in relation to uncertainty in the severity level of the code smells and in the importance of the classes that contain the code smells [39]. Results obtained were first compared to Random Search as a basic comparison. After, NSGA-II is compared with MOPSO(multi-objective PSO) and Mono-Objective Evolutionary Algorithm (Genetic Algorithm). In most open-source projects evaluated, proposed approach outperforms others approaches.

Ali Ouni, in his Ph.D. thesis [41], collected what he has done, being part of refactoring research group, for code-smells detection and software refactoring. For software refactoring, next to separately published work [35, 3] author came with two other approaches for prioritization of refactoring. The first approach has four objectives: 1)to fix the code smells, 2)reduce the number of modifications/adaptations needed to apply refactorings, 3) preserve the semantic coherence of the refactored program, and 4) maintain the consistency with development/maintenance history [41]. NSGA-II heuristic is used to explore search space and results, with metrics defined for each objective, were reported with the comparison to other heuristic algorithms (MOGA, Random Search) and JDeodorant. This study rises, among others, an importance of semantic coherence preservation and with, on average, 80% of semantically feasible achievement of applied method, this is a significant improvement in comparison with other approaches evaluated. NSGA-II outperforms significantly MOGA, random-search, and mono-objective algorithm in terms of code-smells correction ratio (CCR), semantics preservation (RP), and code changes reduction [41]. In the second approach, multi-objective recommendation refactoring system is developed with emphasizing on introducing design patterns and fixing anti-patterns. Tool MORE ((Multi-Objective REfactoring) is developed to support proposed approach and empirical study is conducted with a quantitative and qualitative evaluation including developers and common open-source projects. Based on metrics for evaluations (code-smells correction ratio (CCR), the number of newly designed pattern instances (NP), quality gain (QG)) and authors defined metric (refactoring meaningfulness(RM)) in comparison with the state-of-the-art approaches, authors denote that their proposed method outperforms others work.

Malhotra et al. [42] prioritize four types of code-smells Feature Envy, Long Method, God Class and Type Checking on the class level based on time constraints for their refactoring. Chidamber and Kamerer metric suite is used with different combination to identify highly affected classes for immediate refactoring. This approach is only based on software metrics (C&K) for determining software quality and its work is based on source-code refactoring. Recently published work [3] on prioritizing emphasizes the riskiest code-smells to be refactored first. They used, for the first time in software refactoring problem, Chemical Reaction Optimization (CRO) scheme to find the best combination that outputs optimal refactoring sequence improving overall quality with a minimized number of detected code-smells. Authors used open-source projects for evaluation (with well-known smells and required refactorings). Common datasets allowed comparison with other metaheuristic approaches and authors denoted uniqueness of CRO and its promising results. Some good properties of CRO revealed in their optimization problems (such as its advantage on both, GA and SA algorithm) were shown useful in this work to tackle refactoring problem. One of the main goals of the work was to ensure that the riskiest code-smells are fixed first [3]. Authors stated that 'the riskiest' code-smells, in terms of developers' preferences, are concerned. But, we also find the usage of 'common-sense' in determining the riskiest code-smells stating some well-known preferences between code-smells (for instance, taking for a grant that blob class has a higher risk than functional decomposition). In comparison with other approaches without prioritization(in terms of ratios of importance, risk, severity, and precision) authors reported significant outperform. In addition, comparison of CRO with other metaheuristics (GA, SA, and PSO) is performed and authors report that CRO is a much better solution in terms of refactoring ratios (importance, risk, severity, and precision) on average of all experiments. The main drawback of this approach might lay down in its lack of generalization and its prior requirement of the history of code changes applied to the system during its life-cycle [3]. Moreover, all evaluations were done on the well-known open-source projects with well-studied code-smells and their effects. While it is an advantage for studying improvement of approach and its replicability; for generalization, shifts from standard datasets may reveal other aspects of approach applied. Summary of related work with prioritization of refactoring is presented in Table 3.1.

3.3 Refactoring and Machine Learning Techniques

Authors in [43] applied LSSVM with some preprocessing (Wilcoxon test, PCA) and smoothing techniques (SMOTE) to predict refactoring classes based on software metrics using datasets of 7 open source projects [44]. From 120 software metrics, 31 of them were chosen as a more relevant discriminant of refactored and non-refactored classes based on mean value and statistical test. Refactored and non-refactored classes are from different consecutive versions of the same system. PCA was used to reduce 31 software metrics to 6 PCs and imbalanced data of refactored and non-refactored classes were addressed by SMOTE technique. LSSVM binary classifiers, with different kernels, were used and high results were reported in prediction/classification part ((the highest of 0.96 AUC). The interesting part of our topic is that cohesion metric (LCOM) did not pass the first step of this framework since they were not chosen as a consideration after the first

Ref.	Algorithm	Level	Dataset	Smell	Objective
(Liu et al. 2007)	**graph-based**	Code-based	**private:** PMMT	NN	**Maximize** Quality
(Meananeatra 2012)	**graph-based**	Code-based	NN	Long Method	**Maximize** Removed Smells Maintainability **Minimize** Refactoring Sequence Elements to Modify
(Liu et al. 2012)	**graph-based**	Code-based	**public:** Java Source Metrics Thout Reader	Duplicated Code Long Method Large Class Long Parameter List Feature Envy Primitive Obsession Useless Field Useless Method Useless Class	**Minimize** Time Effort
(Malhotra et al. 2015)	**meric-based**	Code-based	**public:** OrDrumbox	Blob Feature Envy Long Method Type Checking	**Maximize** Quality
(Chisalita 2009)	**metaheuristic-based:** GA	Code-based	**public:** Simulation of LAN	NN	**Minimize** Cost
(Kessentini et al. 2011)	**metaheuristic-based:** GP,GA	Code-based	**public:** Xereces-J GhanttProject Quick UML ArgoUML	Blob Spaghetti Code Functional Composition	**Minimize** Detected Defects
(Ouni et al. 2013)	**metaheuristic-based:** GP, NSGA-II	Code-based	**public:** Xereces-J GhanttProject Quick UML AZUREUS LOG4J ArgoUML	Blob Spaghetti Code Functional Composition	**Maximize** Quality Maintability **Minimize** Effort
(Mkaouer et al. 2014a)	**metaheuristic-based:** Local Search, NSGA-II	Code-based	**public:** Xerces-J JFreeChart GanttProject JHotDraw **private:** JDI-Ford	Blob Spaghetti Code Functional Composition	**Maximize** Quality Semantic Coherence **Minimize** Refactorings
(Mkaouer et al. 2014b)	**metaheuristic-based:** NSGA-II, MOPSO, GA	Code-based	**public:** Xereces-J JFreeChart GhanttProject JHotDraw ApacheAnt Rhino	Blob Spaghetti Code Functional Composition Data Class	**Maximize** Quality and Rrobustness
(Ouni et al. 2015)	**metaheuristic-based:** CRO, GS, SA, PSO	Code-based	**public:** Xereces-J JFreeChart GhanttProject JHotDraw ArtOfIllusion	Blob Spaghetti Code Functional Composition Data Class Schizophrenic Class Shotgun Surgery Feature Envy	**Maximize** Number of Prioritized (Riskiest) Corrected Smells
(Ouni 2015)	**metaheuristic-based:** NSGA-II, MOGA, Random Search, GA	Code-based	**public:** Xerces-J JFreeChart GanttProject JHotDraw ApacheAnt Rhino	Blob Spaghetti Code Functional Composition Data Class Shotgun Surgery Feature Envy	**Maximize:** Fixed Smells Semantic Coherence Consistency with Development/Maintenance History **Minimize** Modifications/Adaptations
(Ouni 2015)	**metaheuristic-based:** NSGA-II, MOGA, Random Search	Code-based	**public:** Xerces-J GanttProject JHotDraw ApacheAnt	Blob Spaghetti Code Data Class Feature Envy	**Maximize:** Introducing Design Patterns Fixed Smells Design Quality.

Table 3.1: Refactoring Prioritization Related Work.

statistical test. On the other hand, coupling metrics (RFC, for example) were one of the 31 metrics considered in feature selection and PCA feature extraction. This is reasonable in refactoring, since refactoring tasks more often address highly coupled classes and any approach to reorganize and "simplify" the system by refactoring affects more coupling than cohesion, on average. Improvement of coupling metrics (with RFC one of them) in refactored classes is found in the previous studies and might represent a common fact among relation of software metrics and refactoring [43, 45, 46].

3.4 Summary

After literature survey, we can see that no reinforcement learning-based method is proposed in the literature to address refactoring automation problem. Moreover, a majority of proposed techniques are simulation-based, transferring source-code to the meta-representation and then applying automation techniques on this meta-representation aiming to reach refactoring sequence. At the end, refactoring sequence is applied on the source-code. While this might be an efficient approach, transferring source-code to meta-model might also create more noise and affect the accuracy of proposed techniques. Furthermore, with meta-representations, we lose direct relation with the source-code, where, in reality, refactoring happen. Reinforcement learning methods might fill this gap with direct interaction with the source-code learning how to optimize refactoring sequence.

CHAPTER 4

PROPOSED METHOD

For a better understanding of the proposed method, we begin with a simple example of Reinforcement Learning application. This example might serve as a motivation behind the proposed method. For example, the proposed method is presented with its several phases discussed with more details.

4.1 Reinforcement Learning: A Simple Example

Before we begin discussion and explanation of the proposed method, let us consider a layout map depicted in Figure 4.1. The map represents a flat where the cleaning robot lives for a while. Flat's world is divided into twelve rooms numbered from 1 to 12. The robot has a daily task to start from the room number 1 and reach the room it needs to clean. Actions are the side directions the robot moves (north, south, east, west) and after each move, the robot is expecting to be in a different room. Each move costs the robot few percentages of battery drain and his owner gave it the instruction to clean the room number 12 (the owner's room) and doing it will give the robot full battery recharge plus additional benefits. Otherwise, if the robot does not reach the room 12 and its battery got drained, its owner will "punish" him by not recharging its battery for 3 days plus the robot will be "sad" because it was late 3 days to clean the owner's room (let us imagine that the robot is conscious). Furthermore, the robot knows that the room 6 is closed and trying to enter it will just a drain his battery without any progress. Moreover, the owner knows that his robot is not 100% reliable and rational, because not always it takes an intentional move. Sometimes the robot displays that it is moving north but it ends moving in other directions.

His owner wants to help it and gave it a sequence of the directions that will lead it to the owner's room. Without much thinking the owner orders the robot to move [east - east - south - south - east] and it will reach the room number 12. After the month of observations, the owner was not very happy with his robot not always cleaning his room, even after he gave it the right sequence of moves. On the other side, the robot was sad that it cannot reach the room number 12 every time it starts with the room number 1. Most of the time the owner will find it in another room.

After a while, the owner realized that battery drains sometimes pushed the robot to stop. Sometimes its unreliable nature prevented it to take the right directions and it took its additional battery drains. The owner was thinking in the direction to calculate the percentage of battery drain for each move but then realized that the battery, by every recharge, requires another recalculation of the drain-per-move variable. Lastly, he realized that he will need a general and easy strategy to recalculate the optimal sequence based on the state of the robot and his battery level on the long-term maximizing its benefits.

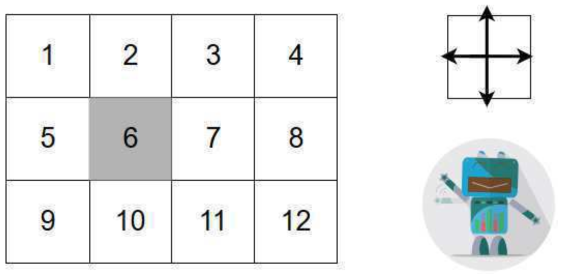


Figure 4.1: Robot World.

What is the best overall policy for the robot? What will maximize overall happiness of its owner?

Luckily, the owner remembered his friend who had courses in AI (a friend owns a degree in computer science). A friend explained to him that once the robot is unreliable, he has to represent its moves with the probabilities of every direction. Observing the robot for the past month gave him insights that 80% of the time the robot is reliable and 20% of the time it acts randomly. Furthermore, a friend told him that battery drains were, for him, clearly negative rewards (or costs) for moves the robot takes. Pairing the probability with the directions, adding a negative reward and room number to it, they created a utility function. His friend also gave him a tip that the maximum he can get from his robot, in the long term, is to maximize this utility function for each room. The direction with the maximum utility function is the best direction for the robot to take in that room. The owner, feeling enlightened with the explanations of his friend, started to calculate these utility functions for every possibility in his twelve rooms map. Starting from the room number 1, to get its maximum utility function, he needs to calculate utility function for every direction and multiple it with the maximum utility function of the next room. It could be the either room number 2 or 5. To easily track his calculations, he organized it in a tree-like structure.

After a while, he gave up realizing that it might take him a lot of time to calculate maximum utility function for each room. For example, to calculate it for the room number 2, he needs also the maximum utility for the room number 3 that needs the maximum utility for the room number 4 and the room number 7, and so on, for every direction (a friend obviously did not want to explain the owner every detail that could lead him to solution easily, but wanted him to understand it gradually with a little effort).

The next meeting with the friend he understood that he had to calculate 16 777 216 utility functions to cover all twelve rooms (without closing the doors of the room number 6). Every room with all directions gave him 4^{12} possibilities. The owner realized that the problem was with the maximum operator and no other way, except for the calculation of all possibilities then taking the maximum, will lead to the optimal solution. Moreover, a friend explained to him that his and similar examples can be represented with a simple, yet powerful framework called Markov Decision Processes (MDP) that can be addressed with many successful algorithms. This framework has rewards, states, actions, probabilities, discount factor, map and an agent that wants to reach some position on the map. The solution is a policy that gives the action for each state agent can enter. The optimal policy is the one that gives the optimal action for each state in the long term. One of the algorithms that can find the optimal solution is to assign for each state some value for a utility function, they discussed previously, and then update it in an iterative manner till no change on these utility functions is observed. A friend gave him a small program that automates calculations for him. Finally, it was not hard for the owner to realize that his robot is an agent, negative rewards are battery drains, states are the owner's rooms and actions are robot's directions. A discount factor was a new concept for him, but soon he recognized that the agent prefers rewards that he get sooner over the rewards he gets latter, and discount factor is there to achieve this preference. The solution, the owner is searching for, was calculated using the friend's small program, to save his time of manual calculations. After a month, the owner was able to place the robot in any room and give it an order to clean any other room with the policy he had already calculated, being satisfied with its maximization of the robot's performance.

This toy example of Reinforcement Learning and sequential decision problems gave us an idea of the MDP application in the real case scenarios.

What if our agent is a developer who is maintaining large project having a lot of refactoring tasks to execute optimizing the source code. Yet, every refactoring task takes developer's time and effort and not always he takes the right task. His stakeholders appreciate his current output and developer wants to maximize it by optimizing the sequence of refactoring tasks to execute. Can we help developers and other software project's stakeholders by transferring refactoring sequencing problem to an MDP instance and solving it efficiently? Can we automate refactoring sequence process by having its MDP instance being solved by optimizing a certain part of software quality measures? Can we properly automate refactoring sequence process minimizing developers' effort reducing the number of refactoring task manually executed?

Searching for an answer to these and other questions is the main motivation behind the thesis. In the next sections we discuss proposed method, formally define refactoring MDP instance and algorithms to solve it.

4.2 Proposed Method

In this section, we propose an automated approach for optimizing coupling and cohesion values of class refactoring sequence. The method is based on MDP and reinforcement learning. The method consists of eight phases depicted in Figure 4.2. These phases are summarized in the next paragraphs and next sections come with more details of each phase:

- **Phase 1**: Input is the class with its metric values.
- **Phase 2**: Metric values (two metrics in our case) of the class are used to create a 2-dimensional world (map). This world has an entrance in its default class values and the goal on the diagonal end of the map.
- **Phase 3**: Rewards, discount factor, and terminal function are defined.

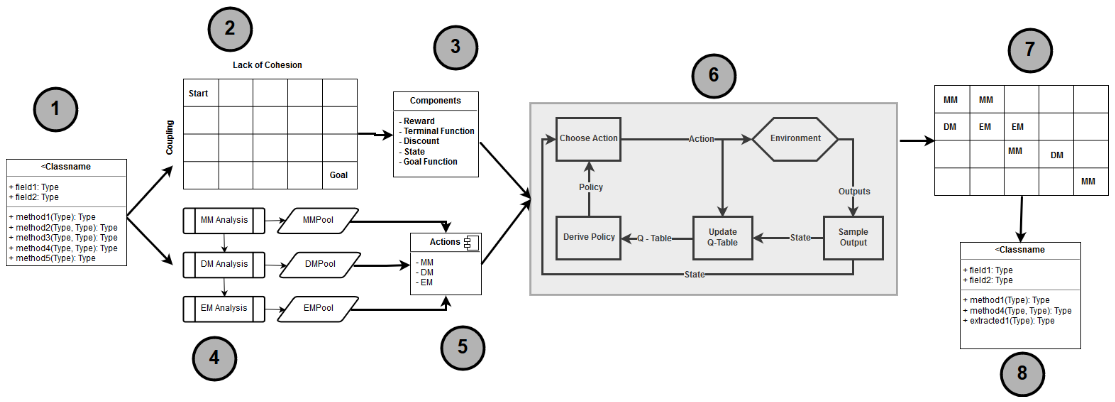


Figure 4.2: Proposed Method.

- **Phase 4**: Input (class) is analyzed for possible action generations. In our case, we search for all correct MM, DM and EM tasks.

- **Phase 5**: All feasible tasks generated from the Phase 4 are grouped and presented as an "actions" in MDP instance. Phase 5, with the phases 2 and 3 together, represents an MDP instance definition of the refactoring problem.

- **Phase 6**: With the previous phase, MDP definition of the refactoring sequencing problem is completed. This phase employs different algorithms to solve an MDP and find the optimal policy.

- **Phase 7**: The optimal policy is found as a result of the previous phase.

- **Phase 8**: Final output is the optimized class with tasks applied according to the optimal policy.

Next sections address these phases with more details.

4.3 Phase 1 - Input

The proposed method has a goal to optimize coupling and cohesion values on a class level. As an input, we have a Java class with its source code. Values for coupling and cohesion are calculated as part of input and next phases are employing these values for the further processing. It is worth to mention that the proposed method is general and that optimization goal depends on software quality measures calculated in this phase. Our approach exploits coupling and cohesion values, if other quality measures were used, the objective optimization differs.

4.4 Phase 2 - Environment representation

S - state function (environment of an MDP) in the refactoring is defined as a 2-dimensional world, where first-dimension represents a class lack of cohesion value and its second-dimension class coupling value. A specific value of coupling and cohesion represents the detailed state, position, on this world/map. As one of our objectives of the refactoring automation is to find the "optimal" ratio between coupling and cohesion, this state space representation is reasonable and suitable. Let us consider it more concretely, in Figure 4.3, we have a 2d map, with coupling and cohesion dimensions. If we know that many software practitioners believe that the class has to be loosely coupled and highly cohesive, in the most cases the goal of the class is to reduce its coupling while improving its cohesion. Moreover, having lack of cohesion value, the goal is to reduce both quality measures. This rationalization leads that the optimal places on the map could be on the diagonal line to the opposite end of the map. The diagonal line gives the shortest distance to the opposite end, means, the less number of refactoring tasks, while achieving the highest reduction rate for both quality measures. The states far from the diagonal line will reduce one of the two metrics while increasing or unchanging the other. This is the reason why the goal state is on the opposite side of the start value. motivating the agent to strive for that location with the least number of actions taken. Moreover, defining the goal state in this way aids to generalize representation of the environment for all classes in the project.

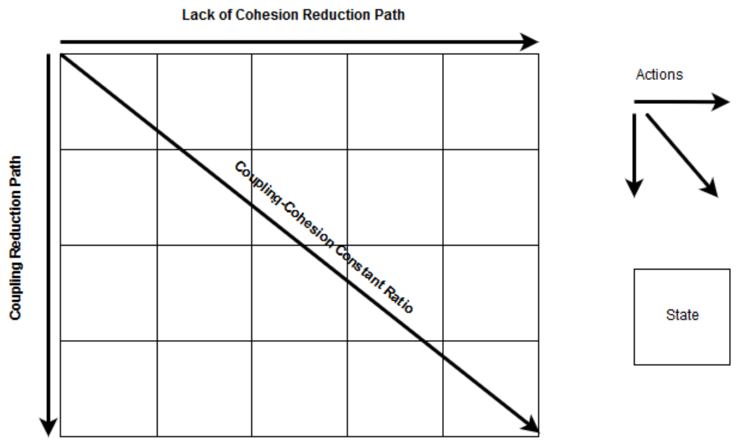

Lack of Cohesion Reduction Path
Coupling Reduction Path
Coupling-Cohesion Constant Ratio
Actions
State

Figure 4.3: Refactoring MDP Environment.

4.5 Phase 3 - MDP Components

After the world definition, the rest parts of the MDP environment are defined in this phase (except actions and its stochastic nature that are defined in the next phases).

$R()$ - reward function, as an immediate reward for the execution of refactoring task, was presented as a cost "that is paid" to execute each refactoring task. Every refactoring task requires an effort from the developer and it costs his time and his morale. The reward for each refactoring task has to try to manifest instant charge that developer pays while executing a certain refactoring task without counting its overall satisfaction of the refactoring process (that is addressed later by value and Q-value functions). So, any refactoring task is a negative experience and cost for the developer.

γ - discount factor, in a simple word, current refactoring task execution reward is a more valued than the same reward after several or more refactoring tasks. Naturally, people tend to value more what they get today then what they would/will get after a month even if the numbered value is the same.

Terminal Function - refactoring sequence stops when no change in the improvement of the utility function is observed for several consecutive actions.

4.6 Phase 4 - Actions Extraction

Only actions, in refactoring, that allow moving to the different states are refactoring tasks (in our case: Move Method (MM), Extract Method (EM) and Delete Method (DM)). Actions for the MDP environment are generated in the two phases. Phase 4 analyzes a class for all possible refactoring tasks and Phase 5 integrates actions found. In our case, we search for MM, DM and EM tasks. For each kind of task, the analysis is performed to eliminate all tasks that could produce any compilation or semantic error. Each analysis generates a pool of tasks that are eligible for actions in MDP environment. Next to actions, transition probabilities have to be assign. The state transition probability ($P()$) function requires some prior knowledge about refactoring tasks and the system since it plays one of the main roles in the success of the automation process. We assume based on a manual inspection, that MM and DM refactoring tasks have a much higher probabilities to occur than EM refactoring task. Moreover, MM probability should be twice higher than its DM counterpart. This knowledge has to be incorporated in transition probabilities. It would be a common to define the highest probability for the MM, the second highest for the DM and the rest for the EM. However, we decided that all refactoring tasks are equiprobable. This might be surprising after all knowledge gathered from the manual inspection, but we have to take into consideration a rigorous analysis prior automated refactoring, especially for the EM analysis part, where any error produced could lead to the discarding of the complete EMPool. After manual refactoring and analysis part inquiry, equiprobable refactoring tasks most probably, we believe, will generate the same effects of the manual refactoring.

Actions are also defined as stochastic was agent 80% of time executes intended actions while 20% of time acts randomly, choosing from the rest of actions.

Note: The number of actions is an infinite, but due to the nature of the problem, all refactoring actions are required to produce "compile-error-free" code. Therefore, the number of actions will be reduced as shown in Section 6.5.

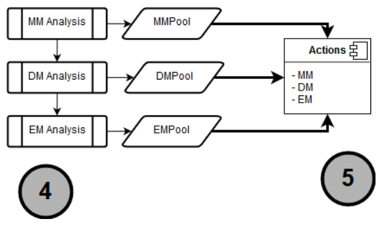


Figure 4.4: Actions Extraction and Integration.

4.7 Phase 5 - Actions Integration

Phase 4 analyzed all feasible refactoring tasks. Every kind of task produced the pool. This phase integrates these pools into actions for the MDP environment. With this phase, the definition of the MDP instance is complete and with the next phase we start to solve this MDP instance.

4.8 Phase 6 - Solving the MDP

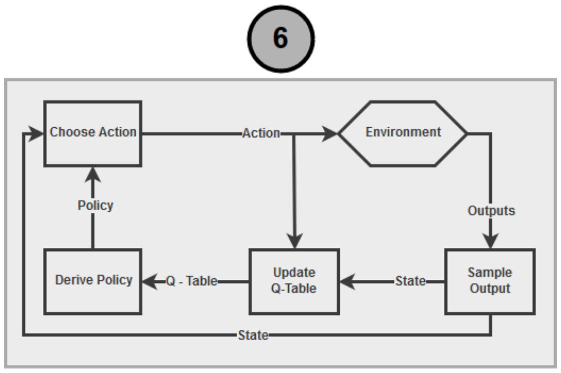


Figure 4.5: Solving the MDP Instance.

As explained in the background chapter of this thesis, an MDP instance is the main input for the RL-based machine learning that can be solved using different approaches. Two of them are followed (planning and learning-based approaches) in this thesis with overall three algorithms employed (VI, Q-learning, and Sarsa). In general, all these algorithms are iterative in nature, calculating and improving utility function from the agent-environment interaction. This general iterative approach is depicted in Figure 4.5. The agent chooses the action, interacts with the environment and outputs its observation with some kind of update on utility function and policy. The way the utility function is calculated, presented and updated generates different algorithms for finding the solution. While the theoretical background is discussed in Chapter 2, next sections extend its discussion with algorithms used to solve refactoring-based MDP instance.

Value Iteration

Value Iteration (VI) is the only planning-based RL algorithm used for solving refactoring MDP instance in this thesis. Its mandatory input requires a complete model of the refactoring environment (transition probabilities have to be predefined with actions). VI updates value function (utility function) of the current state based on improvement on the next state only. For refactoring MDP instance, VI algorithm updated utility function after each change in coupling and cohesion values with refactoring task (action) executed. Pseudocode is depicted in Algorithm 1.

Q Learning

The first learning-based RL algorithm applied on the refactoring MDP instance is a Q-learning. Q-learning agent started with the arbitrary Q-values equal to zero and then, in an iterative manner, improved Q-value for each state by executing and sequencing different tasks.

Sarsa

The second learning-based RL algorithm utilized to solve a refactoring MDP instance is Sarsa. The modified version of Q-learning, it directly improves Q-value of the current state based on improved in the next state.

4.9 Phase 7 - Policy Output

The results generated from the previous phase are the policy and maximized utility function related to each state. The policy contains suitable refactoring task for coupling/cohesion state. With the policy, desired refactoring sequence is found by extracting refactorings from the policy and presenting reached coupling/cohesion values (state). In the best case scenario, the last state with refactoring task mapped is the optimal solution for the class in terms of coupling and cohesion. Furthermore, the policy presentation of refactoring states mapped to the state of coupling and cohesion allows the developer to choose different scenarios for refactoring sequence. If a number of refactoring tasks is limited, a developer can, for example, consult the policy for an only limited number of refactoring task that will give him exact tasks to execute.

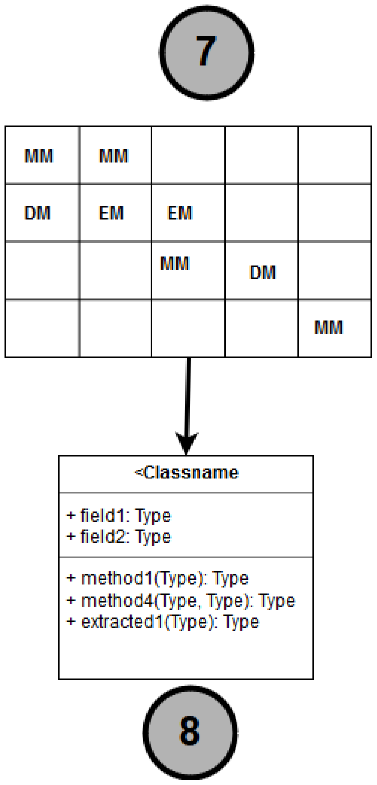


Figure 4.6: Policy Output.

4.10 Phase 8 - Class Output

After the policy generation, in the best case scenario, the Java class is considered optimized. The automated approach takes the output of class with the last state reached in the previous phase. That means that all refactoring tasks presented in the policy are applied to the class.

CHAPTER 5

TOOL SUPPORT

There are various tools proposed for bad-smells detection and refactoring in software engineering community. Some of them are open-source and free, while some other are commercial. To validate our reinforcement learning approach, we have used different tools on different stages of our experimental evaluation. For bad-smell detection, inFusion was our choice since it is based on software metrics and it is highly stable. Its base on software metrics will give us more confident in validation of our RL-based approach. After detection of bad-smells, ckjm-tool was used for coupling and cohesion metrics calculation. Eclipse platform (JDT, PDE, and LTK) was the base for emerging ckjm with our new tool developed and, in the next sections, we explain these and other tools with more details.

5.1 Detection and Metric Tools

Various bad smell detection tools are proposed in the literature. Some of them are proprietary while some are still research prototypes. These tools detect bad smells based on metrics, analysis of the program, machine learning approaches or specially designed specification. In the next paragraphs, we discuss some of these tools and their range of bad smells detection. Moreover, a metric tool used in this thesis is also presented and discussed.

5.1.1 JDeodorant

JDeodorant [47] is bad smells detection tool developed as an Eclipse-plugin. Next to detection, it offers automated refactoring tasks to perform with each of bad smells detected. Next to admirable refactoring task suggestion, a developer has to be careful with its execution. It works with Java projects and offer extension and changed and presented threshold values.

5.1.2 PMD

PMD [48] is another detection tool that can detect, next to Duplicate Code, bad smells related to a large amount of code in a single entity(Large Class, Method, Parameter List). It allows an extension to the existed rules and developer definition of different threshold values for bad smells detection.

5.1.3 CheckStyle

CheckStyle tool [49] offers static analysis of code and, having predefined code rules, tries to convince a developer to develop unified coding style and conventions. Built as a jar file, can be executed standalone inside Java VM, as an Ant task, and can be part of IDE (Eclipse). It is mainly working on Java projects while extensions for other languages do exist (PHP, for example). Related to bad smells, it detects Duplicate Code, Large Class, Large Method and Large Parameter List. CheckStyle is extensible with customized rules possible defined by the developer. While concentrating on the syntax of code and possible smells, it doesn't prioritize nor suggest refactoring tasks to improve addressed bad smells.

5.1.4 inFusion

inFusion was a proprietary software built by Intooitus. The tool was a result of improvements in the previous tool (iPlasma) and well-studied research on software metrics, bad smells and object-oriented programming paradigm also published in the book called *Object-Oriented Metrics in Practice* [50]. According to one of its creator, with a plan to "put your development team in control of the quality of your project's architecture and design. It is designed to make quality assurance of multi-million LOC projects practical and effective" [51], inFusion wanted to tackle development companies and help them address their software complexity, maintainability, and other quality measures by visualizing relation between classes with an identification of bad smells and possible programming drawbacks. inFusion detects and prioritizes, with severity score, bad smells and classes affected based on software metrics and suggest possible solutions with explanation what might cause founded software drawback. Its structural, code and polymetric views address quality of software dimensions in various ways and provide insights about each dimension. With tutorial-like guides, it helps beginners to overcome most of ”unknown” terms and its behinds. The same team offered the lighter version of inFusion, called inCode [52], for smaller teams and developers, as a fully integrated into Eclipse [51]. inCode followed a similar approach as inFusion with support for the smaller project (limited to 100 000 lines of code). inFusion tool was able to analyze Java, C and C++ projects. Unfortunately, company Intooitus doesn't exist anymore and their tools are no more available.

Next to presented bad smell detection tools, there are other various tools proposed by researches and interested reader my check tools: JSpIRIT [53], DECOR [10], CodeNose [54], Stench Blossom [55], jCOSMO [56], SMURF [57], JCodeCanine [58], JCodeOdor and DFMC4J [59], SCOOP [60], BSDT Eclipse Plugin [61], CodeVizard [62], EvoOnt [63], Anti-Pattern Scanner [64].

5.1.5 ckjm

Ckjm tool [65] calculates Chidamber and Kemerer OO metrics. It processes compiled Java class calculating following metrics:

- WMC: Weighted methods per class
- DIT: Depth of Inheritance Tree
- NOC: Number of Children
- CBO: Coupling between object classes
- RFC: Response for a Class

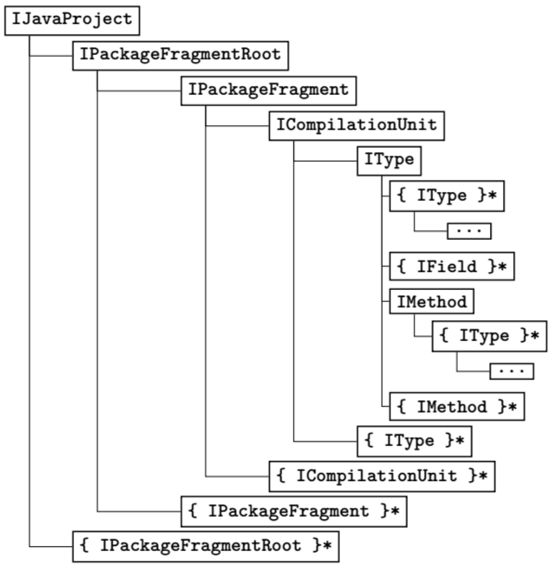


Figure 5.1: Java Elements Hierarchy.

- LCOM: Lack of cohesion in methods
- Ca: Afferent couplings
- NPM: Number of public methods

It's headless, fast and yet reliable calculation makes it a good choice for integration with other tools. Our main concern regarding the choice of metric tool was related to precision, consistency and time consumption. Ckjm was the fastest, consistently giving the same values for the same classes. Furthermore, headless and open source solution was easy to incorporate into a tool.

5.2 Eclipse Platform

Eclipse is an open source project maintained by the community of developers. It is the best known as a Java IDE but it supports a wide range of programming languages, platforms, frameworks and tools through to its extensions and plugins. Its extensibility and openness made it a suitable tool for fast prototyping. This is especially true for software engineering research where Eclipse founds its full potential and support. In this thesis, we are interested in Eclipse Java development support, extensibility and underline refactoring structure. Next paragraphs Eclipse Java support is discussed following with its source code analysis and refactoring.

Eclipse offers support for Java programming language through the Java Developing Tool (JDT) project. It is a top-level project of Eclipse Platform and it provides full support for Java development experience. JDT consists of many sub-parts, from its core packages to graphical user interface experience and its perspective in Eclipse. Of our particular interest are parts related to source code analysis and refactorings. JDT source code analysis of actual Java project is processed on the two levels:

1. The first level consists of modeling Java project with a representation of each of its part as a model element. This model is a light-weight representation of Java project and every part of Java source code (packages, classes, methods, fields, etc.) is modeled with suitable handles. The whole Java program is represented on class-base with a tree-like structure. These handles (elements) are explained in Table 5.1. Its hierarchy is depicted in Figure 5.1 [66]. Since it is a model, the actual existence of the modeled elements has to be checked with the predefined method exists().

2. Actual Java source code analysis and manipulation in JDT are handled in a compiler-alike manner, where the code is analyzed and parsed to create Abstract Syntax Tree (AST) of Java class. The class is divided into tokens by the scanner (lexer) then tokens serve as an input for the parser. The parser analyzes the tokens and verifies its consistency with the grammar of the language. The output is an AST, a heavy-weight actual representation of Java source code that contains detailed information of every element of Java class. AST hierarchy is presented in Figure 5.2 [66]. ASTNode is Eclipse abstract superclass that holds all information related to AST manipulation. Every AST node acts as a child of ASTNode inheriting all of its functionalities. Considering heavy-weight nature of AST and its actual code representation, careful and effective AST operation are crucial for successful code manipulation. Usually, there are some unwritten rules that developers have to adhere when dealing with AST in Eclipse. One of these rules demands that no more than one AST is open in the whole Eclipse workspace. The efficient traversing of the AST is crucial for computationally feasible code manipulation and JDT facilitates different design patterns for this purpose (ASTVisitor, ASTRequester, etc.). For example, AST Visitor, following Visitor pattern [67], allows every AST node to be visited by its visit() method opening gradually its children nodes for traversing. If parent nodes indicate (by its visit method) that it was not traversed, its children are unreachable for traversing and analysis stops saving intentionally time and space of analysis

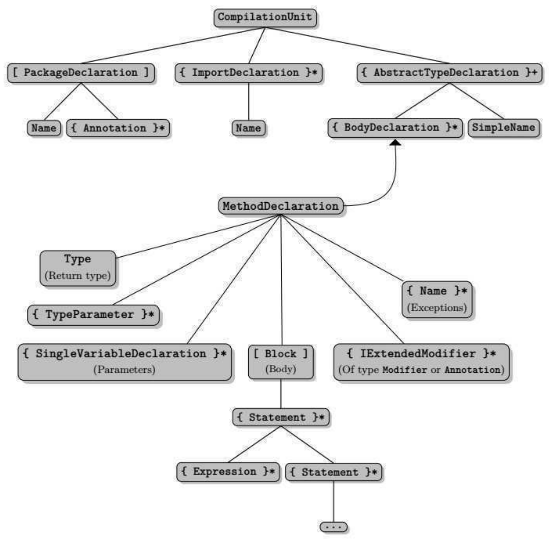


Figure 5.2: Eclipse AST Hierarchy.

procedure. Next to prefix traversal (of visit() method), ASTVisitor offers postfix traversal (with endVist() method), where child node is visited first then its parent node.

5.2.1 Refactoring in Eclipse

Eclipse Platform provides essential refactoring support through to general, language-independent, framework. The Language Toolkit framework (LTK) offers abstract classes for refactoring creation, execution and user interface.

Eclipse refactoring comes with a detailed life cycle:

- Refactoring is initiated by the user (or script) with detailed information - **Refactoring** subclass is created with suitable refactoring action that addresses user's (script's) preferences.

- Initial conditions are checked - **checkInitialCondition(...)** method of

Element	**Description**
IJavaModel	the root Java element, corresponding to the workspace.
IJavaProject	a Java project in the workspace. (Child of IJavaModel)
IPackageFragmentRoot	a set of package fragments
IPackageFragment	a portion (or entire) of the package
ICompilationUnit	a Java source (.java) file.
IPackageDeclaration	a package declaration in a compilation unit.
IImportContainer	the collection of package import declarations in the unit.
IImportDeclaration	a single package import declaration
IType	a source type (of unit), or a binary type (of class)
IField	a field inside a type
IMethod	a method or constructor inside a type
IInitializer	a static or instance initializer inside a type.
IClassFile	a compiled (binary) type.
ITypeParameter	a type parameter.
ILocalVariable	a local variable in a method or an initializer.

Table 5.1: Java Elements.

Refactoring class is executed checking that refactoring action is possible in context issued. If passed, next step is following, otherwise, refactoring is aborted with RefactoringStatus#FATAL status.

- Depends on nature of refactoring task, more information is gathered from the user for the continuation of refactoring action.

- After passing initial conditions, **checkFinalCondition(...)** method is executed with detailed precondition checks. Furthermore, this method attains most of the necessary information for complete refactoring execution and generation of change. If passed, change information is generated in terms of change descriptors for actual change creation in the next step.

- Refactoring task is executed by **createChange(IProgressMonitor)** returning **Change** class object. Change class objects contains all information for possible undo operation and maintenance of refactoring history. With proper execution of refactoring task, the life cycle of refactoring is finished.

This thesis uses automated refactoring tasks without user interaction. In the next chapter, we describe how different tools explained in this chapter are incorporated into reinforcement learning methods to automate refactoring tasks generating a suitable sequence.

CHAPTER 6

REFMARK - AUTOMATED REFACTORING FRAMEWORK

By an automated-refactoring, we are pointing to the fully automated system that doesn't require any interaction with the developer except for the pointing to the class to be refactored. Automated-refactoring outputs a class with, in the best case, all resolved deficiencies (bad-smells, etc). A general framework, for an automated refactoring, considered in this thesis is depicted in Figure 6.1. The first part of the framework is related to the input, a class, to be refactored. A developer is asked to provide a class to be optimized and refactored. After pointing to the class, a framework continues with processing and refactoring execution. As can be seen in Figure 6.1, processing part of the framework has three sub-parts: refactoring tasks, metrics, and methods. Each part defines the subset of dependent variables to be used for processing and complete automation. Variations of these three parts lead to different automated approaches and techniques. Refac-

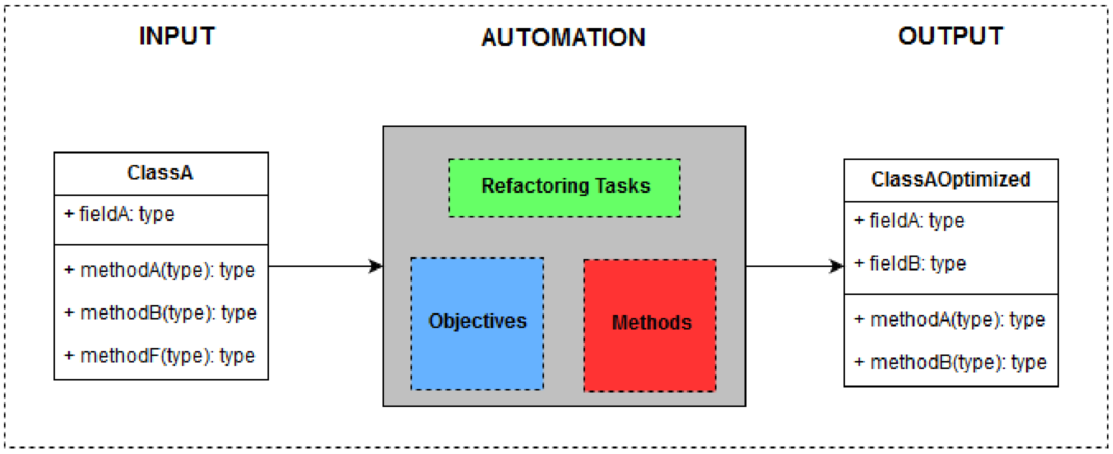


Figure 6.1: Automated Refactoring Framework.

toring tasks may include one or many different refactoring tasks chosen from the catalog of refactoring. Processing and automation can focus on an improvement of the class based on a single metric or it can be multi-objective where automation tries to optimize the class based on many (possibly contradicting) metrics. Third sub-part of a processing is related to the method chosen for optimization of these metrics using tasks. The method can range from simple naive method till more involved and complex methods ranging from search-based to machine learning methods. After the definition of each sub-part of processing framework and its utility, system outputs refactored and optimized class. With the output, automation is finished and class resulted is (at the best case scenario) the desired solution for the proposed processing details. In the next sections, we present a *RefMark* tool that implements general automated refactoring framework following the details discussed in the proposed method chapter.

6.1 Introduction

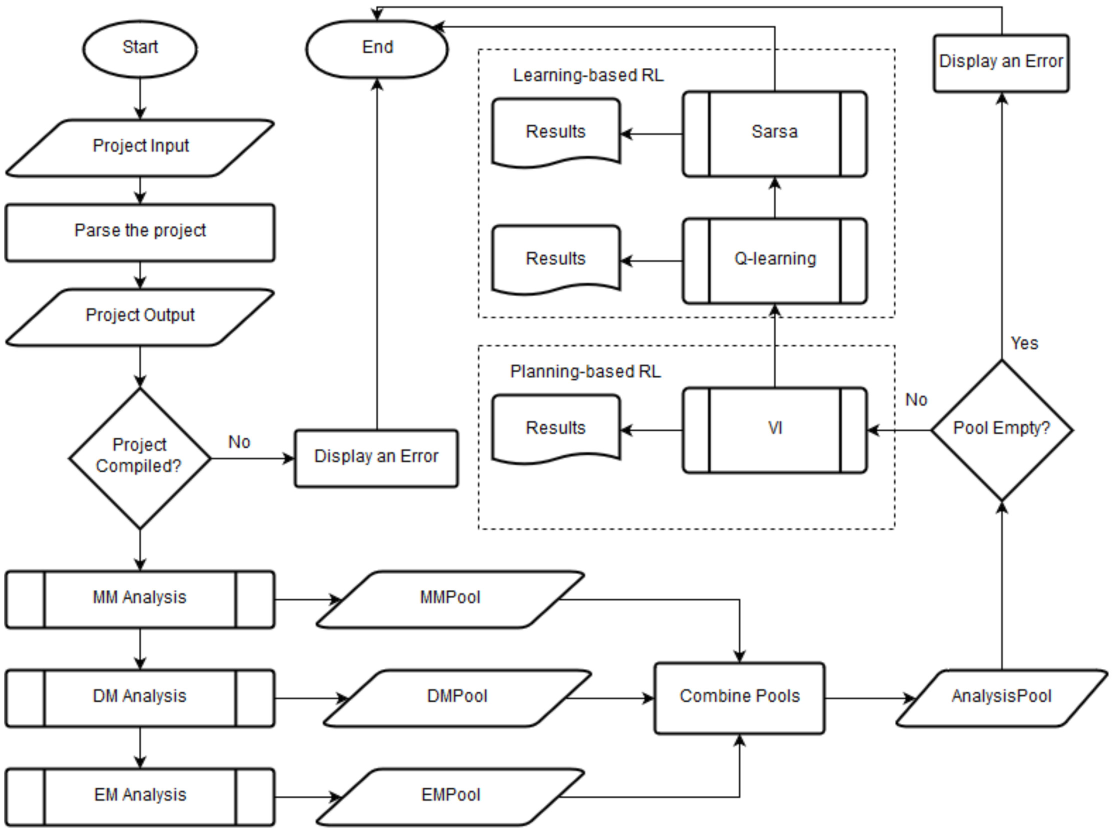


Figure 6.2: Optimized RefMark Flowchart.

RefMark tool is an Eclipse-based plugin that follows the root canal refactoring approach, where a complete system is refactored in one phase in the contrast to the floss refactoring where refactoring of the system, is conducted on a long-term, in small steps [68]. Through the development of *RefMark* tool, it passed different stages and improvements. In general, we differentiate between two major *RefMark* versions, the first we consider as an initial prototype and the second as a more optimized version. Optimized RefMark flowchart is depicted in Figure 6.2. Next paragraphs discuss both versions.

6.2 Initial *RefMark*

RefMark solely depends on Eclipse mechanism for checking preconditions, final condition, and execution. Initial *RefMark* was considering all refactoring tasks possibilities (no matter valid or invalid), without analysis or previous filtering, allowing the agent to learn blindly. This was leading to a generation of a huge number of unfeasible solution scenarios and high expense in computational time and space requirements. For example, from several hours until a day of continuous execution sometimes were not enough to found refactoring sequence for an even a simple class. In optimized *RefMark*, we address this by analysis part as an optimization of procedure to reduce its complexity with assumptions and considerations that:

- not all tasks will preserve semantic behavior
- not all tasks will affect coupling and cohesion values
- not all tasks, executed by Eclipse, will be valid [69, 70]

Initial *RefMark* started with the simplest method that does not consider effects of executed task on the metrics involved and just executes a predefined number of tasks on a random base. **Random Method** is useful for comparison and to describe the usefulness of more sophisticated methods (that are more often much more expensive in the terms of computation complexity for time and space) and their utility and justification. Simply, if Random method outperforms more sophisticated method than more sophisticated method has questionable effects and usually does not justify its additional requirements of resource used (human and computer resources) and has to be considered as a waste. *RefMark* tool's only parameter in the random method is its limit on a number of randomly executed tasks. After the limit is reached, the Random method outputs its sequence of tasks. Even its name points that it is random, discussion behind truly random methods, distribution behind random function, expected value and other theoretical topics are behind the scope of this work and we consider the implementation of Java random function (either SecureRandom or its older predecessor, Random) on "as it is" base. Our main satisfaction with "any" random implementation lie down in the main point that does not require any extra effort or guidance and it does not depend on metrics chosen. **Greedy Method** is the second method that was employed by the first version of *RefMark* tool. Greedy algorithm prefers task that, at each execution step, has more improvement effect than other (two) tasks. For instance, each step greedy (reflex) algorithm executes task available and examines its results (by saving its improvements/degradation) then return system to its starting position and execute next, different, task and again process it (examines and returns to its starting position) and so on till every different task is executed once. After execution of all tasks, greedy method choose the one with the highest positive improvement, executes it and continues with the same process on the next step. The process is continued until the limit of a number of tasks is reached or till no more improvement is observed for 3 consecutive steps (algorithm converged). Beside Random and Greedy method, **Value Iteration** was used as the only reinforcement learning approach in initial *RefMark*. In optimized *RefMark*, random and greedy method were not anymore supported, and learning-based RL methods are employed (Q-Learning and Sarsa).

MDP environment of the initial *RefMark* was normalized and quantized to 10x10 map. For a generalization of each class, to be represented by ”the same” map, coupling and cohesion values are normalized and quantized on 10x10 state map, starting from (0,0) to (1,1) with 0.1 increments on both dimensions. Figure 6.3 depicts the initial MDP Environment. Normalization and quantization phase was later omitted when it is realized that exact values of coupling and cohesion will better represent differences of nature of classes and their shapes.

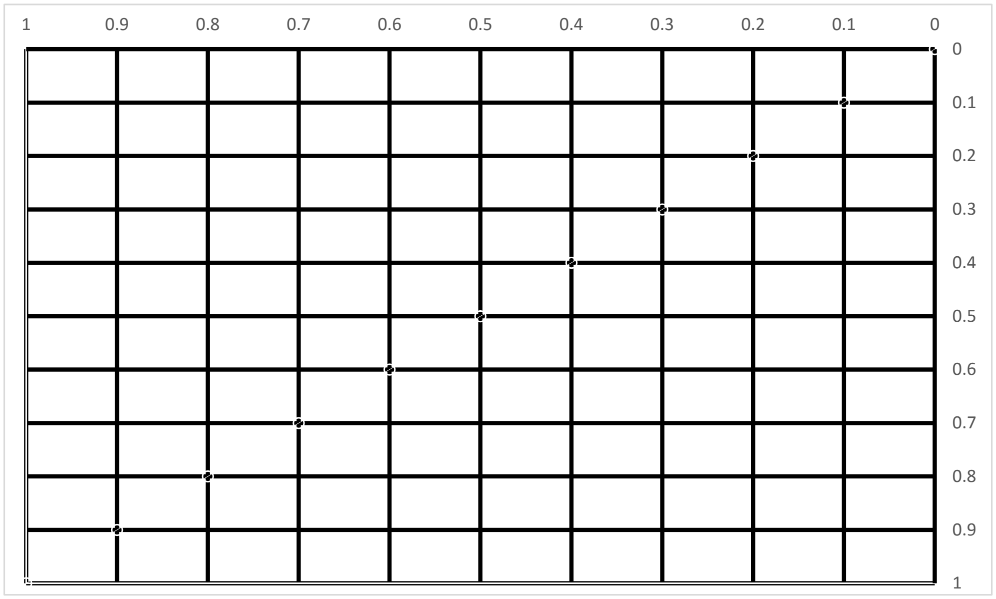


Figure 6.3: Initial MDP Environment.

6.3 Optimized *RefMark*

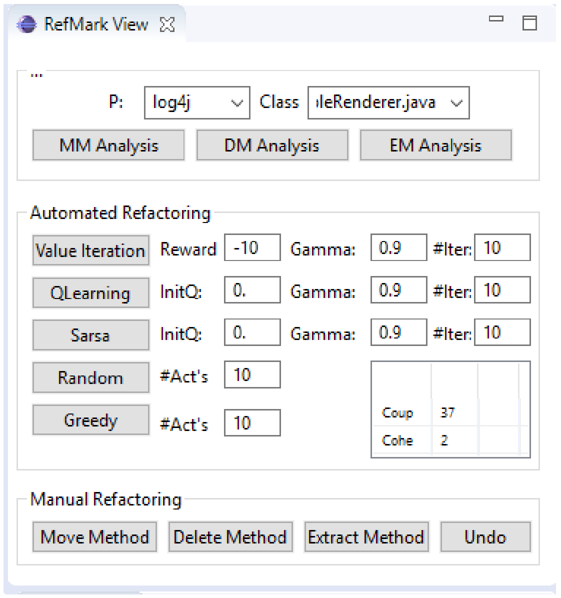
After relatively basic methods for an automation in the initial version, optimized *RefMark* added **Reinforcement Learning-based (RL)** methods while automating the process of the refactoring. Random and Greedy methods do not benefit from their experience and "easily" forget what they faced in the past actions, hence algorithms that do not remember their history are doomed to repeat it. In short, reinforcement learning approaches allowed an agent to learn from its environment. With obvious disadvantages of random and greedy methods in their memory-less activity, the RL-based agent tries to interact with the environment by executing refactoring tasks and observing their immediate effects. While executing refactoring tasks, RL-based agent reinforces its experience by instance reward and overall quality of sequence. Based on our knowledge, this is the first attempt to address refactoring automation problem with this part of a machine learning algorithms. Initial *RefMark* experiments were very costly, as explained. Before optimized *RefMark* analysis layer was introduced, experiments have taken a lot of time and space resources. Next to this layer, the terminal function of *RefMark* domain was changed adding the condition that no agent can stay in the environment for more than a certain time (we choose it to be sixty seconds). Based on manual inspections of results of several experiments conducted with and without an analysis layer, we realized that each algorithm converges very fast and within first 40-50 steps. This means that limiting the time of execution will give better overall execution time and space requirements with the same results.

6.4 Eclipse Preferences and Parsing Projects in *RefMark*

For the *RefMark* better performance and more accurate refactoring tasks, auto-format option for Java code has to be disabled in Eclipse. This also addresses different coding style "issues" in an analysis and decreases possible code-errors due to "auto-format" option. From our observation, we found that Eclipse auto-format could be the reason for the compilation errors between different automated refactoring tasks. For example, if automated approach required execution of refactoring tasks then their reverts, in-between, auto-correct could corrupt starting and end positions of these refactoring changes and simple undo operation will create compilation errors.

RefMark has to meet all general requirements stated for the refactoring in the previous chapters. Next to it, it has to meet some runtime requirements. By **runtime requirements** we mean conditions that are met before refactoring processing is considered:

- J2SE JDK (1.5 and higher) - project has to be on standard J2SE Runtime Environment
- Eclipse 3.5 or higher - IDE where *RefMark* plugin will reside. (The work was tested on Eclipse Neon and Eclipse Oxygen)



P: log4j Class ıleRenderer.java

Value Iteration Reward -10 Gamma: 0.9 #Iter: 10
QLearning InitQ: 0. Gamma: 0.9 #Iter: 10
Sarsa InitQ: 0. Gamma: 0.9 #Iter: 10
Random #Act's 10
Greedy #Act's 10

Coup	37
Cohe	2

Figure 6.4: *RefMark* Plugin - RefMark View.

Runtime requirements also addressed what has to be mentioned for the input of automated refactoring framework and its implementation in *RefMark*, to point out, *RefMark* input has to be Java class and part of Java project. Java project has to be in the workspace (JAVA_NATURE) and build and compiled automatically, by Eclipse, with no compilation errors. Maven (or Ant) and tools with a similar purpose are not guaranteed to work, projects are exhaustively tested with Eclipse Java project nature only. The project must be imported to Eclipse workspace before *RefMark* view is open.

6.5 *RefMark* Analysis Layer

After unfeasible all refactoring tasks consideration in initial *RefMark*, as already mentioned, analysis layer was the main optimization part of the second version of *RefMark*. This layer is divided into three parts, one for every kind of refactoring tasks. Next subsections discuss more each kind of refactoring task and its filtering by analysis layer.

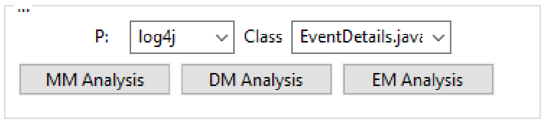


Figure 6.5: *RefMark* View - Analysis.

6.5.1 Move Method Analysis

Move method analysis aims to find all possible MM tasks within the class. It starts with checking the correct state of the class by its compilation. After confirmation that class is in a solid and error-free state, all methods are saved into the pool for consideration of its methods. Next, one by one method is tried to be moved and every successful refactoring tasks candidate is added to the specially created pool for automation. It is worth to mention that class is compiled before and after execution of every task to ensure that generate candidates will not create any compilation error. The output of MM analysis is a pool of possible MM candidates. MM analysis flowchart is depicted in Figure 6.6.

6.5.2 Delete Method Analysis

In the same manner, as MM analysis, DM analysis generates possible candidates for DM refactoring tasks. For both analysis is crucial that tasks do not generate any error and that output pool contains only atomic, error-free actions. DM analysis flowchart is represented in Figure 6.7.

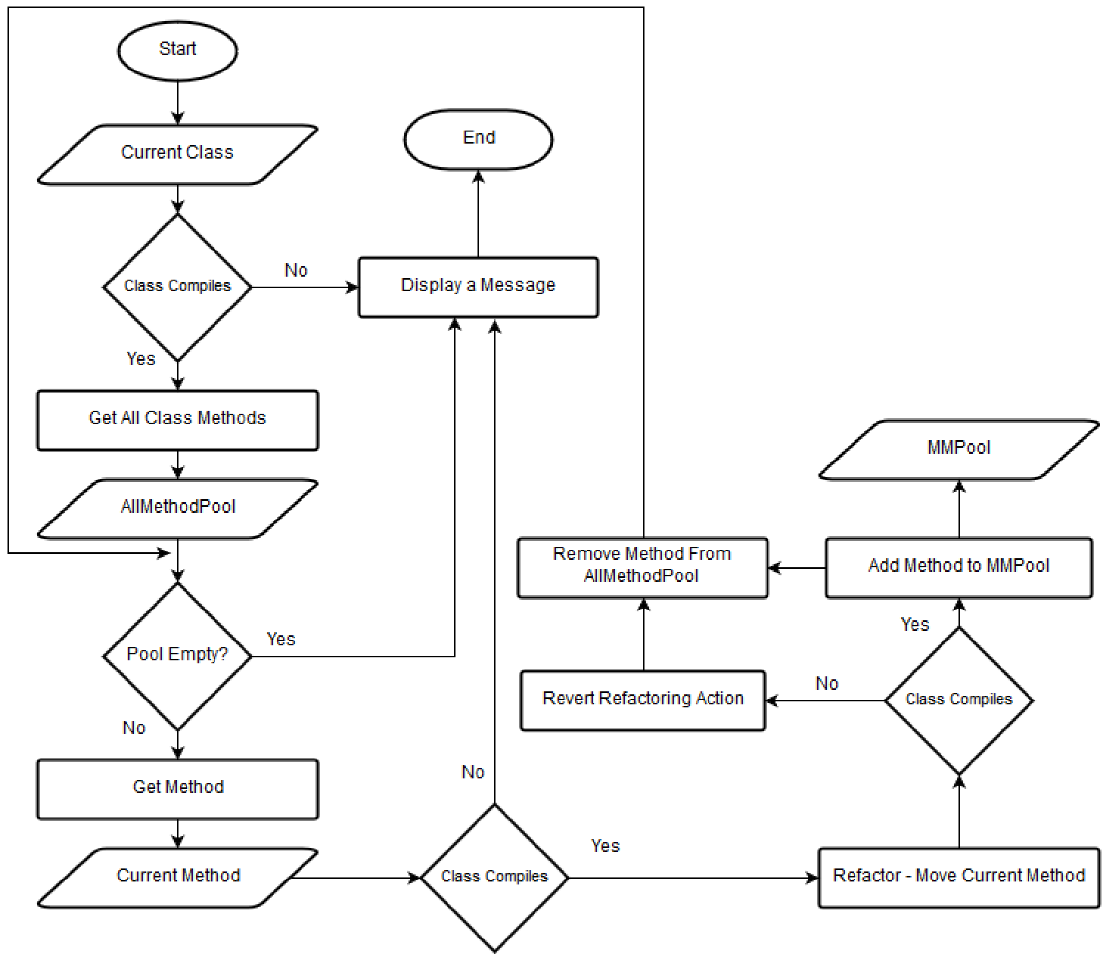


Figure 6.6: MM Analysis Flowchart.

6.5.3 Extract Method Analysis

Finding the candidates for an EM, in the analysis, is based on a brute-force search on all methods in the class. Every method is parsed and analyzed for a possible blocks, methods, return values and invocations that can represent method by-itself. Information is extracted and refactoring is performed for the checking of possible compilation errors, if passed, EM action is added to EMPool where its refactoring details are saved. If an EM analysis fails, we completely ignore an entire EM refactoring pool. From our experience and tests performed, a sequence

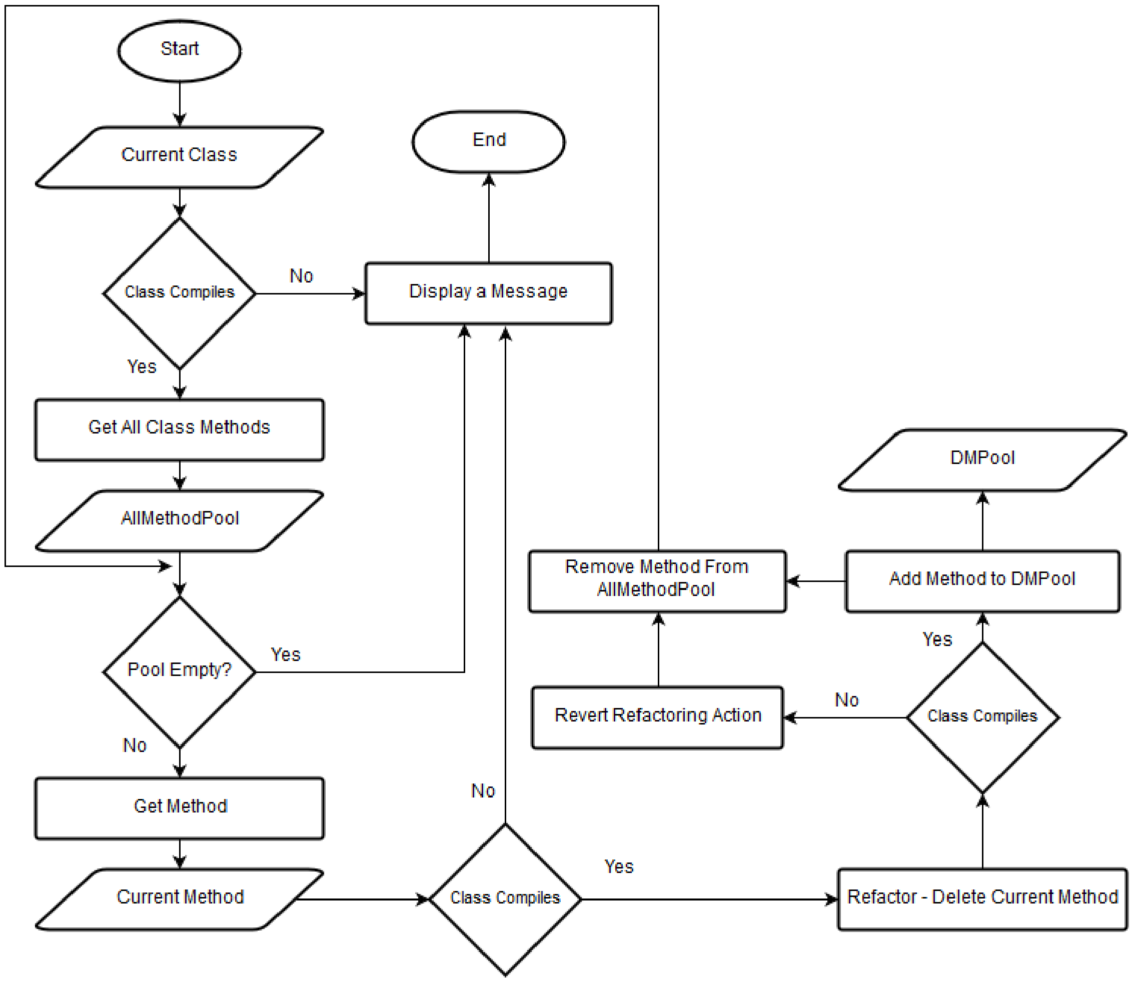


Figure 6.7: DM Analysis Flowchart.

of EM actions may overlap, creating a new method from an already extracted method, then two actions are not anymore independent and atomic (they are composite). Executing single EM action would not be possible anymore without breaking a code and compromising all consecutive tasks leading to an unpractical and incorrect refactoring sequencing wasting computational time and space resources. On the other hand, Move Method and Delete Method Analyses do not have scenarios similar to this since all MM/DM actions are independent and atomic. This is rather a radical approach, but for the sake of time and complete-

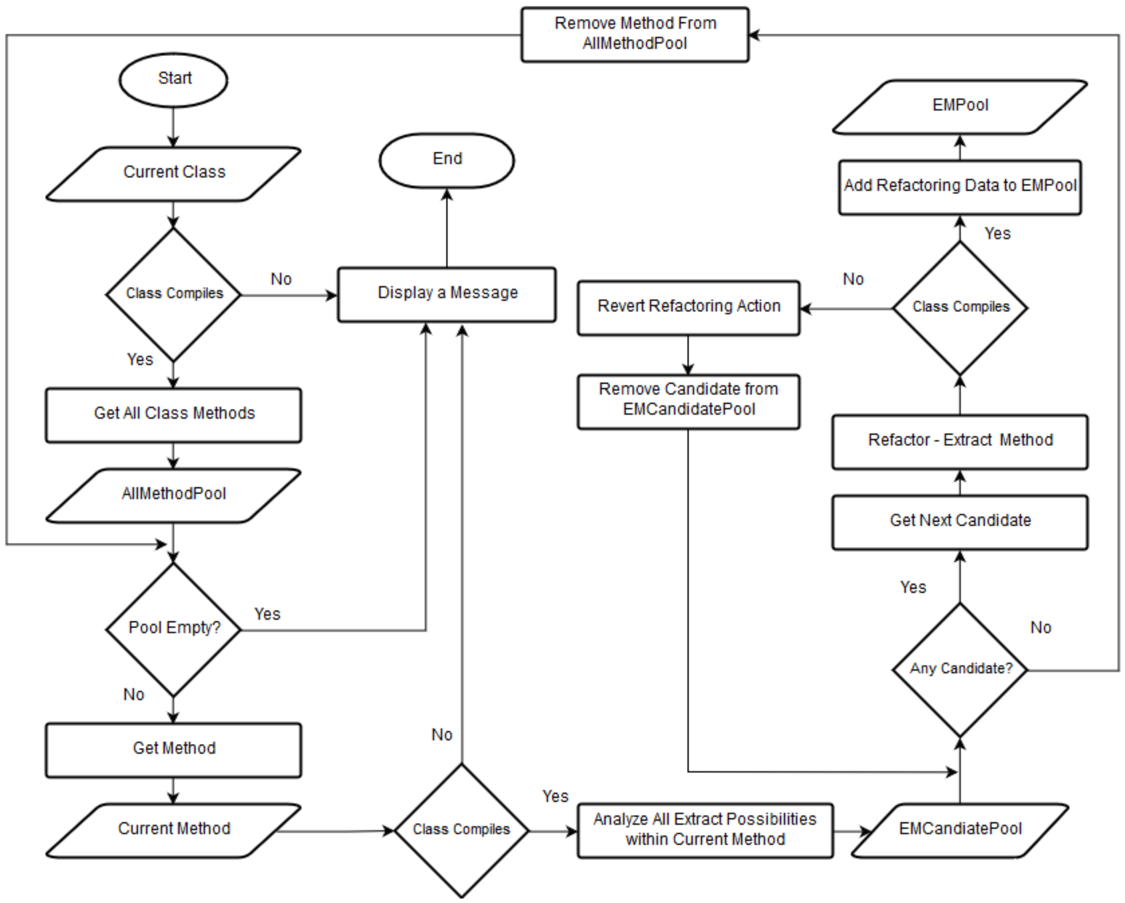
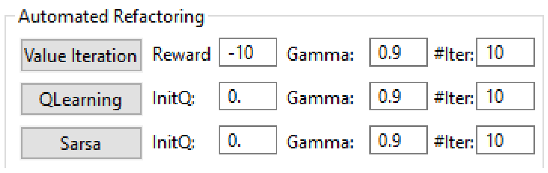


Figure 6.8: EM Analysis Flowchart.

ness, addressing deeper EM refactoring issues is out of the scope of this thesis. EM-related issues require more dedicated and detailed approach, an interesting reader may refer to [70] for more information. EM analysis flowchart is depicted in Figure 6.8. The name chosen for a newly extracted method is with a format: createMethodName_number. The number is generated by a counter that increases with each EM execution. This is to overcome possible obstacles with similar method name already existed in the class, that could create compilation error.

6.6 *RefMark* Automated Refactoring

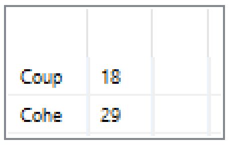
After parsing Java project and choosing the aimed class to be optimized, analysis layer filters reliable refactoring tasks dividing them into the different pools based on their kind (MM, DM or EM). Automated refactoring sequencing is solved by the employment of planning-based (Value Iteration) and learning-based RL methods (Q-Learning and Sarsa). *RefMark* interface for these algorithms is built to provide flexibility to the software teams in deciding each variable for algorithms used. For example, negative reward (cost of executing refactoring) defined by software teams might depend based on the actual cost of developers' hour in their region (the price of an hour of developer might not be the same in the US, for example, and China). A discount factor might vary based on time-space allocated for maintaining phase. If the phase has extended the time period, then heavily discounting the cost of refactoring execution might not be appropriate, but if the period is short, then the current cost might be more valued. The same reasoning might be applied to the number of iterations for each algorithm. It has to be noted also, that after refactoring task analysis is performed, automated and manual refactoring execution of any task will pick randomly from the pool of correct refactoring tasks found. This is because the concentration of this work is on a different kind of refactoring tasks and for us it matters to differentiate between Move Method, Delete Method and Extract Method, what are the object of these tasks is related with a certain class but not with overall analysis and experience.


Automated Refactoring

Value Iteration	Reward	-10	Gamma:	0.9	#Iter:	10
QLearning	InitQ:	0.	Gamma:	0.9	#Iter:	10
Sarsa	InitQ:	0.	Gamma:	0.9	#Iter:	10

Figure 6.9: *RefMark* View - Automated Refactoring.

6.7 *RefMark* Metrics Monitor



Coup	18	
Cohe	29	

Figure 6.10: *RefMark* View - Metrics Monitor.

Displaying metrics, in *RefMark*, is addressed with Metrics Monitor (Figure 6.10) where the initial value for coupling and cohesion is shown in the second column of the table. The third column is reserved for the current values of metrics when tasks are manually triggered (manual refactoring part). This third column is aimed to address direct inspection of tasks' pools and their direct effects on the metrics used.

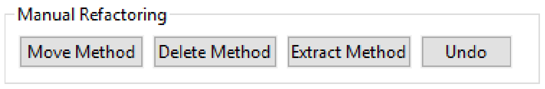


Figure 6.11: *RefMark* View - Manual Refactoring.

6.8 Manual Refactoring

The last part of the *RefMark* tool is reserved for the manual refactoring. Based on analysis of different refactoring tasks and their output saved in the pools, developer can manually examine the effects of every task. Metrics Monitor (Figure 6.10), as mentioned, will display immediate effects of triggered task on the third column stating current coupling and cohesion values.

CHAPTER 7

EXPERIMENTAL DESIGN

RefMark automated framework is empirically evaluated and reported following well-defined software engineering practices [71]. Goals of experiments are defined based on thesis research objectives with an identified experimental material, units, and procedure. Validation continues with hypotheses, experimental design and analysis and discussion. In this and next chapter we present complete empirical evaluation.

7.1 Goals

Based on our general research objectives in this thesis, our goals are related to the effective representation of refactoring sequencing as a Markov Decision Process (MDP) instance. This effective representation has to use all benefits of RL-based approach and be valuable input for different RL-based algorithms. After this representation, employment of different RL-based algorithms is the next step followed by analysis and discussion.

1. Effective representation of refactoring sequencing as a Markov Decision Process (MDP) instance. This effective representation has to use all benefits of RL-based approach and be valuable input for different RL-based algorithms.

2. When effective MDP instance is defined, next goal is to employ different RL-based algorithms to solve this instance in efficient way in terms of computational time and space requirements.

3. Analysis of these different algorithms is our last goal where possible conclusions can be drawn from different insights gained with different settings and quantization of trade-offs.

7.2 Experimental Material, Units, and Procedure

For empirical evaluation of proposed method and tool developed, initial checks for suitable representatives of Java projects are performed for case study evaluation. After initial checks on datasets available, their structure, analysis, and information; currently available public datasets did not contain information that could meet our objectives of reinforcement learning approach employed. Most of the current datasets present refactoring information established on the analysis of projects' refactoring tasks based on the changes between different consecutive versions using some of the tools (Ref-Finder [72, 73], for example), manual inspection or combining both. Having in mind this information, knowing that developers more often do not follow design patterns, software engineering recommended practices [74] and that these automated tools usually are not able to find all refactoring tasks (for example, Extra Class cannot be easily detected by tools), we decided that proposed method requires specially designed data to validate its concept and appropriateness. This data has to have the refactored project that specifically:

- follows the best practices to the maximum extent possible (avoiding bad smells)
- is optimized for a certain software quality aspects (in our case, coupling, and cohesion)
- represents the well-coded mid-range project.

As explained, in Chapter 5, inFusion is able to detect different bad smells in software, on a class, package and project level. For our proposed framework, where we address an improvement in coupling and cohesion of class, our concentration was on the indicators of bad practices in terms of these two software quality measures. Initial pointers were leading to the bad smells. For that particular purpose, all bad smells, detected by the inFusion, were analyzed. Table 7.1 summarizes bad smells, detected by inFusion, and points to the bad smells that were used in our data creation. It has to be mention that our main concentration is on the refactoring tasks that optimize class coupling and cohesion ratios. Since we are focusing on Java-based projects, we are interested in an object-oriented

(OOP) paradigm, class level, and bad smells that affect both, coupling and cohesion values. Bad smells used for a creation of data validation were only indicators. After data is created, a concrete relation of bad smells and results is not examined nor validated. After narrowing bad smells in inFusion (refer to Table 7.2), the log4j project was used as a sample for the validation process. These bad smells were found, by inFusion, in around 20 classes of log4j project. After manual examination, configuration classes were intentionally ignored due to well-known high coupling and cohesion values that are hard to avoid. They are not considered as a malformed nor do contain any bad smell. After exclusion of configuration classes, 11 regular classes were left for experimental evaluation. The author performed refactoring tasks required to address these bad smells and after an exhaustive trail-and-error process of manual refactoring sequencing, the optimal number of refactoring tasks (minimum number of required refactoring tasks to address bad smell in a certain class) is found for each class. Statistics are drawn at the last stage, where we assume that refactoring tasks are refined and optimal. Table 7.5 and 7.3 present this statistic of log4j classes and refactoring tasks applied. To conclude, our experimental material includes a log4j Java project and its regular classes that jeopardize good software practices negatively affecting coupling and cohesion. These classes represent experimental units and each class is the object of independent case study and evaluation. The summary of all classes with their manual refactoring tasks is depicted in Table 7.5.

Smell	Coupling	Cohesion	Paradigm	Entity
Blob Class	Yes	Yes	oop	class
Blob Module	Yes	Yes	p	module
Blob Oper.	No	No	oop, p	operation
Cyclic Dependency	Yes	No	oop, p	subsystem
Data Class	Yes	Yes	oop	class
Data Clumps	Yes	No	oop, p	operation
Data Module	Yes	Yes	p	module
Distorted Hierarchy	No	No	oop	class
External Duplic.	Yes	No	oop, p	operation
Feature Envy	Yes	Yes	oop, p	operation
God Class	Yes	Yes	oop	class
God Module	Yes	Yes	p	module
Intensive Coupling	Yes	No	oop, p	operation
Internal Duplic.	No	No	oop, p	operation
Message Chains	Yes	No	oop	operation
Refused Parent Bequest	No	Yes	oop	class
Stable Abstraction Breaker	Yes	No	oop	subsystem
Schizophrenic Class	Yes	Yes	oop	class
Schizophrenic Module	Yes	Yes	p	module
Shotgun Surgery	Yes	No	oop, p	operation
Sibiling Duplic.	Yes	No	oop	operation
Tradition Breaker	Yes	Yes	oop	class
Underutilized Interface	No	Yes	p	module
Unstable Dependency	Yes	No	oop, p	subsystem

Table 7.1: Bad Smells Detection - inFusion.

7.3 Hypotheses

Our hypotheses are designed to address defined goals and research objectives. While our first two objectives are related to effective representation and utilization of different algorithms, the third objective aims to quantify and analyze this representation. In that direction, we have four hypotheses that validate proposed method and evaluate its different settings.

Null Hypothesis 1 *Proposed automated methods are not able to meet optimal*

No.	Bad Smell
1	God Class
2	Data Class
3	Feature Envy
4	Schizophrenic Class
5	Blob Class
6	Tradition Breaker

Table 7.2: Coupling and Cohesion Related Bad Smells.

Task	Abbr.
Delete Class	DCl
Delete Field	DF
Delete Method	DM
Extract Class	EC
Encapsulate Field	EF
Extract Method	EM
Move Class	MC
Move Method	MM
Temp to Local	TtL

Table 7.3: Refactoring Tasks Abbreviations.

manual refactoring in terms of number and kind of refactoring tasks.

Alternative Hypothesis 1 *Proposed automated methods are able to meet optimal manual refactoring in terms of number and kind of refactoring tasks.*

Null Hypothesis 2 *Proposed methods are not able to improve coupling and cohesion on class-level.*

Alternative Hypothesis 2 *Proposed methods are able to improve coupling and cohesion on class-level.*

Null Hypothesis 3 *Planning-based RL algorithm has faster converge rate comparing to learning-based algorithms.*

refactoring Task	**#Instances**
Delete Class	1
Delete Field	1
Delete Method	36
Encapsulate Field	9
Extract Class	1
Extract Method	1
Move Class	1
Move Method	56
Temp to Local	2

Table 7.4: Log4j: Refactoring Tasks Applied.

Class	Refactoring Tasks								
	DCl	DF	DM	EC	EF	EM	MC	MM	TtL
EventDetails	1							8	
LogRecord			1		6	1	1	2	
TTCCLayout								3	
LevelRangeFilter			5						
AdapterLogRecord			5					5	
JDBCAppender								4	
LogBrokerMonitor		1		1	3			28	
Hierarchy			16					1	
XMLLayout								1	
SyslogAppender			2					4	2
MyTableModel			7						
SUM:	1	1	36	1	9	1	1	56	2

Table 7.5: Classes and Refactoring Tasks Executed.

Alternative Hypothesis 3 *Planning-based RL algorithm does not have faster converge rate comparing to learning-based algorithms.*

Null Hypothesis 4 *Planning-based RL algorithm requires less number of steps for complete execution comparing to learning-based algorithms.*

Alternative Hypothesis 4 *Planning-based RL algorithm requires more number of steps for complete execution comparing to learning-based algorithms.*

7.3.1 Variables and Software Quality Measures

Refactoring process, in general, has two key players that affect complete process: refactoring task and quality measures affected by these tasks. For the experimental part, these two players represent independent and dependent variables.

Independent Variables - Refactoring Tasks

From the refactoring tasks applied, Table 7.4, we can conclude that Move method, Delete Method, Encapsulate Field, and Extract Method are the most used refactoring tasks. From these four refactoring tasks, three of them are related to the method-based refactoring and one is related to the field (variable). This is important to notice since for the automated refactoring we have to choose refactoring tasks that are concentrated on a similar level to minimize, as much as possible, the distance between different refactoring tasks, maximizing their overall effect on the class. To sum up, Move Method, Delete Method and Extract Method are chosen for an automated refactoring tasks and their effects are examined. There are several reasons behind why proposed method automates only these three refactoring tasks:

- The first reason is related with **comparability** with manual refactoring output since they are one of the most used refactoring tasks.
- The second reason is related to a class-based optimization where proposed method improvement is related to a class and these three refactorings have the main effect on the class optimization.

- The third reason is related to **practicability**, in terms that the refactoring tasks chosen are with the maximum possible effects on the quality measures used with a minimal number of used operations.

- The fourth reason is related to the **complexity** of the proposed automated system. Complexity in terms of computational time and space of the proposed solution. Any refactoring task added to the framework exposes the system to an extra degree (exponentially) of its computational complexity. With a more combination added, more time is required by the system to execute them, more space is needed to save their execution. Knowing refactoring AST-related execution of a code manipulation, a considerable amount of time and space is needed for each refactoring task added. Complexity is also one of the main reason why three refactoring tasks are only used for an automation. Simply, limiting refactoring tasks assure feasibility of the automation. And authors believe that these three refactoring tasks are a good base for an evaluation of automated refactoring approach, and, if improvement is recognized, after its tests, adding more refactoring tasks might be a part future work path.

Dependent Variables - Quality Measures

Quality measures used by any automated refactoring tool have been self-discriminative and have to be affected by the refactoring tasks executed. Next to these requirements, metrics have to be widely used, accepted and acknowledged by the software developing practitioners. Many software metrics are proposed in the literature and many of them have complex formulas that aim to address a certain software qualities. Our main goal of automated software refactoring is related to improving a coupling and cohesion of a certain class. Beside improvements, the ratio between a coupling and cohesion that suits the best certain purposes is another objective of the proposed method. For these, and other reasons, authors decided that standard Chidamber and Kemerer Metrics (CK) [75] related with a coupling and cohesion of the class will be the strong base for the proposed automated approach. Specifically, ckjm tool was used [65] with its RFC and LCOM software metrics calculation. Authors admit that CK metrics are well-studied and many drawbacks have been stated and analyzed with other metrics' suites proposed [76, 77, 78, 79, 80], but their simplicity, and yet descriptiveness make them suitable prototyping mean. Path of the future work can include consideration of other software metrics.

CBO coupling metric was initially used for a class coupling calculation. After several experiments, discriminative power of CBO was very low in the comparison with the cohesion metric used (LCOM). For example, we could execute many refactoring tasks that highly affect cohesion value while coupling changes were minimal. This was leading to diminishing effects of single refactoring task on a coupling. After several experiments, analysis, and different coupling metrics, we found that RFC coupling metric would be a better counterpart for our cohesion metric used (LCOM). After deploying RFC metric, our coupling and cohesion have almost similar discriminative power and every single refactoring task would affect both, no matter how that refactoring task was small in the amount of code changed.

Next to the main dependent variables, we have another measure required for meaningful analysis of the results of proposed methods.

For the measuring of the general success of particular algorithm in its ability to reach the end without producing an error on class-level, we have the measure called: Level of Success. Level of Success calculates ratio or percentage of successfully completed executions of the certain algorithm on the batch of case studies:
LoS - number of successful executions
nE - successful executions

$$\%LoS = \frac{LoS}{nE} * 100\% \tag{7.1}$$

For accuracy of proposed methods, we need a suitable measure that will determine the similarity of results between proposed methods and manual refactoring. For that purpose we use Cosine Similarity:

$$\mathbf{cosineSimilarity}(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{||\boldsymbol{x}|| \cdot ||\boldsymbol{y}||} \tag{7.2}$$

Cosine Similarity has shown successful in text-based similarity calculations. Moreover, it is length agnostic and provides very efficient measure regardless of the length of its input.

Improvement of coupling and cohesion is measured by the difference between its default and achieved values:

Cp - starting coupling value

CCp - convergence coupling value

ICp - improvement coupling value

$$\mathbf{ICp} = Cp - CCp \tag{7.3}$$

Ch - starting cohesion value

CCh - convergence cohesion value

ICh - improvement cohesion value

$$\mathbf{ICh} = Ch - CCh \tag{7.4}$$

Here convergence points to values achieved after exact solution with refactoring tasks applied and compile-error-free class. In the case that class achieved certain coupling and cohesion values but the class was not compile-error-free we consider only coupling and cohesion values till the class was maintaining compile-error-free state and rest of tasks are not considered into solution but are listed in table results (in Refactoring Tasks part, column Other with X added notation to task name).

For the efficiency measurement of proposed methods, a number of steps was a relevant indicator. We differentiate between a number of steps required for convergence and the end of execution.

Conv - number of steps required for convergence of method

Exec - number of steps required for overall execution of method

With the assumption of normal distribution of data, standard statistics is used to quantify the significance of given results and its confirmation or rejection of null hypotheses.

7.4 Design

Our experiments were divided into case studies (one for each unit of experimental material). Each case study consisted of manual refactoring part, analysis, and automated refactoring part. Automated refactoring part was divided based on the algorithm used for automation. In the next subsections, parts of the case study are discussed with an explanation of results presentation.

7.4.1 Case Study Class Description

A general description of the nature of the object of the case study is given with the coupling and cohesion values presented. Also, UML Class Diagram (or shortly: class diagram) is drawn. It is worth to mention that UML relationships (dependency, association, generalization, and realization) are shown for the classes that are created within the project of the case study class. That means that relationships with well-known Java classes (for example, from java.* packages) are not included in the UML diagram. This simplifies the presentation of UML diagram and gives a space for analyzing only relationships that are tightly related with refactoring sequence. In case that any well-known Java class is crucial for analysis, its relationship will be included in a diagram shown.

7.4.2 Manual Refactoring

Manual refactoring includes the results of an inFusion analysis of the object of a case study following the manual inspection of the required refactoring tasks to optimize coupling and cohesion values. Default values are presented in a tabular form for further analysis and comparison. Manual refactoring tasks are executed and post-analysis is done with the presentation of the results and conclusion of manual refactoring.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:		**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring											
	Analysis										
	VI										
	Q Learning										
Automated Refactoring	**Sarsa**										

Table 7.6: Results Sample.

7.4.3 Automated Refactoring

After inFusion and manual inspection, different RL algorithms were used to solve an RL-based instance of the refactoring problem, as explained in the previous chapters. Different parts of an automated analysis are divided into different subsections and discussed individually. Automated analysis is done for suitable refactoring task's candidates and then each algorithm is employed to find the best-refactoring tasks for optimization of class coupling and cohesion values. There-after, the results are presented for each of the algorithms with suitable analysis. Each algorithm outputs the results that are divided into refactoring tasks' column,

Variable	**Value**
P()	.33
Stochastic	80% — 20%
R	-10 — +100
γ	.9
#iter	10

Table 7.7: VI Preferences.

steps' column and coupling and cohesion values' column.

1. **Refactoring tasks' column** represents the number of instances of each refactoring task sequenced with the algorithm used.

2. **Steps' column** outputs the sum of the steps (actions, in terms of RL instance) for the algorithm that reached the certain point. We differentiate between:

    - the number of steps till algorithm converged to the solution (the last step where the difference in a sequencing of refactoring tasks' optimization results occurred)

    - the number of all steps in the execution of an algorithm, even after no difference in the results is observed.

3. **Coupling and cohesion column**, also, based on the convergence of algorithm, differentiates between coupling and cohesion values till convergence and at the end of execution of an algorithm.

Variable	**Value**
Stochastic	80% — 20%
initQ	0
γ	.9
#episodes	10
#iter	10

Table 7.8: Q-learning Preferences.

All results, for every case study(class), are gathered in a table, for more appropriate presentation and comparison. The sample can be found in Table 7.6. For hypothesis testing purposes, parts of this table from all case studies are extracted and grouped together to confirm or rebut certain null hypothesis. Transition probabilities are defined for each refactoring task (1/3 probability for each task to be executed) with stochastic nature (80% of time intended action is executed, 20% of the time the agent goes with the random action). Developer's cost for execution of refactoring task is presented as a -10 with the goal reward +100. A discount factor was equal to .9 and it was consistent through all iterations. VI executed with 10 iterations for each case study. VI settings are also depicted in Table 7.7. Q-learning agent learned the model of each case study (class) with discount factor equals to .9, executing 10 iterations after 10 episodes of learning. Q-learning details are also denoted in Table 7.8. The same preferences of the Q-learning were used for Sarsa except that Sarsa agent was learning the environment in 50 episodes.

CHAPTER 8

ANALYSIS AND DISCUSSION

After manually refactored dataset, *RefMark* plugin was used to automatically refactor the same dataset again and results are presented in the next sections, with the analysis and the discussion.

8.1 Analysis

Our experimental design described case study based experiments. In next paragraph, we present a case study of EventDetails as one of the 11 case studies performed. EventDetails case study presents the results with detailed explanations. Results are generated in tables, as explained in experimental design. The rest of case studies, we present their results only while a detailed explanation is left to avoid redundancy of information.

8.1.1 Case Study - EventDetails

EventDetails is a small class in the log4j project (LOC: 121, Coupling: 18, Cohesion: 29). Relatively small class with highly coupled methods and very low cohesion.

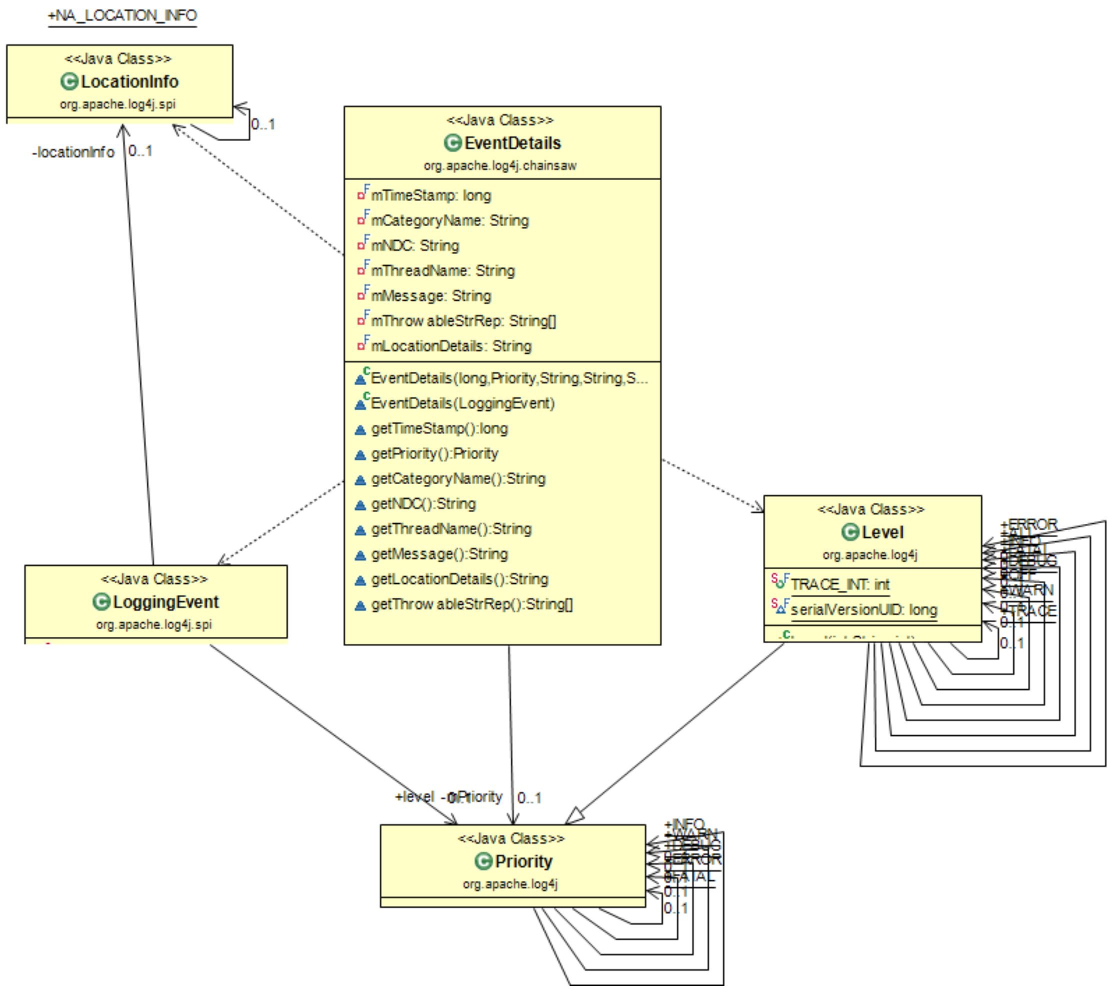


Figure 8.1: EventDetails Class Diagram.

Manual Refactoring

Detailed analysis of this class, by inFusion, followed with manual inspection, led to the conclusion that this class doesn't justify its existence and it suffers from the several design issues. As can be seen from the class diagram depicted in Figure 8.1, all EventDetails class methods are actually "get" methods that collect data from the other classes that are coupled with it. This class only absorbs what has been offered from the other classes without active interaction (extending or offering a new functionality to the system). This is the classical example of Data Class bad smell where a class acts as a parasite absorbing what has been offered by others only without any active role inside its methods or in an overall design. Regarding

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	EventDetails	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		8	-	-	1 x DCI			18		29	
Automated Refactoring	**Analysis**	8	8	0	-						
	VI	6	2	0	-	20	27010	10		1	
	Q Learning	3	5	0	-	22	14351	10		1	
	Sarsa	1	7	0	-	26	24468	10		1	

Table 8.1: EventDetails Results.

our main objective related to a coupling and cohesion improvement, EventDetails has very low cohesion and very high coupling value. To improve its cohesion part and reduce coupling, we have to bring its accessed data to this class from other classes. Looking more precisely at its coupled classes and data, this does not seem to be an appropriate approach where all coupled classes are well-designed with appropriate overall function and good coupling and cohesion balance. If data accessed by EventDetails is moved to it, there is a high probability that coupled classes will be highly affected creating a sequence of other issues and breaking all its design principles. A better approach was to move all methods to coupled classes, near to data they are accessing, improving the cohesion of coupled classes and reducing EventDetails coupling. Since all EventDetails methods are getters, all of them should be moved to their respective classes from where they accessed data. Indeed, by manual refactoring, all methods were moved and the class was deleted. Details of manual refactoring task are presented in Table 8.1.

Automated Refactoring

The automated refactoring analysis is divided into analysis and algorithms sections.

Analysis

Move Method (MM) analysis found that 8 methods were represented as valid candidates for MM refactoring tasks. Delete Method (DM) analysis found also that 8 methods could be safe candidates for DM refactoring tasks. Extract Method (EM) analysis did not find any suitable candidate for EM refactoring tasks. So automated refactoring was used to run on MM and DM refactoring tasks sequencing.

Value Iteration

Value Iteration (VI) was the first RL algorithm used to try to find the optimal refactoring tasks for an EventDetails. As introduced in the second chapter, section Reinforcement Learning, Vi is planning-based RL method for solving an MDP instance and as a such, it requires the model of the environment. The model, explained in the previous chapters, defined transitions probabilities for each refactoring task. The actions (refactoring tasks) also included a stochastic environment with 20% of time randomly executing an unintended action.

Reward Function was uniform; for each action, it was the cost of -10. This severe cost was combined with the reward of the goal state, +100, and the point of this high cost for each action was for the agent to make his living in the domain painful so that the agent quickly tries to find the goal or to quickly finish his refactoring tasks' sequencing. A discount factor was equal to .9 and this relaxed the agent's future cost experience. Due to high expenses in a timely execution of a real automated refactoring, VI was executed with the 10 iterations. This was enough time and iterations, for all case studies, except one, to converge to an exact solution.

VI found that EventDetails has to be refactored with moving the six of its method and deleting the rest (two) methods. It converged to the solution in the first 20 actions executed while the algorithm was ended with the overall 27010 steps in all 10 iterations. Since algorithm converged without producing any error, converged and end of execution generated the same optimized, coupling and cohesion, values. Coupling was reduced to 10, and the lack of cohesion was reduced to 1.

VI approached moved six of EventDetails' methods and deleted the rest 2 methods, if we know that no method is left in the class, why the coupling is still high?! EventDetails constructors are the main consumers of others' class information and automated refactoring approach did not include constructors in a moving or deleting refactoring tasks, they are intentionally skipped.

On the overall, VI optimized the class with the remarkable reduction of its coupling and lack of cohesion values. According to this algorithm, EventDetails suitable values are 10 for the coupling and 1 for the lack of cohesion value. Results are also presented in Table 8.1.

Q-learning

The second, RL-based, algorithm used for an automated refactoring sequence is Q-learning. As discussed in the previous chapters, this learning-based RL method doesn't require a model and instead allowed the agent to learn a model from the interaction with the environment. Due to time constraints and high complexity of real-time refactoring, the agent was learning in not more than 10 episodes the environment with initial Q values set to zero. A discount factor was the same as the one used with VI, 0.9, and algorithm was executing in 10 iterations.

Q-learning optimized the class with moving three methods and deleting the rest five methods. It converged in 22 steps and the overall running algorithm was executing for 14351 steps. As the case of VI algorithms, since no mistake was found in an automated refactoring, converged and the end values for coupling and cohesion are the same, 10 and 1, respectively. Q-learning converged at a slower rate than VI (with the difference of 2 steps only), but its overall execution required significantly less number of steps to end the execution than it was the case of VI. In that case, Q-learning was a much cheaper solution in terms of the time complexity. As all RL method depends on the same analysis of class, q-learning also did not refactor constructors and their access to coupled classes remained what can be seen from the same results produced. Results are also presented in Table 8.1.

Sarsa

Sarsa is the third algorithm used for an automated approach. It is another learning-based approach that allows an agent to learn the model from an interaction with the environment. The same settings used for Q-learning were used for Sarsa (initial Q values equal to zero, discount factor equals 0.9, 10 iterations) except that Sarsa was allowed 50 episodes to learn the environment. This number of episodes were chosen by default following the nature of Sarsa that it doesn't take the maximum expected Q-value of the next state, but its discounted return, (refer to the formula of Sarsa in the background chapter) and it can require more episodes till learning the model in an efficient way. Sarsa achieved convergence after 26 steps with overall 24468 actions executed. If we have in mind that 50 episodes were used for learning, this huge number of steps did not surprise where Q-learning with 10 learning episodes have 14351 overall steps. Results are also presented in Table 8.1.

It is worth to mention that EventDetails, indeed, was just a part of "workover" done by the developers to overcome some compile errors of accessing LoggingEvent and it was not intentionally designed as a part of the Log4J project. This makes the more clear situation of this poorly designed class in a well-design project coded by experienced programmers. The results of other case studies are presented in their tables in this chapter, while their UML diagrams can be found in appendices.

8.1.2 Hypothesis Testing

Null Hypothesis 1 *Proposed automated methods are not able to meet optimal manual refactoring in terms of number and kind of refactoring tasks.*

Alternative Hypothesis 1 *Proposed automated methods are able to meet optimal manual refactoring in terms of number and kind of refactoring tasks.*

For the analysis of the first hypothesis, related to the validation of proposed methods, we analyzed results of all methods in terms of their accuracy. Accuracy here represents to what extent results match the optimal combination of refactoring tasks. For that purpose, we use Cosine similarity measure to determine the similarity of each case study for every method proposed. Next to average similarity, the calculation of a number of case studies with accuracy greater than 50% is considered. Final results are depicted in 8.2. Moreover, methods are divided into planning-based and learning-based and its results are denoted in Table 8.3. It is worth to mention, in terms of the success of proposed methods, two cases studies (out of eleven) did not succeed to finish execution without error. For the sake of reliability, these two case studies were not counted in analysis related to refactoring tasks.

Null Hypothesis 2 *Proposed methods are not able to improve coupling and cohesion on class-level.*

Alternative Hypothesis 2 *Proposed methods are able to improve coupling and cohesion on class-level.*

ClassName	VI-cosSim		Q-cosSim		Sarsa-cosSim	
EventDetails	VI	0.949	Q-Learning	0.514	Sarsa	0.141
LogRecord		0.408		0.807		0.560
TTCCLayout		1.000		1.000		0.000
LevelRangeFilter		0.949		1.000		0.894
AdapterLogRecord		0.894		0.949		0.707
JDBCAppender		0.077		NN		0.000
XMLLayout		0.447		0.800		0.894
SysLogAppender		0.949		0.447		0.447
MyTableModel		0.000		NN		1.000
Average	0.63		0.79		0.52	
Standard Deviation	0.40		0.23		0.39	
>=.5	5		6		5	
<.5	4		1		4	

Table 8.2: Accuracy of Proposed Methods.

	Planning-Based	Learning-Based
Average	0.63	0.64
Standard Deviation	0.40	0.35
>=.5	5	11
<.5	4	5

Table 8.3: Planning-Based vs Learning-Based Overall Accuracy.

The second hypothesis examines to what extent proposed methods were able to improve coupling and cohesion. For that purpose, convergence results of coupling and cohesion of all case studies were analyzed. Next to convergence results, improvements are calculated with emphasizing average improvement of each algorithm. Table 8.7 presents the output of all results.

Null Hypothesis 3 *Planning-based RL algorithm has faster converge rate comparing to learning-based algorithms.*

Alternative Hypothesis 3 *Planning-based RL algorithm does not have faster converge rate comparing to learning-based algorithms.*

Null Hypothesis 4 *Planning-based RL algorithm requires less number of steps for complete execution comparing to learning-based algorithms.*

Starting Values			Convergence								
ClassName	**Cp**	**Ch**	**VI**	**VI - CCp**	**VI - CCh**	**Q-Learning**	**Q - CCp**	**Q - CCh**	**Sarsa**	**Sarsa - CCp**	**Sarsa - CCh**
EventDetails	18	29		10	1		10	1		10	1
LogRecord	49	306		13	13		14	15		19	48
TTCCLayout	25	0		24	0		24	0		24	0
LevelRangeFilter	12	4		8	0		8	0		8	0
AdapterLogRecord	24	22		10	0		10	4		23	20
JDBCAppender	54	177		38	27		NN	NN		38	27
XMLLayout	35	6		34	5		32	4		32	4
SysLogAppender	67	105		57	39		66	87		66	98
MyTableModel	68	66		67	46		NN	NN		66	73
Improvement											
ClassName			**VI**	**VI - ICp**	**VI - ICh**	**Q-Learning**	**Q - ICp**	**Q - ICh**	**Sarsa**	**Sarsa - ICp**	**Sarsa - ICh**
EventDetails				8	28		8	28		8	28
LogRecord				36	293		35	291		30	258
TTCCLayout				1	0		1	0		1	0
LevelRangeFilter				4	4		4	4		4	4
AdapterLogRecord				14	22		14	18		1	2
JDBCAppender				16	150		NN	NN		16	150
XMLLayout				1	1		3	2		3	2
SysLogAppender				10	66		1	18		1	7
MyTableModel				1	20		NN	NN		2	-7
		AVG		**10.1**	**64.9**		**9.4**	**51.6**		**7.3**	**49.3**
		STD		**11.2**	**97.8**		**12.2**	**106.1**		**9.8**	**92.2**
		LoS		**9**	**9**		**7**	**7**		**9**	**9**
		%LoS		**81.82%**	**81.82%**		**63.64%**	**63.64%**		**81.82%**	**81.82%**

	Planning-Based		Learning-Based	
	ICp	**ICh**	**ICp**	**ICh**
AVG	**10.1**	**64.9**	**8.4**	**50.5**
LoS	**9**		**16**	
%LoS	**81.82%**		**72.73%**	

Table 8.4: Overall Coupling and Cohesion Results.

Alternative Hypothesis 4 *Planning-based RL algorithm requires more number of steps for complete execution comparing to learning-based algorithms.*

The last two hypotheses are related to the efficiency. By the efficiency, we mean a number steps required for the method to converge to an exact solution. Next to convergence steps (Conv), efficiency is also depicted with the all steps taken to the end of execution of the method (Exec). The difference is a simple, converge steps are steps required to reach the final state of coupling and cohesion while maintaining compile-error-free class. On the other hand, execution steps are all steps taken to the end of execution, regardless of the state of the class. Based on convergence and execution steps we are able to differentiate efficiency of different algorithms. Results are presented in Table 8.5. Next to efficiency, we analyze how the number of tasks correlates with convergence and execution steps. The result is depicted in Table 8.6.

	Analysis			Tasks	VI	Value Iteration		Q-Learning	Q-Learning		Sarsa	Sarsa	
ClassName	MM	DM	EM	max()		CSteps	AllSteps		CSteps	AllSteps		CSteps	AllSteps
EventDetails	8	8	0	8		20	27010		22	14351		26	24468
LogRecord	26	22	5f	26		67	58549		226	14780		80	10159
TTCCLayout	8	6	0	8		21	23848		39	15051		22	18284
LevelRangeFilter	7	6	0	7		26	17940		19	39803		19	28697
AdapterLogRecord	7	6	3f	7		30	17981		44	25443		37	31373
JDBCAppender	22	13	0	22		40	39920		83	9981		35	42471
LogBrokerMonitor	103	19	39f	103		NN	711		NN	122		NN	135
Hierarchy	25	5	10f	25		119	5968		117	7087		135	7551
XMLLayout	7	4	0	7		24	6406		18	29765		12	23813
SysLogAppender	19	8	12f	19		77	8253		70	9361		52	9270
MyTableModel	17	12	3f	17		42	9014		42	7467		47	11981
					AVG	46.6	19600		68	15,746		46.5	18927.5
					STD	32.0	17221.4		63.9	11,551		36.8	12492.4
					LoS	10	11		10	11		10	11
					%LoS	90.91%	100.00%		90.91%	100.00%		90.91%	100.00%

	Planning-Based		Learning-Based	
	Conv	Exec	Conv	Exec
AVG	46.6	19600	57.3	17,337
STD	32.0	17221.4	51.9	11853.4
LoS	10	11	20	22
%LoS	90.91%	100.00%	90.91%	100.00%

Table 8.5: Overall Convergence and Execution Results.

Convergence			
	Tasks/VI-Conv	Tasks/Q-Conv	Tasks/Sarsa-Conv
Correlation	0.805	0.824	0.785
t-value	-1.8388	-2.14806	-1.68944
p-value	0.081628	0.044818	0.107478
Execution			
	Tasks/VI-Exec	Tasks/Q-Exec	Tasks/Sarsa-Exec
Correlation	-0.261	-0.600	-0.578
t-value	-3.77034	-4.51468	-5.01906
p-value	0.000601	0.000106	0.000033

Table 8.6: Convergence and Execution Correlation with Tasks Results.

8.2 Discussion

In the next paragraphs, we discuss the results and possible rejections of null hypotheses.

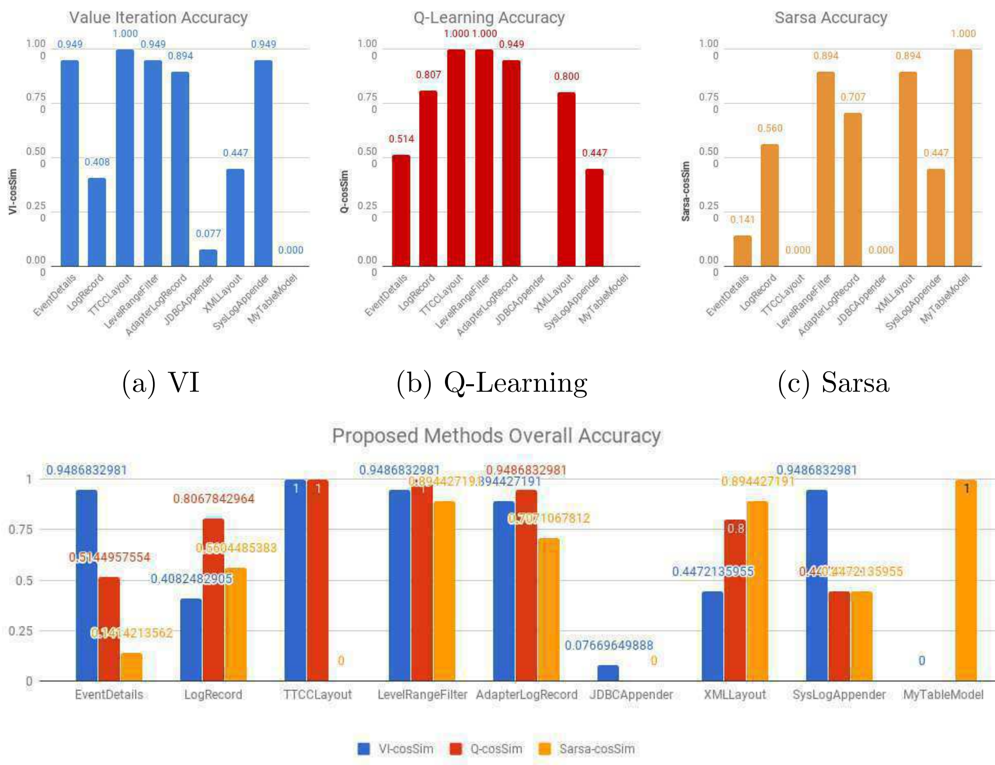
8.2.1 H1 and Accuracy Implications of Proposed Methods

In the first hypothesis, we are examining the radius of similarity between the proposed method and optimal solution (manual refactoring). Summary of the results is shown in Table 8.2.

Value Iteration algorithm has, on average, .63 similarity with the optimal solution. Five out of its nine case studies have above .5 similarity levels. Q-learning achieved .79 with six case studies with above .5 similarity. Sarsa had .52 with five out of nine case studies with above .5 similarity level. The highest results of accuracy are achieved with Q-learning, but its LoS is the lowest (7 out of 9, 63.64%). The highest results might be due to nature of Q-learning and calculation of improvement of the current state. From its six case studies with above .5, two of them have 1 similarity, one is .949, two of them are .8 and only one is with .514 similarity. These are remarkable results achieved in only 10 learning episodes. The lower value of LoS might undermine high accuracy results for Q-Learning, to reduce this effect, we want to compare the accuracy achieved with Q-Learning with other algorithms on the same case studies. To do that, we rank accuracy results for each case study based on algorithms. After the ranking, we take the average of these rankings algorithm-wise, the lowest average will point to the highest accuracy results (obviously pointing to the highest number of the best results, first positions). According to results, LoS had no effect on the overall accuracy of algorithms and Q-learning was again achieving the highest accuracy levels. The careful reader might point that relation between planning-based and learning-based comparison changed from slightly favoring learning-based to equal, and two methods are the same now. On the first look, this might be true observation, but if we further consider standard deviation that is higher in the planning-based algorithm, we are still in favor of learning-based algorithms and as conclusion no change on overall accuracy results. With these results, we reject the null-hypothesis and confirm its alternative hypothesis and state that proposed automated methods are able to meet manual refactoring in terms of number and kind of refactoring tasks.

ClassName	**VI**	**Q**	**Sarsa**
EventDetails	1	2	3
LogRecord	3	1	2
TTCCLayout	1	1	2
LevelRangeFilter	2	1	3
AdapterLogRecord	2	1	3
XMLLayout	3	2	1
SysLogAppender	1	2	2
Average (lower better)	1.86	1.43	2.29
Standard Deviation	0.90	0.53	0.76
	Planning-Based	**Learning-Based**	
Average	1.86	1.86	
Standard Deviation	0.90	0.77	

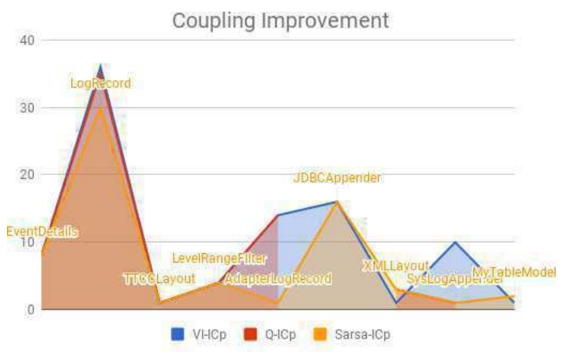
Table 8.7: Ranking Accuracy Levels of Algorithms Based on Successful Q-Learning Case Studies.



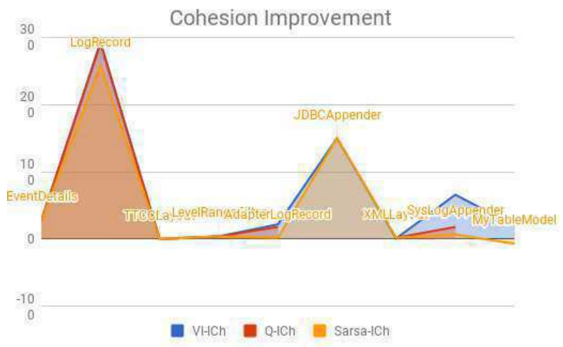
(a) VI (b) Q-Learning (c) Sarsa

(d) Overall

Figure 8.2: Histogram of Accuracy of Proposed Methods.



(a) Coupling

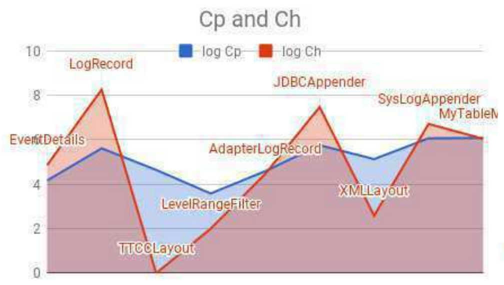


(b) Cohesion

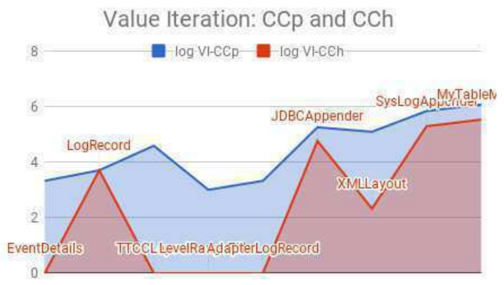
Figure 8.3: Coupling and Cohesion Improvements.

8.2.2 H2 and Quality Measures Improvement of Proposed Methods

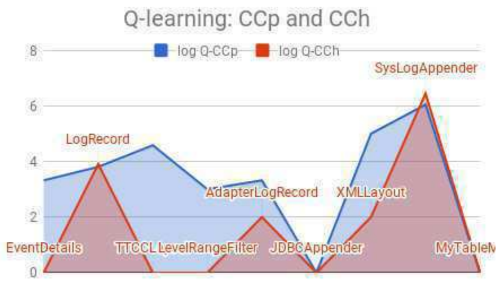
According to Table 8.7 majority of values achieved after applied methods are improved compared to its starting values. In Figure 8.3 we see, from the results plotted, to what extent each class improved. While the majority of classes were improved, Sarsa failed in the improvement of cohesion in MyTableModel class. Improvement ratio of 49:1 give us high confidence to reject null-hypothesis and state that proposed methods are able to improve coupling and cohesion on class-level.



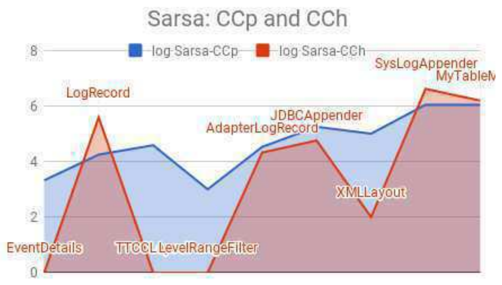
(a) Default



(b) VI

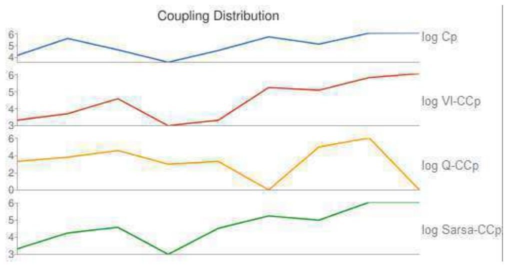


(c) Q-Learning

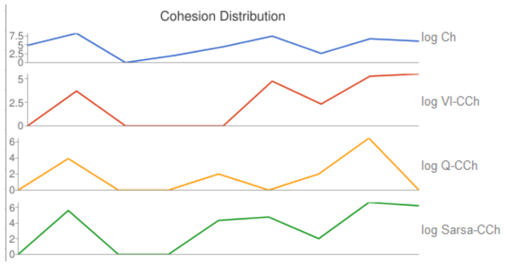


(d) Sarsa

Figure 8.4: Distribution of Coupling and Cohesion Before and After Proposed Methods.



(a) Coupling



(b) Cohesion

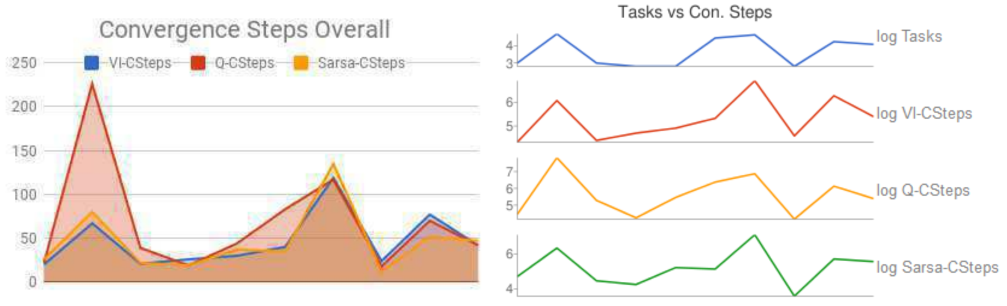
Figure 8.5: Comparison of Coupling and Cohesion Distributions.

8.2.3 H3 and Convergences of Proposed Methods

After the accuracy and improvement discussion of proposed method, next, we are interested in the efficiency of automated approaches. As defined, the first efficiency is related to the convergence steps of the exact solution of proposed methods. From Table 8.5 we can see that all methods require, on average, not more than 70 steps to converge. Sarsa achieved the best results requiring the lowest number of steps on average to converge (46.5). Value Iteration was the next with the average of 46.6. Q-learning requires the highest number of steps to converge, on average (68). When the algorithms are analyzed based on their category, the planning-based algorithm performs better then learning-based algorithms with, on average, 46.6 and 57.3 steps, respectively. These results are confirming our null hypothesis and we state that planning-based RL algorithm has faster converge rate comparing to learning-based algorithms.

Level of Success (LoS) is very high for all algorithms (the lowest is 90.91%). The only failed case study to converge is LogBrokerMonitor. Unfortunately, this is the largest class (in terms of LOC and number of tasks) in our experiments conducted and it might point us that MDP environment limitation for agent spending only sixty seconds in the environment might not be suitable for extra large classes to converge to an exact solution. As a part of future work, we might concentrate more on examining the relationship between extremely large classes and terminal function (the time required for the agent to converge) in MDP environment.

After the rejection of the null hypothesis, we want to see is there any correlation between the number of tasks and convergence steps. In particular, this is very important for possible extension of the automated framework and what we might expect if more number of tasks is to be considered. To quantify this relation, we first need to address tasks and what they represent. If we take summation of all tasks (MM, DM, EM) this might point us to the wrong conclusion, especially when there is a high overlapping rate between MM and DM refactoring tasks. To avoid misleading and overlapping tasks, we consider the maximum number of tasks of single kind of refactoring in analysis part of the automated approach appropriate representation. For example, if the analysis found: 10MM, 7DM, and 1 EM, our number of tasks is a 10 since it is the maximum value of single kind of refactoring tasks. In Table 8.5, the third column depicts tasks value for each case study. Pearson correlation [81] is used to calculate the correlation between tasks and convergence steps. Furthermore, the t-test is performed with the calculation of p-value for each test. From the results (Table 8.6), we can see that there is a high positive linear correlation between convergence steps and the number of tasks. At p-value 0.05, the results are significant only for Q-learning convergence steps. Other p values are not greater than .11 and if we relax our significance level till .1 or .15, which is reasonable with the limited number of case studies, all results of correlation become significant. This leads us to the conclusion that increasing the number of tasks will increase the number of steps required for convergence, hence increasing time and space complexity of automated framework.

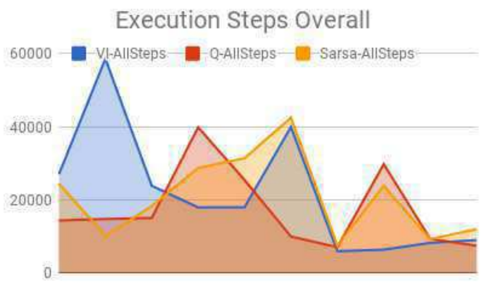


(a) Convergence Steps (b) Tasks and Convergence Steps

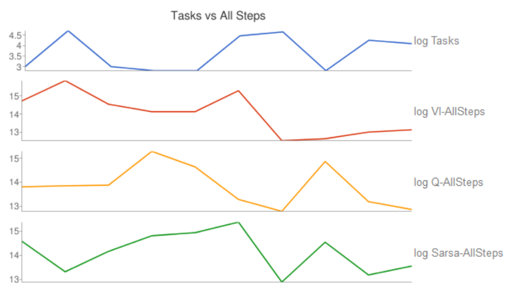
Figure 8.6: Convergence Rates.

8.2.4 H4 and Execution Rates of Proposed Methods

Our last hypothesis is related to the execution rates and the way of testing it is very similar to the previous hypothesis except that instead of convergence steps, we consider execution steps. On the contrary to the results of convergence steps, Value Iteration required the highest number of steps to finish execution. Q-learning was with the lowest number of execution steps (Table 8.5). All algorithms were fully successful in finishing execution (LoS = 100%). If we categorize the results to planning and learning-based, learning-based methods outperform planning-based method (17.337 vs 19.600 required steps, on average). With these results, we reject the null hypothesis and confirm that planning-based RL algorithm requires more number of steps for complete execution comparing to learning-based algorithms. Next to rejection of the null hypothesis, the relationship between tasks and execution steps is analyzed in the same manner as with convergence steps (Table 8.6) Execution steps have a negative linear correlation with the tasks. In the case of Q-Learning and Sarsa, this negative correlation is relatively high compared to low negative correlation in Value Iteration. All results are statistically significant



(a) Execution Steps



(b) Tasks and Execution Steps

Figure 8.7: Execution Rates.

(at p < .05). The results might seem surprising knowing that convergence steps had opposite effects, but they are very reasonable considering the limitation of time for agent and nature of real-time refactoring. We have to remember that agent has limited time to spend in the environment (sixty seconds) and real-time refactoring requires a time for each existed task to be executed. This reduces the number of effective steps and takes more time for the agent to execute actions. More tasks mean more time to spend on the execution of refactoring decreasing the number of "empty" steps from the agent leading to a negative linear correlation between tasks and execution steps. More real refactorings to sequence lead to more convergence steps resulting in the less number of overall execution steps.

8.2.5 Algorithms' Implications on Results

Empirical evaluation of the proposed methods has shown that *RefMark* automated framework, in general, is efficient with reliable accuracy results. Next to the hypothesis testing, we are further interested in the relationship between results and particular algorithm used. For that purpose, we summarize and rank all results based on algorithms employed (Table 8.8 and Table 8.9, respectively).

Value Iteration

Planning-based RL method has shown notable results in improvement of coupling and cohesion outperforming other methods used. Its accuracy of 63% is acceptable and might represent to what extent the model provided and its transition probabilities meet the reality of case studies. LoS was the highest achievable in the improvement and efficiency parts. VI was the second, after Sarsa, in the convergence efficiency rate and the last in the execution efficiency rate. VI was the best in overall ranking representing the best choice for overall performance. VI is the best choice also for improvement, and this has to be taken into account in the development cycle. If the goal of developer team is to have the highest improvement in quality measures regardless of its precision and accuracy, VI is the best choice.

Q-Learning

The first learning-based RL method used shows extremely beneficial in high accurate refactoring sequencing. The q-learning agent was able to learn environment in 10 learning episodes and achieve the accuracy of 78.82%. This might be due to the nature of Q-Learning update step of Q-value function where it takes the maximum expectation of the improvement of all next states. As an offline method, this might seem very expensive but it paid off in accuracy results achieved (later, we will see that Sarsa did not reach this accuracy level with less expensive update step of Q-values). On the overall, Q-Learning had the second best performance, after VI. On the accuracy alone, Q-Learning represents the best choice and this has to be taken into account when maintaining the phase of the software development requires the highest accuracy results regardless of the price in terms of time and space.

Sarsa

Second learning-based RL method used achieved the best results in efficiency (convergence steps). For the improvement rate and accuracy, Sarsa has shown poor performance. Even with 50 episodes of learning (Q-Learning had only 10 episodes), Sarsa was not able to catch good model of the environment and achieve more accurate results with higher improvements. This might be also due to its nature as an online method and its update of Q-values based on any improvement of the next state. Besides efficiency, comparing to Q-Learning, Sarsa dominates only in LoS of improvement part, where it was able to succeed in 81.82% of the time (Q-Learning succeeded in 63.64% of the time). Any further usage of Sarsa in refactoring sequencing requires relaxation of a number of learning episodes allowing the agent to better learn environment. On overall ranking, Sarsa was the worst choice.

	VI		Q-Learning		Sarsa	
Accuracy	63.03%		78.82%		51.61%	
Average Improvement	CCp	CCh	CCp	CCh	CCp	CCh
	10.1	64.60	9.40	51.60	7.30	49.30
%LoS	81.82%		63.64%		81.82%	
Efficiency	Conv	Exec	Conv	Exec	Conv	Exec
	46.6	19600	68	15746.5	46.5	18927.5
%LoS	90.91%	100.00%	90.91%	100.00%	90.91%	100.00%

Table 8.8: Summary of All Results Based on Algorithms Used.

	VI		Q-Learning		Sarsa		Precedence Results	
Accuracy	2		1		3		Q-VI-Sarsa	
Improvement	CCp	CCh	CCp	CCh	CCp	CCh	Coupling	Cohesion
	1	1	2	2	3	3	VI-Q-Sarsa	VI-Q-Sarsa
%LoS	1		2		1		VI/Sarsa-Q	
Efficiency	Conv	Exec	Conv	Exec	Conv	Exec	Convergence	Execution
	2	3	3	1	1	2	Sarsa-VI-Q	Q-Sarsa-VI
%LoS	1	1	1	1	1	1	NoDiff	
	1s	5	**1s**	4	**1s**	4		
	2s	2	**2s**	3	**2s**	1		
	3s	1	**3s**	1	**3s**	3		
Overall Rank	1st		2nd		3rd			
	Planning-Based		Learning-Based					
	1s	5	**1s**		4			
	2s	2	**2s**		2			
	3s	1	**3s**		2			
Overall Rank	1st		2nd					

Table 8.9: Overall Methods Ranking Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	LogRecord	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		2	1	1	6Ef,1MC			49		306	
Automated Refactoring	**Analysis**	26	22	5f	-						
	VI	0	22	0		67	58549	13		13	
	Q Learning	10	14	0	2XMM,3XDM	226	14780	14	3	15	1
	Sarsa	2	10	0	6XMM,5XDM	80	10159	19	3	48	1

Table 8.10: LogRecord Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	TTCCLayout	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		3	0	0	0			25		0	
Automated Refactoring	**Analysis**	8	6	0	-						
	VI	1	0	0	4XMM,1XDM	21	23848	24	9	0	4
	Q Learning	1	0	0	5XMM,2XDM	39	15051	24	9	0	4
	Sarsa	0	1	0	3XMM,3XDM	22	18284	24	9	0	4

Table 8.11: TTCCLayout Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	LevelRangeFilter	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		0	5	0	0			12		4	
Automated Refactoring	**Analysis**	7	6	0	-						
	VI	1	3	0	3XDM	26	17940	8	4	0	1
	Q Learning	0	3	0	1XMM,3XDM	19	39803	8	4	0	1
	Sarsa	1	2	0	2XMM,2XDM	19	28697	8	4	0	1

Table 8.12: LevelRangeFilter Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	AdapterLogRecord	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		5	5	0	0	10		24		22	
Automated Refactoring	**Analysis**	7	6	3f		7+7+7					
	VI	1	3	0	2XMM	30	17981	10	7	0	3
	Q Learning	1	2	0	3XMM;2XDM	44	25443	10	7	4	3
	Sarsa	0	1	0	1XMM,5XDM	37	31373	23	7	20	3

Table 8.13: AdapterLogRecord Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	JDBCAppender	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		4	0	0				54		177	
Automated Refactoring	**Analysis**	22	13	0							
	VI	1	13	0		40	39920	38		27	
	Q Learning	0	0	0	8XMM,12XDM	83	9981		7		6
	Sarsa	0	13	0	0	35	42471	38		27	

Table 8.14: JDBCAppender Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	LogBrokerMonitor	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		28	0	0	1Df,1EC,3Ef			327		4796	
Automated Refactoring	**Analysis**	103	19	39f							
	VI	0	0	0	58XMM,17XDM	-	711	-	7	-	15
	Q Learning	0	1	0	29XMM,22XDM	-	122	326	164	4697	1290
	Sarsa	0	0	0	25XMM,28XDM	-	135	-	165	-	1239

Table 8.15: LogBrokerMonitor Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	Hierarchy	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		1	16	0				74		241	
Automated Refactoring	**Analysis**	25	5	10f							
	VI	0	0	0	17XMM,5XDM	119	5968	-	27	-	190
	Q Learning	0	0	0	15XMM,7XDM	117	7087	-	27	-	190
	Sarsa	0	0	0	16XMM,7XDM	135	7551	-	27	-	190

Table 8.16: Hierarchy Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	XMLLayout	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		1	2	0				35		6	
Automated Refactoring	**Analysis**	7	4	0							
	VI	1	0	0	2XMM,2XDM	24	6406	34	7	5	6
	Q Learning	2	1	0	1XMM,2XDM	18	29765	32	7	4	6
	Sarsa	0	4	0	1XMM	12	23813	32	7	4	6

Table 8.17: XMLLayout Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	SysLogAppender	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		4	2	0	2Ttl			67		105	
Automated Refactoring	**Analysis**	19	8	12f							
	VI	3	3	0	8XMM,2XDM	77	8253	57	7	39	15
	Q Learning	0	1	0	7XMM,9XDM	70	9361	66	7	87	15
	Sarsa	0	1	0	12XMM,4XDM	52	9270	66	7	98	15

Table 8.18: SysLogAppender Results.

		Refactoring Tasks				Steps		Coupling		Cohesion	
Class Name:	MyTableModel	**MM**	**DM**	**EM**	**Other**	**Conv**	**All**	**Corr.**	**End**	**Corr.**	**End**
Manual Refactoring		0	7	0				68		66	
	Analysis	17	12	3f							
	VI	1	0	0	10XMM,5XDM	42	9014	67	17	46	8
	Q Learning	0	0	0	5XMM,10XDM	42	7467	-	17	-	8
Automated Refactoring	**Sarsa**	0	1	0	11XMM,4XDM	47	11981	66	17	73	8

Table 8.19: MyTableModel Results.

CHAPTER 9

CONCLUSION AND FUTURE WORK

With this chapter, we conclude this thesis and summarize what has been done to meet proposed objectives, contributions, threats to validity and possible extension and future work directions.

9.1 Summary

This thesis represents a research on refactoring and reinforcement learning methods combined in an automated framework that resulted in the *RefMark* plugin. Some refactoring tasks (Extract Method, Move Method, Delete Method) were used as a driving force to improve coupling and cohesion values on a class-level. Coupling and cohesion values of the class were used as leading factors representing the current state of the system. Model environment, as an MDP, was an input for RL-based algorithms. Designed as a 2d map with coupling and cohesion values as coordinate points, the environment became suitable MDP instance. Aforementioned tasks were used as actions in this environment and transition model (probabilities) is defined based on an initial experience and possible distribution between tasks. Model environment (states), actions and transitions were used to solve MDP instance with VI, Q-Learning and Sarsa algorithms giving state-value and Q-value function for each state and policy to follow if the desired effect of coupling and cohesion improvement was a goal to achieve. Incorporating discounted rewards to better represents desirability of the current effects of tasks over effects of the future tasks, next to stochastic environment, the solution of VI, Q-learning and Sarsa algorithms is represented. The proposed method and these algorithms are integrated into automated framework *RefMark* representing a complete solution for an automated refactoring approach. As a prototype, automated framework shows potential usability for a further research. *RefMark* uses concrete refactoring tasks as actions making the system interacts directly with the source code. The initial prototype was developed considering all possible refactoring tasks. After its initial tests and high complexity in time and space requirements, optimized version added additional analysis layer that filtered error-prone refactoring tasks and proceeded with reliable, error-free, refactoring tasks. The optimized version of *RefMark* was empirically evaluated and its accuracy, improvement ratios, and efficiency have shown promising results. *RefMark* potential place, in a real software development cycle, can be seen as a supportive tool where developers want to quantify to which extent they can improve certain parts of the software quality measures using the limited number of tasks in a limited time frame.

9.2 Threats to Validity

This thesis is an attempt to look at a refactoring prioritization problem from the other than its common SE perspective. This new perspective (reinforcement learning) has well-established theory and background and well-known successful application in different fields. Matching SE representation and this perspective in a successful story is not an easy task and many factors are playing the key roles in its success. As a first attempt, with time-constrained thesis work, there are some threats that might undermine our work. In the next paragraphs we discuss identified threats and what we did to minimize its possible negative effects.

- Threats to *construct* validity explain what is the magnitude of theory and observation in our empirical study.
  - The main threat to construct validity in our work is related to a software quality measures used for a coupling and cohesion. Are RFC and LCOM good indicators of coupling and cohesion on a class-level as a structural software metrics? Will their semantic counterparts (for instance, CCBC and CE) lead to the same results? These questions might be properly answered after the same framework is tested with these different quality measures. With the limited scope, we tried our

    best to concentrate on coupling and cohesion metrics that are well-understood, with its "pros and cons", and yet have stable and consistent tool for faster calculation avoiding additional time complexity to our framework. C&K [65] metric suit was the only candidate that fits these conditions.

- Threats to *conclusion* validity define the magnitude between the way of treatment of information and its outcome.

  – The first threat to conclusion validity is related to behavior preservation of software projects used for refactoring. We solely depend on Eclipse internal checks for refactoring execution. This might not be enough to preserve the correct semantic behavior of the project. While more appropriate way might be to provide the complete set of unit tests to cover complete project behavior, for our automated framework this was not feasible at this stage of development. We are exploring techniques that optimize coupling and cohesion on the class-level following automated approach based on reinforcement learning using real-time refactoring. Having execution of the complete set of unit tests after each refactoring task will simply make complete approach impractical and unfeasible. For this stage of development of the automated framework, Eclipse implementations of checking refactoring conditions with additional analysis layer of *RefMark* were tolerable elements. For future, unit tests might be employed for complete system behavior preservation checks.

  - The second threat to conclusion validity might be related to values used for the cost of refactoring, discount factor and number of iterations in solving MDP instance. To minimize this effect, we were trying to follow common MDP values (for example, .9 for discount factor) and the common ratio between the goal state and reward (cost of refactoring tasks). The common ratio between the goal state and reward was given after rationalization of a number of tasks used and the size of the environment. We were trying to balance by penalizing execution of the refactoring agent in the proper way for the agent to motivate it to sequence refactoring tasks while still maintaining its desire to reach the goal. If a penalty was more severe for the agent, it might not try to sequence refactoring tasks and it would prefer to stay in place, simply, for an agent, moving will cost more than it might get reaching the goal state. A number of iterations were chosen by trial-and-error with concern to allow enough time for the agent and the algorithm to converge while still have reasonable time execution.

- Threats to *internal* validity explain possible circumstances that could influence our observation.

  - As a prototype, there are many threats to internal validity, starting to beginning decisions to follow Eclipse platform and its JDT library for a Java. As an open source, community-driven, platform with a good, but not perfect, documentation and support, our plugin implementation depends on different external plugins that are developed on different Eclipse versions. The first threat can be seen that these different plugins (for UI, for example) may not behave the same on different Eclipse distributions due to different dependencies, changed Eclipse source, etc. We tried to minimize this threat by building our solutions on a well-known and stable dependency that reached maturity and are well-accepted in the Eclipse community. For example, the interface is built based on standard SWT Eclipse plugin, metrics are calculated using matured ckjm software metrics suite.

- Threats to *external* validity explain possible generalization concerns of this research.
  - Due to nature of reinforcement learning, the choice to go with real refactoring tasks applied while iterating all possible paths, the main threat to external validity concerns the fact that the limited number of classes were used for testing proposed automated framework. With the limited timeline of thesis research, we tried to minimize this threat by testing automated framework on a different range of classes.

9.3 Contributions

- To the best of our knowledge, this is the first attempt to define refactoring automation problem as a Markov Decision Processes and Reinforcement Learning.
- Empirical evaluation of different RL-based solutions for refactoring MDP instance
- The main contribution of this thesis is in the proposed method and its materialization as a complete automated refactoring framework that is extensible and flexible for further advances.

9.4 Future Work

Possible future improvements can be divided into three directions:

1. Software Engineering (improvements related to general refactoring process):
    - First future consideration has to be related to the more rigorous assurance of semantic preservation of projects refactored. This might be achieved with a suitable set of tests. The more changeable part is how these tests, that cover complete system, can be efficiently incorporated in the automated *RefMark* tool while maintaining practicability and accepted time and space complexity. One of the ways might be to include these tests after generation of refactoring tasks' pools and to further

filter only these refactoring tasks based on semantic preservation. This would generate new pools of refactoring tasks that preserve semantic behavior.

- One of the possible future paths can be an extension of the current approach to simulation-based refactoring automation. This might open a possibility for larger projects but also might force us to a trade-off between real-time refactoring benefits (high accuracy, minimized noise) and simulation-based fast automation.
- Detection of bad smells related to quality measures incorporated in a current tool might be our future focus. Moreover, its detection of these bad smells are metric-based, this might be even easier to follow since current tool already employs metric suite for quality measures calculation.
- Limiting the number of refactoring tasks to prioritize can also be considered as a future improvement.
- Another, possible contradicting, software quality measures might be used for optimization. For example, we might address refactoring prioritization improving reusability and maintainability while maintaining optimal coupling and cohesion ratio.
- Package-based and system-based improvements might be more appropriate than current class-level optimization. This might be our future direction.

2. Development

   - *RefMark* is developed as an Eclipse plugin, our future direction might be to transfer *RefMark* plugin to Eclipse RCP application, providing great benefits of the fully-packaged standalone application without constraints and additional requirements that current plugin-based solution requires.

3. Reinforcement Learning

   - Reformulating refactoring automation process as a partially-observable MDP (POMDP) instance might be our next future direction. POMDP instance requires another set of algorithms to be employed for a solution but might more precisely address nature of new developers in software teams, with less knowledge about software projects maintained.
   - If the number of refactoring tasks are to be limited, MDP instance has to be considered with a finite horizon, requiring different algorithms to be used for the appropriate solution. This might be other future directions and might represent addressing real situation were software projects have limited time frame for the maintenance phase. The limited time frame might also be addressed, as mentioned earlier, with adjusting discount factor, but focusing on a limited number of refactoring tasks might better address refactoring from SE point of view.
   - Terminal function of refactoring MDP instance might be relaxed more

to address extremely large classes. Furthermore, more detailed analysis of the relation between the time allocated for an agent to spend in the environment and the size of the class might be performed.

Appendix

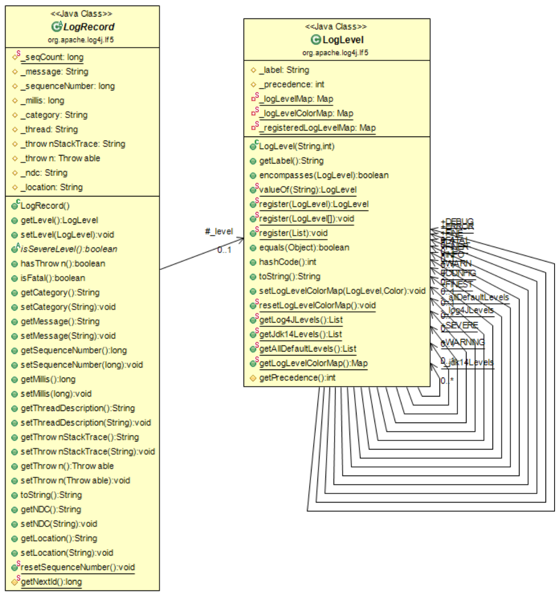


Figure 9.1: LogRecord Class Diagram.

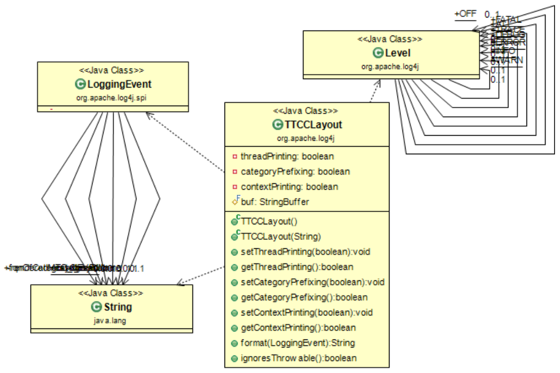


Figure 9.2: TTCCLayout Class Diagram.

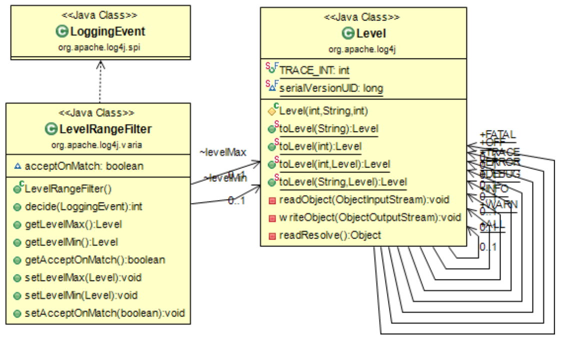


Figure 9.3: LevelRangeFilter Class Diagram.

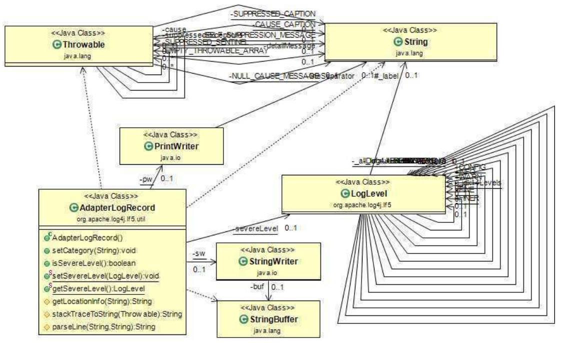


Figure 9.4: AdapterLogRecord Class Diagram.

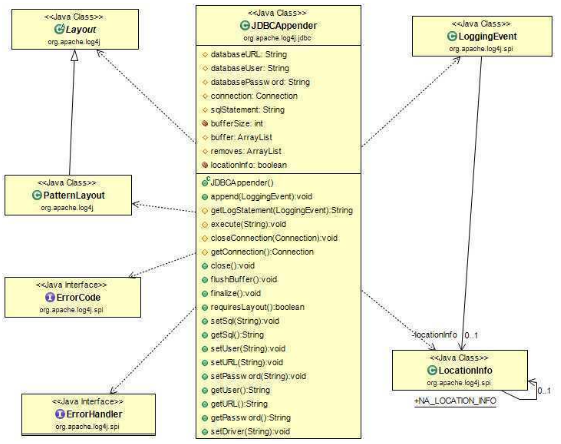


Figure 9.5: JDBCAppender Class Diagram.

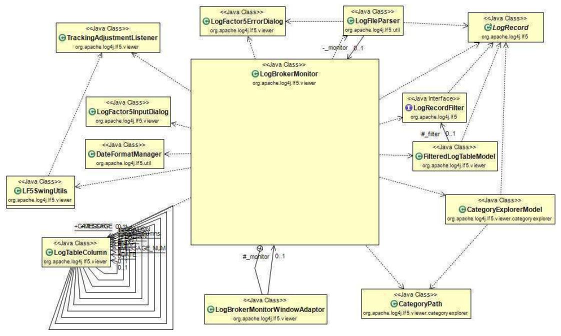


Figure 9.6: LogBrokerMonitor Class Diagram.

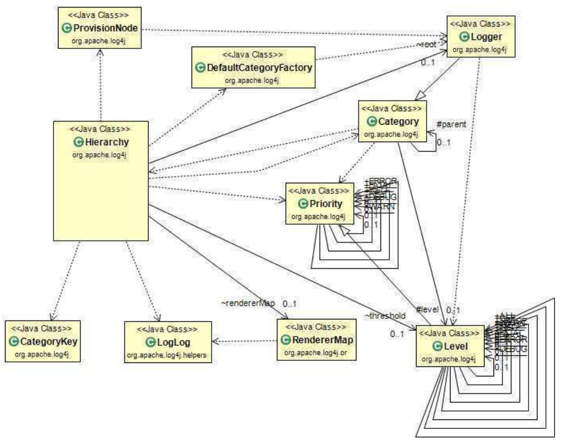


Figure 9.7: Hierarchy Class Diagram.

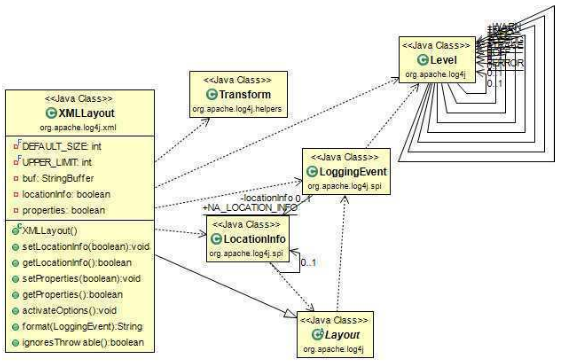


Figure 9.8: XMLLayout Class Diagram.

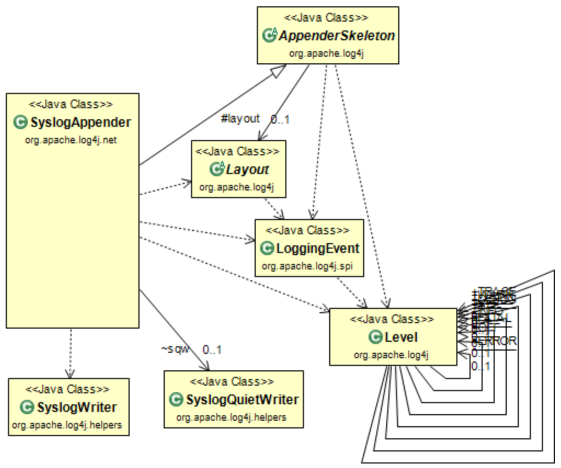
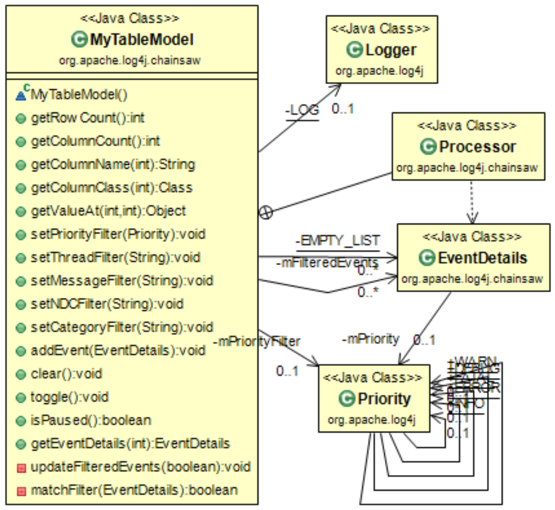


Figure 9.9: SysLogAppender Class Diagram.



<<Java Class>>
MyTableModel
org.apache.log4j.chainsaw
MyTableModel()
getRowCount():int
getColumnCount():int
getColumnName(int):String
getColumnClass(int):Class
getValueAt(int,int):Object
setPriorityFilter(Priority):void
setThreadFilter(String):void
setMessageFilter(String):void
setNDCFilter(String):void
setCategoryFilter(String):void
addEvent(EventDetails):void
clear():void
toggle():void
isPaused():boolean
getEventDetails(int):EventDetails
updateFilteredEvents(boolean):void
matchFilter(EventDetails):boolean
<<Java Class>>
Logger
org.apache.log4j
-LOG
0..1
<<Java Class>>
Processor
org.apache.log4j.chainsaw
-EMPTY_LIST
-mFilteredEvents
0..*
0..*
<<Java Class>>
EventDetails
org.apache.log4j.chainsaw
-mPriorityFilter
0..1
-mPriority
0..1
<<Java Class>>
Priority
org.apache.log4j

Figure 9.10: MyTableModel Class Diagram.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: improving the design of existing code. 1999," *ISBN: 0-201-48567-2*.

[2] G. H. Pinto and F. Kamei, "What programmers say about refactoring tools?: An empirical investigation of stack overflow," in *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*. ACM, 2013, pp. 33–36.

[3] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Journal*, vol. 23, no. 2, pp. 323–361, 2015.

[4] L. Martin, A. Giesl, and J. Martin, "Dynamic component program visualization," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 289–298.

[5] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[6] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on.* IEEE, 2004, pp. 350–359.

[7] W. H. Brown, R. C. Malveau, and T. J. Mowbray, “Antipatterns: refactoring software, architectures, and projects in crisis,” 1998.

[8] T. M. T. T. F. Munoz, “Beyond the refactoring browser: Advanced tool support for software refactoring,” 2003.

[9] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, “A bayesian approach for the detection of code and design smells,” in *Quality Software, 2009. QSIC’09. 9th International Conference on.* IEEE, 2009, pp. 305–314.

[10] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, “Decor: A method for the specification and detection of code and design smells,” *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010.

[11] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, “Facilitating software refactoring with appropriate resolution order of bad smells,” in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* ACM, 2009, pp. 265–268.

[12] M. Kessentini, S. Vaucher, and H. Sahraoui, “Deviance from perfection is a better criterion than closeness to evil when identifying risky code,” in *Pro-*

ceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 2010, pp. 113–122.

[13] D. I. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, 2013.

[14] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[15] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos, "An empirical investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, no. 2, pp. 127–139, 2003.

[16] W. F. Opdyke, "Refactoring: A program restructuring aid in designing object-oriented application frameworks," Ph.D. dissertation, PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 1998.

[18] S. Zhioua and Q. Ebec, "Stochastic systems divergence through reinforcement learning," Ph.D. dissertation, Université Laval, 2008.

[19] B. H. Abed-alguni, "Action-selection method for reinforcement learning based on cuckoo search algorithm," *Arabian Journal for Science and Engineering*, pp. 1–15, 2017.

[20] G. Rasool and Z. Arshad, “A review of code smell mining techniques,” *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.

[21] J. Al Dallal, “Identifying refactoring opportunities in object-oriented code: A systematic literature review,” *Information and software Technology*, vol. 58, pp. 231–249, 2015.

[22] M. Misbhauddin and M. Alshayeb, “Uml model refactoring: a systematic literature review,” *Empirical Software Engineering*, vol. 20, no. 1, pp. 206–251, 2015.

[23] S. Rochimah, S. Arifiani, and V. F. Insanittaqwa, “Non-source code refactoring: A systematic literature review,” *International Journal of Software Engineering and Its Applications*, vol. 9, no. 6, pp. 197–214, 2015.

[24] M. Abebe and C.-J. Yoo, “Trends, opportunities and challenges of software refactoring: A systematic literature review,” *structure*, vol. 8, no. 6, 2014.

[25] G. Vale, E. Figueiredo, R. Abilio, and H. Costa, “Bad smells in software product lines: A systematic review,” in *Software Components, Architectures and Reuse (SBCARS), 2014 Eighth Brazilian Symposium on.* IEEE, 2014, pp. 84–94.

[26] M. Orrú, S. Porru, M. Marchesi, and R. Tonelli, “The evolution of knowledge in the refactoring research field,” in *Scientific Workshop Proceedings of the XP2015.* ACM, 2015, p. 10.

[27] T. Mens and T. Tourwé, “A survey of software refactoring,” *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 126–139, 2004.

[28] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, “Systematic literature reviews in software engineering–a systematic literature review,” *Information and software technology*, vol. 51, no. 1, pp. 7–15, 2009.

[29] R. Wangberg, “A literature review on code smells and refactoring,” Ph.D. dissertation, University of Oslo, 2010.

[30] M. Zhang, T. Hall, and N. Baddoo, “Code bad smells: a review of current knowledge,” *Journal of Software Maintenance and Evolution: research and practice*, vol. 23, no. 3, pp. 179–202, 2011.

[31] H. Liu, G. Li, Z. Ma, and W. Shao, “Scheduling of conflicting refactorings to promote quality improvement,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* ACM, 2007, pp. 489–492.

[32] C. Chisalita-Cretu, “The entity refactoring set selection problem-practical experiments for an evolutionary approach,” in *Proceedings of the World Congress on Engineering and Computer Science*, vol. 1. Citeseer, 2009.

[33] H. Liu, Z. Ma, W. Shao, and Z. Niu, “Schedule of bad smell detection and resolution: A new way to save effort,” *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 220–235, 2012.

[34] P. Meananeatra, “Identifying refactoring sequences for improving software maintainability,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2012, pp. 406–409.

[35] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, “Maintainability defects detection and correction: a multi-objective approach,” *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[36] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, 2002.

[37] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, “Recommendation system for software refactoring using innovization and interactive dynamic optimization,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering.* ACM, 2014a, pp. 331–336.

[38] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, “Design defects detection and correction by example,” in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on.* IEEE, 2011, pp. 81–90.

[39] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó’Cinnéide, and K. Deb, “Software refactoring under uncertainty: a robust multi-objective approach,”

in *Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion.* ACM, 2014b, pp. 187–188.

[40] H.-G. Beyer and B. Sendhoff, "Robust optimization–a comprehensive survey," *Computer methods in applied mechanics and engineering*, vol. 196, no. 33, pp. 3190–3218, 2007.

[41] A. Ouni, "A mono-and multi-objective approach for recommending software refactoring," Ph.D. dissertation, 2015.

[42] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of classes for refactoring: A step towards improvement in software quality," in *Proceedings of the Third International Symposium on Women in Computing and Informatics.* ACM, 2015, pp. 228–234.

[43] L. Kumar and A. Sureka, "Application of lssvm and smote on seven open source projects for predicting refactoring at class level," 2017.

[44] I. Kádár, P. Hegedus, R. Ferenc, and T. Gyimóthy, "A code refactoring dataset and its assessment regarding software maintainability," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 599–603.

[45] A. Murgia, R. Tonelli, M. Marchesi, G. Concas, S. Counsell, J. McFall, and S. Swift, "Refactoring and its relationship with fan-in and fan-out: An empirical study," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on.* IEEE, 2012, pp. 63–72.

[46] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Software Maintenance, 2002. Proceedings. International Conference on.* IEEE, 2002, pp. 576–585.

[47] JDeodorant. (2017, Oct) Jdeodorant. [Online]. Available: https://users.encs.concordia.ca/~nikolaos/jdeodorant/

[48] PMD. (2017, oct) Pmd. [Online]. Available: https://pmd.github.io/

[49] (2017) Checkstyle. [Online]. Available: http://checkstyle.sourceforge.net/

[50] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media, 2007.

[51] R. for Industry. (2013) Intooitus software assessment tools. [Online]. Available: http://www.researchforindustry.eu/story/5

[52] G. Ganea, I. Verebi, and R. Marinescu, "Continuous quality assessment with incode," *Science of Computer Programming*, 2015.

[53] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, "Jspirit: a flexible tool for the analysis of code smells," in *2015 34th International Conference of the Chilean Computer Science Society (SCCC).* IEEE, 2015, pp. 1–6.

[54] S. Slinger, "Code smell detection in eclipse," *Delft University of Technology*, 2005.

[55] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th international symposium on Software visualization.* ACM, 2010, pp. 5–14.

[56] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on.* IEEE, 2002, pp. 97–106.

[57] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *Reverse engineering (WCRE), 2012 19th working conference on.* IEEE, 2012, pp. 466–475.

[58] K. Nongpong, "Integrating" code smells" detection with refactoring tool support," Ph.D. dissertation, The University of Wisconsin-Milwaukee, 2012.

[59] V. Ferme, "Jcodeodor: A software quality advisor through design flaws detection," *Master's thesis, University of Milano-Bicocca, Milano, Italy*, 2013.

[60] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on.* IEEE, 2012, pp. 662–665.

[61] P. Danphitsanuphan and T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in *Engineering and Technology (S-CET), 2012 Spring Congress on.* IEEE, 2012, pp. 1–5.

[62] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, “Building empirical support for automated code smell detection,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.* ACM, 2010, p. 8.

[63] J. Tappolet, C. Kiefer, and A. Bernstein, “Semantic web enabled software analysis,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 8, no. 2, pp. 225–240, 2010.

[64] R. Wieman, “Anti-pattern scanner: An approach to detect anti-patterns and design violations,” 2011.

[65] D. Spinellis, “Tool writing: a forgotten art?(software tools),” *IEEE Software*, vol. 22, no. 4, pp. 9–11, 2005.

[66] E. Kristiansen, “Automated composition of refactorings,” Master’s thesis, 2014.

[67] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, “Design patterns: Elements of reusable object-oriented software,” *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.

[68] E. Murphy-Hill and A. P. Black, “Refactoring tools: Fitness for purpose,” *Software, IEEE*, vol. 25, no. 5, pp. 38–44, 2008.

[69] A. M. Eilertsen, “Making software refactoring safer,” Ph.D. dissertation, Masters thesis, Department of Informatics, University of Bergen, 2016.

[70] A. M. Eilertsen, A. H. Bagge, and V. Stolz, "Safer refactorings," in *International Symposium on Leveraging Applications of Formal Methods.* Springer, 2016, pp. 517–531.

[71] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to advanced empirical software engineering.* Springer, 2008, pp. 201–228.

[72] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering.* ACM, 2010, pp. 371–372.

[73] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Software Maintenance (ICSM), 2010 IEEE International Conference on.* IEEE, 2010, pp. 1–10.

[74] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ACM, 2015, pp. 179–190.

[75] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[76] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo, "An empirical exploration of the distributions of the chidamber and kemerer object-oriented

metrics suite," *Empirical Software Engineering*, vol. 10, no. 1, pp. 81–104, 2005.

[77] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[78] N. I. Churcher, M. J. Shepperd, S. Chidamber, and C. Kemerer, "Comments on" a metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 263–265, 1995.

[79] M. Hitz and B. Montazeri, "Chidamber and kemerer's metrics suite: a measurement theory perspective," *IEEE Transactions on software Engineering*, vol. 22, no. 4, pp. 267–271, 1996.

[80] W. Li, "Another metric suite for object-oriented programming," *Journal of Systems and Software*, vol. 44, no. 2, pp. 155–162, 1998.

[81] K. Pearson, "Note on regression and inheritance in the case of two parents," *Proceedings of the Royal Society of London*, vol. 58, pp. 240–242, 1895.

Vitae

- Name: Armin Kobilica
- Nationality: Bosnian
- Date of Birth: 24.01.1990
- Email: *arminkobilica@gmail.com*
- Permanent Address: Sehida bb, 72240 Kakanj, Bosnia and Herzegovina
- Work Experience: Before joining to the graduate studies at KFUPM, I have worked as a developer in a telecommunication company and as a lab teaching assistant at university (C++ and Advanced Java courses). During my graduate studies at KFUPM, I have worked as teaching assistant (AI and Machine Learning courses), data and system analyst, and as IT Support Staff.
- Research Interests: My main interests are related to an application of Statistical Learning Theory, AI, and Reinforcement Learning.