Expert Systems with Applications 42 (2015) 1652–1664

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

AutoRefactoring: A platform to build refactoring agents

Baldoino Fonseca dos Santos Neto [a,*], Márcio Ribeiro [a], Viviane Torres da Silva [b], Christiano Braga [b], Carlos José Pereira de Lucena [c], Evandro de Barros Costa [a]

[a] Federal University of Alagoas, Maceió, AL, Brazil
[b] Fluminense Federal University, Niteroi, RJ, Brazil
[c] Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil

ARTICLE INFO

Article history:
Available online 28 September 2014

Keywords:
Autonomous agents
Software refactoring
Smells detection
Smells correction
Code quality

ABSTRACT

Software maintenance may degrade the software quality. One of the primary ways to reduce undesired effects of maintenance is refactoring, which is a technique to improve software code quality without changing its observable behavior. To safely apply a refactoring, several issues must be considered: (i) identify the code parts that should be improved; (ii) determine the changes that must be applied to the code in order to improve its; (iii) evaluate the corrections impacts on code quality; and (iv) check that the observable behavior of the software will be preserved after applying the corrections. Given the amount of issues to consider, refactoring by hand has been assumed to be an expensive and error-prone task. Therefore, in this paper, we propose an agent-based platform that enables to implement an agent able to autonomously deal with the above mentioned refactoring issues. To evaluate our approach, we performed an empirical study on code smells detection and correction, code quality improvement and preservation of the software observable behavior. To answer our research questions, we analyze 5 releases of Java open source projects, ranging from 166 to 711 classes.

1. Introduction

As software ages, there is a need to maintain it to reflect evolving user requirements and also to correct detected errors. Maintenance is a complex and time-consuming task, costing around 40% to 75% of the total cost of the software (Brown, Malveau, Brown, McCormick, & Mowbray, 1998; Foster, 1993). If the developers simply perform the changes in order to reflect new requirements but do not consider the design of the software code, the code tends to degrade (Bertran, Arcoverde, Garcia, Chavez, & von Staa, 2012; van Gurp, Brinkkemper, & Bosch, 2005). The cumulative effects of such changes can lead to software that are unreliable, difficult to reason on, and difficult to change (Fowler, Beck, Brant, Opdyke, & Roberts, 1999).

One of the primary ways to maintain software is *Refactoring*, which is a technique to improve software code quality without changing its observable behavior (Fowler et al., 1999). Empirical studies related to the use of refactoring have found that it can improve maintainability (Kolb, Muthing, Patzke, & Yamauchi, 2005; Moser, Sillitti, Abrahamsson, & Succi, 2006). Not only does existing work suggest that refactoring is useful and important, but it also suggests that refactoring is a frequent practice (Murphy-Hill et al., 2011).

In this work we consider that, in order to apply a refactoring, four main requirements must be fulfilled (Mens & Tourwé, 2004): (i) code smells identification: the identification of the software code parts that should be improved (by following Fowler et al. (1999) approach); code smells identification: the identification of the software code parts that should be improved; (ii) corrections determination: to determine the changes (or corrections) that must be applied to the code in order to improve its quality (e. g. *reusability*,[1] *flexibility*,[2] *extendibility*,[3] *effectiveness*[4]); (iii) quality evaluation: to evaluate the impact on the software code quality as a result of applying the corrections; and (iv) observable behavior

* Corresponding author.
E-mail addresses: baldoino@ic.ufal.br (B.F.d. Santos Neto), marcio@ic.ufal.br (M. Ribeiro), viviane.silva@ic.uff.br (V.T.d. Silva), cbraga@ic.uff.br (C. Braga), lucena@inf.puc-rio.br (C.J.P. de Lucena), evandro@ic.ufal.br (E. de Barros Costa).

[1] *Reusability* is the degree to which a software module can be used in more than one computer program (Raed, Li, Swain, & Newman, 2011).
[2] *Flexibility* is the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed (Raed et al., 2011).
[3] *Extendibility* is the ease with which a system or component can be modified to increase its storage or functional capacity (Raed et al., 2011).
[4] *Effectiveness* is the degree to which a design is able to achieve the desired functionality and behavior using object oriented design techniques (Raed et al., 2011).

http://dx.doi.org/10.1016/j.eswa.2014.09.022
0957-4174/

preservation: to guarantee that the observable behavior of the software will be preserved after applying the corrections. Given the amount of issues to be considered, refactoring by hand has been assumed to be error-prone and extremely expensive (Brown et al., 1998; Foster, 1993; Ge, Dubose, & Murphy-hill, 2012; Murphy-Hill et al., 2011). Therefore, given the benefits that refactoring has on maintainability, there is a need for solutions that can help developers to perform refactoring activities in an autonomous way, i.e., to automatically and independently decide whether a correction should be applied or not in order to improve software code quality without changing its observable behavior.

Several techniques and tools have been proposed to deal with specific refactoring requirements. In Marinescu (2004), Fontana, Braione, and Zanoni (2012), Padilha, Pereira, Figueiredo, Almeida, and Garcia (2014), Tourwe and Mens (2003), Kim, Kim, and Seu (2013), Moha, Gueheneuc, Duchien, and Meur (2010), Guéhéneuc and Sahraoui (2011), Ouni, Kessentini, Sahraoui, and Boukadoum (2012), Kessentini, Kessentini, and Erradi (2011) and Palomba et al. (2013) the authors propose the identification of code smells by the application of different techniques based on declarative rules specification (Fontana et al., 2012; Kim et al., 2013; Marinescu, 2004; Moha et al., 2010; Padilha et al., 2014; Tourwe & Mens, 2003), software change history (Palomba et al., 2013), bayesian belief networks (Guéhéneuc & Sahraoui, 2011) or genetic programming (Kessentini et al., 2011; Ouni et al., 2012). In Fowler et al. (1999), Ouni et al. (2012), Kessentini et al. (2011), Sahraoui, Godin, and Miceli (2000), Liu, Yang, Niu, Ma, and Shao (2009), Seng, Stammel, and Burkhart (2006), Harman and Tratt (2007), O'Keeffe and Cinnéide (2008), Ouni, Kessentini, and Sahraoui (2013) and Ouni, Kessentini, Sahraoui, and Hamdi (2013) the authors deal with the correction of code smells by using general solutions that can be applied in manual way (Fowler et al., 1999; Liu et al., 2009; Sahraoui et al., 2000) or search-based techniques (Harman & Tratt, 2007; Kessentini et al., 2011; O'Keeffe & Cinnéide, 2008; Ouni et al., 2012, 2013; Ouni, Kessentini, Sahraoui, & Hamdi, 2013; Seng et al., 2006). Approaches to evaluate the software code quality after applying corrections are described in Raed et al. (2011), Kataoka, Imai, and Andou (2002) and Alshayeb (2011). Moreover, recent works (Daniel, Dig, Garcia, & Marinov, 2007; Pacheco, Lahiri, Ernst, & Ball, 2007; Schafer, Ekman, & Moor, 2010; Soares, Gheyi, Serey, & Massoni, 2010, 2011, 2013a) have been concerned with the preservation of the software observable behavior after applying corrections.

However, two main aspects, common to all approaches, are unsatisfactory: (i) each proposal deals with just some of the above mentioned refactoring requirements, that is, code smell identification, corrections determination, quality improvement, or observable behavior preservation; and (ii) the approaches focus on the correction of the code smells without concerning with the improvement of software code quality, which is in fact the main goal of refactoring. Studies (Raed et al., 2011) demonstrate that not all code smell corrections improve the code quality. Some of the corrections tend to degrade the software code. Therefore, it is necessary solutions able to evaluate the impacts on the code quality before deciding to perform the corrections.

In order to solve the above mentioned unsatisfactory aspects in the state-of the-art of code refactoring, we propose an agent-based platform, called *AutoRefactoring* platform, that enables the development of an agent to autonomously perform refactoring. The proposed platform copes with the above mentioned requirements by focusing on the improvement of the quality of a given software code. Our agent-based platform is implemented as an extension of the *Jason* platform (Bordini, Hubner, & Wooldridge, 2007) with refactoring operations. The resulting agent is able to autonomously: (i) analyze the software code and recognize smells by using different existing detection techniques and tools (Fontana et al., 2012; Guéhéneuc & Sahraoui, 2011; Kessentini et al., 2011; Kim et al., 2013; Marinescu, 2004; Moha et al., 2010; Ouni et al., 2012; Padilha et al., 2014; Palomba et al., 2013; Tourwe & Mens, 2003); (ii) reason on the impact of the corrections on the software code quality in order to determine the ones to be applied and the order of their application; and, (iii) check if the software observable behavior will be preserved after performing the corrections.

We have applied our approach in releases of five Java open source projects, namely Log4j,[5] SweetHome3D,[6] HSQLDB,[7] jEdit[8] and Xerces.[9] Our aim was to refactor such projects in order to improve the quality of their code. We have analyzed such projects with respect to code smells detection and correction, software quality improvement and preservation of the software observable behavior. These projects were selected since different code smells could be detected indicating that the quality of their codes are very different, they have been used in other refactoring approaches (Kessentini et al., 2011; Ouni et al., 2012; Soares, Gheyi, Murphy-hill, & Johnson, 2013b) and they are used in practice. In our experiment, our platform was able to correct up to 80.56% of the detected code smells while evolving the code quality up to 70.54% and preserve the software observable behavior. When comparing the results of our platform with other approaches we have notices that they have frequently corrected more code smells than ours. However, the refactorings performed by our platform have always made more improvements in the quality of the projects, what is the main goal of performed refactoring. Our platform prioritizes the code quality and does not correct the smells that decrease the code quality (as demonstrated in Section 7.1.5). In summary, the main contributions of this article are:

- An agent-based platform to implement an agent able to autonomously perform refactoring (Sections 4 and 5);
- An evaluation of the refactoring platform regarding code smells detection and correction, software code quality improvement and verification preservation of the software observable behavior (Sections 6 and 7).

We organize the remainder of this article as follows. In Section 3, we show an example of software refactoring that motivates our study. Then, in Section 4, we describe how developers specify the code smells to be detected and the way each one must be corrected and in Section 5 we present our platform to perform refactoring. Afterwards, we present the experiment settings in Section 6, and in Section 7 we discuss the results of the experiment. We discuss related work in Section 2 and concluding remarks in Section 8.

2. Related work

There are several proposals that have recently focused on specific refactoring issues, such as, detection and correction of code smells, evaluation of the refactored code quality or preservation of the software observable behavior after performing corrections. These proposals range from automatic to guided manual approaches. Nevertheless, there are very few contributions focused on a solution able to provide support to autonomously deal with the above mentioned refactoring issues. As of today, the literature can be classified into four broad categories: smells detection, smells correction, software code quality evaluation and observable behavior preservation.

In the first category, Marinescu (2004), Fontana et al. (2012), Padilha et al. (2014), Tourwe and Mens (2003), Kim et al. (2013) propose techniques that rely on declarative rule specification to identify parts of the software that need to be improved. In these

[5] Log4j. http://logging.apache.org/log4j/1.2/.
[6] SweetHome3D. http://www.sweethome3d.com.
[7] HSQLDB. http://sourceforge.net/projects/hsqldb/.
[8] jEdit. http://www.jedit.org.
[9] Xerces. http://xerces.apache.org.

settings, rules are manually defined to identify the key symptoms that characterize a smell. These symptoms are described using quantitative metrics, structural, and/or lexical information. In Ouni et al. (2012), Kessentini et al. (2011) the authors propose a technique that, instead of manually specifying the rules for detecting each code smell type, such rules are extracted from instances of maintainability smells by using *Genetic Programming*. The use of Bayesian Belief Networks (BBNs) has been proposed in Guéhéneuc and Sahraoui (2011) to support uncertainty in smell detection, where BBNs present two main benefits: they can work with missing data and can be tuned with analysts' knowledge. Palomba et al. (2013) proposes a technique, named Historical Information for Smell deTection (HIST), to detect source code smells based on change history information extracted from versioning systems. The results shown in Palomba et al. (2013) indicate that HIST is able to identify code smells that cannot be identified through other approaches based on code analysis.

Finally, Fontana et al. (2012) presents a survey of the existing tools for automatic code smell detection, and, comparatively, analyzes some of them against their usefulness in assisting the main stakeholders of the software development process when assessing the code quality of a software. In particular, Fontana et al. (2012) analyzes the tools Checkstyle, DECOR, inCode, inFusion, JDeodorant, PMD and Stench Blossom in order to investigate its ability to expose the regions of code which are most affected by structural decay, and the relevance of tool responses with respect to future software evolution. The authors conclude that such tools are certainly useful to assess which parts of the code need to be improved, but they are not able to determine which is the best one. As discussed in Section 5.2.2 our platform does not provide any technique to detect smells, however it is able to use the above mentioned techniques and tools. In order to do so, its just necessary to define prohibition norms which are able to represent the kind of smells to be detected by such techniques and tools.

In the second category, Fowler et al. (1999), Sahraoui et al. (2000) and Liu et al. (2009) propose "standard" correction solutions that can be applied by hand for each kind of smell. However, it is difficult to prove or ensure the generality of these solutions to all smells. In fact, the same smell type can have different possible corrective solutions. Automated approaches are used in the majority of existing works (Harman & Tratt, 2007; Kessentini et al., 2011; O'Keeffe & Cinnéide, 2008; Seng et al., 2006) that formulates the correction problem as an single-objective optimization problem in order to find the best sequence of corrections to minimize the number of smells. An evolution of these works is proposed in Ouni et al. (2012) that uses a multi-objective optimization approach to find the best sequence of corrections that minimizes the number of smells and changes applied to the code in order to execute the sequence of corrections. Ouni et al. (2012) evolved its work by finding the best sequence of corrections that maximizes the use of refactoring applied in the past to similar contexts, and minimizes the number of code smells and changes applied to the software, as described in Ouni et al. (2013) and Ouni, Kessentini, Sahraoui, and Hamdi (2013). The main drawback of the afore mentioned correction approaches is that they consider that the quality of the software can be improved just decreasing the number of existing smells. However, as discussed in Section 7.1.3, and described in studies such as Raed et al. (2011), not all correction increase the software code quality. Therefore, our platform presents a better performance since its main goal is to increase the software code quality.

The third category concerns about how to evaluate the software code quality evolution resulting from the application of corrections. In this context, Kataoka et al. (2002) proposes a quantitative evaluation method to measure the maintainability enhancement effect of smells correction by using the coupling metrics to evaluate the correction effect. The objective of Alshayeb (2011) is to quantitatively assess, using software metrics, the effect of corrections on different quality attributes to decide whether the cost and time in correcting are worthwhile. In Raed et al. (2011), the effects of the corrections described in the catalog (Fowler et al., 1999) are evaluated by considering four software quality factors: reusability, flexibility, extendibility and effectiveness. The authors found that not all refactoring activities improve the quality factors. They established correction heuristics that can help developers in following a goal-oriented refactoring process, i.e., developers can use particular corrections to improve particular quality. We adopt the quality model proposed in Raed et al. (2011) in our platform since it considers quality factors with high impact on the software code and it presents a detailed description of each impact that enables the automation of the quality evaluation.

At last but not least, the fourth category concerns the preservation of the software observable behavior after performing corrections. Several approaches (Daniel et al., 2007; Pacheco et al., 2007; Schafer et al., 2010; Soares et al., 2010, Soares, Mongiovi, & Gheyi, 2011, 2013a) have been proposed to deal with such issue. We have incorporated the work described in Soares, Gheyi, and Massoni (2013a) in our approach because it is the most recent one and it presents the more relevant results when compared with Daniel et al. (2007), Pacheco et al. (2007), Schafer et al. (2010), Soares et al. (2010) and Soares et al. (2011). The main novelties described in Soares et al. (2013a) are its technique for generating input programs and its test oracles for checking behavioral preservation based on dynamic analysis. It performs four major steps. Fist, a program generator automatically yields programs as test inputs for a refactoring. Second, the refactoring under test is automatically applied to each generated program. The transformation is evaluated by test oracles, in terms of overly weak and overly strong preconditions. In the end, it may have detected a number of failures.

3. Motivational example

Let's consider the classes *Person* and *Company*, shown in Fig. 1, we can observe some code smells in these classes, such as.

3.1. Public Fields

All classes have *public fields*, which may break *encapsulation*[10] of a class. A recent study (Raed et al., 2011) demonstrates that the decrease of the *encapsulation* level has negative impact on the *flexibility* and *effectiveness* of the code.

Such code smells can be solved by applying the *Encapsulate Field* correction, which safely changes the visibility of the fields from *public* to *private*, and creates *get* and *set* methods for each *public* field. The application of this correction to the classes shown in Fig. 1 generates the ones shown in Fig. 2.

We observe that the application of the *Encapsulate Fields* increases the encapsulation, since it changes the public fields to private, and the *messaging*,[11] since it creates new get and set methods, therefore enabling a greater reusability of the software, as described in Raed et al. (2011).

3.2. Data Clumps

The classes *Person* and *Company* have a couple of duplicated fields: *street*, *city*, *state* and *postalcode*. This smell is known as *Data Clumps*. It occurs, for instance, when we have similar fields in different classes (Fowler et al., 1999).

[10] *Encapsulation* is the ratio of private fields to the total number of fields (Raed et al., 2011).

[11] *Messaging* is a count of public methods in a class (Raed et al., 2011).

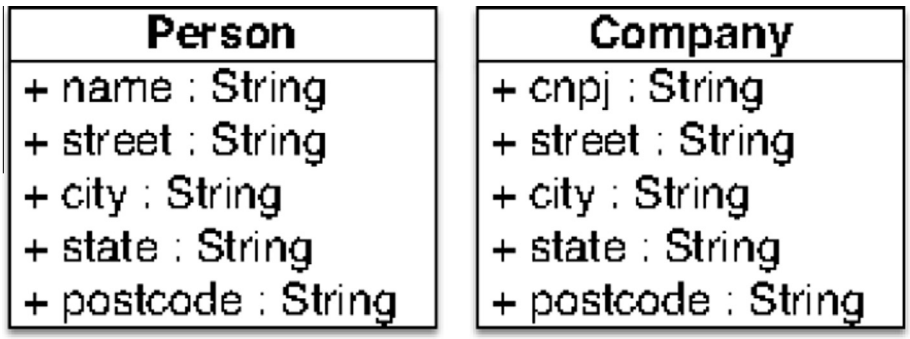

Fig. 1. The Person and Company classes.

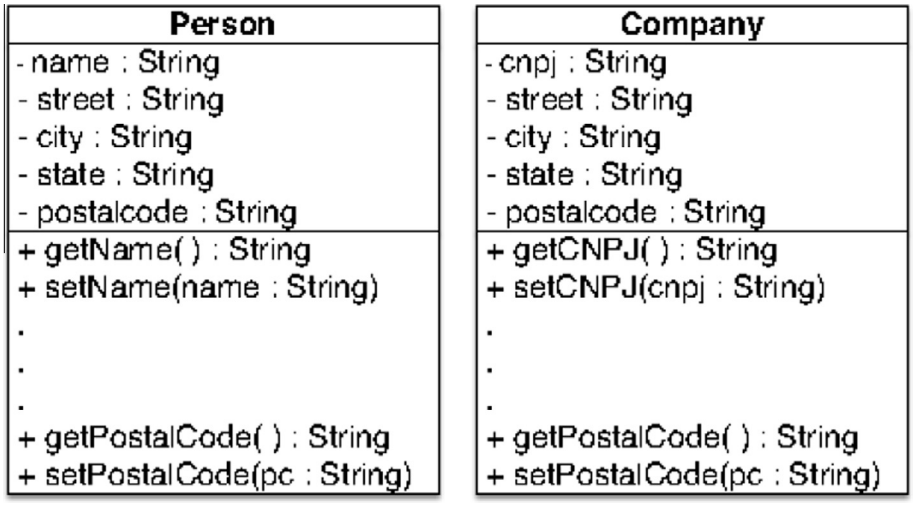

Fig. 2. Encapsulate Fields application to the original classes.

Data Clumps can be corrected by the application of the *Extract Class* correction, which creates a new class and move the clumps from the old classes into the new class. The result of the application of such correction to the classes *Person* and *Company* is shown in Fig. 3.

Even though the application of the *Extract Class* correction has decreased the code duplication, it increases the size of the code since it creates new classes. Therefore, increasing the software complexity, as described in Raed et al. (2011).

In a large software, it may be difficult to answer the following questions (Fontana et al., 2012; Liu et al., 2009; Mens & Tourwé, 2004): (i) how can we identify these code smells? (ii) which of these corrections must be applied in order to increase the software code quality and in which order they must be implemented? (iii) how can we check if the software observable behavior will be preserved after performing the corrections?

These questions are the driving force of our research and are being answer through Sections 4 and 5. In Section 4 we detail how we are able to describe the kinds of code smells to be automatically detected by the agents of our platforms and the corrections that such agents can automatically apply when the code smells are detected. In Section 5 we present our *AutoRefactoring* platform and in Sections 6 and 7 we describe the experiment we have conducted to evaluate the platform.

4. Specifying smells and its corrections

In his book named *Refactoring*, Fowler et al. (1999) presents a catalog of code smells and the corrections that should be applied in each case. The author also discusses about the impact of the corrections on the software code quality. Since our goal is to implement agents to autonomously perform refactoring, there is a need for mechanisms that enable developers to specify the code smells that must be avoided and the corrections that can be applied. In addition, it is also important to be able to describe the benefits and consequences of applying such corrections. The agent instantiated by our platform is able to follow such specifications when investigating the occurrence of code smells and the applicability of the corrections.

We cope with such issues by using *norms* (da Silva, 2008). The *norms* will describe (i) the code smells that must be avoided (as prohibitions) or the corrections that must be applied (as obligations); (ii) the circumstances that the agent is prohibited to have a code smells or the circumstances where the agent is obliged to apply a correction; and (iii) the impact on the software code quality for having a code smell or applying a correction. In case of code smells that have different implementations, in order to find out all such implementations, is necessary to define one norm for each implementation. In such a case, there will be several norms dedicated to the same code smell.

According to da Silva (2008), *norms* regulate the behavior of agents by obligating or prohibiting them to achieve a specific environment state. A *norm* (typically) describes (i) the environment state that an agent must achieve or the state that the agent must avoid; (ii) the circumstance when the norm is active and should be followed by the agent; (iii) the benefits of fulfilling the norm; and also (iv) the consequences of violating it. Following such idea, we adopt the norm description below:

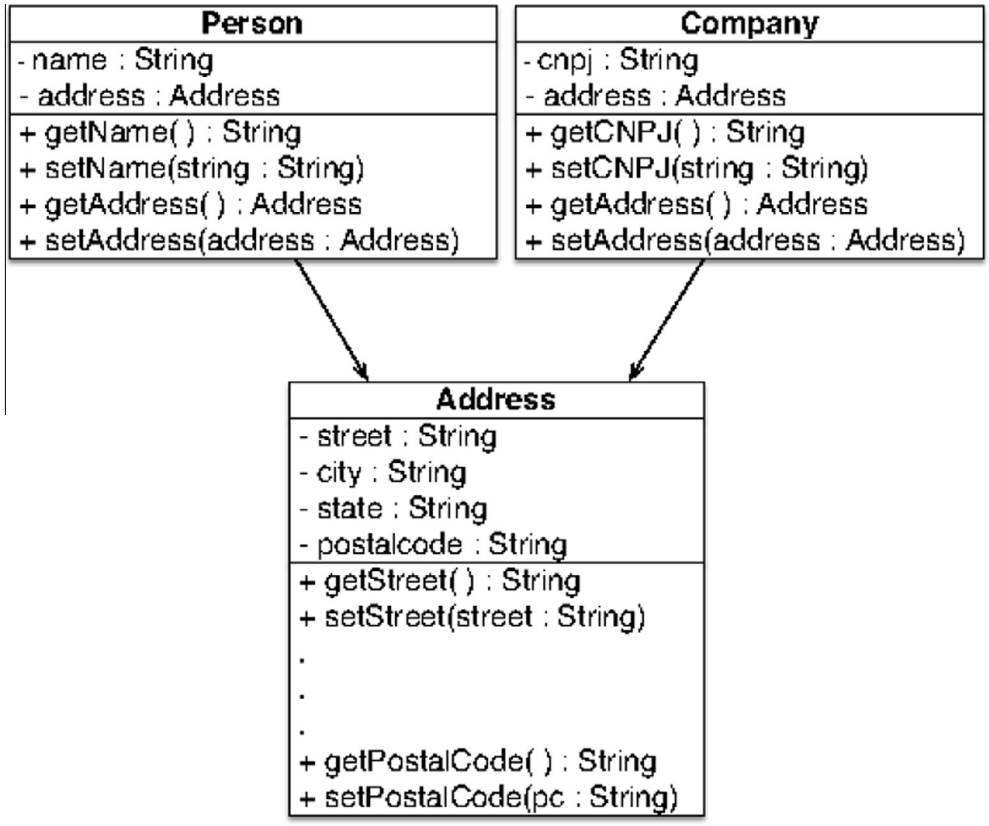

Fig. 3. *Extract Class* application to the original classes.

norm(*deontic*, *circumstance*, *state*, *impact*)

where *deontic* is used to inform if the norm is a *prohibition* or an *obligation*; *state* is the situation to be avoided or achieved (i.e., the situation that indicates the code smell that should be avoided or the situation that indicates the correction that should be applied); *circumstance* defines when the norm is active (i.e., when a code smell is identified or when a correction must be applied); and *impact* describes the benefits and consequences of having a smell or applying a correction.

The norm *impact* is measured according to the improvement or not on the software code quality. In this work, we adopt the quality model proposed in Raed et al. (2011), which describes how to measure norm *impact* according to 4 factors: *reusability*, *flexibility*, *extendibility* and *effectiveness*. The approach (Raed et al., 2011) proposes index calculations to evaluate each quality factor based on different software metrics (Lanza & Marinescu, 2006). Table 1, extracted from Raed et al. (2011), depicts such calculations.

The definition of a norm (prohibition or obligation) does not depend on the project itself but on the code smell the developer wants identify and on the quality factor one wants to improve on each project. Therefore, a given norm can be reused in different projects if the developer's goals in all projects are the same. For instance, in our experiment, we have used the same set of norms in all four projects since our goal was to improve the same quality factors in all of them. If the code smell to be detected and the quality factors to be improved are different then different norms must be manually defined, it does not matter if they are to be applied on a single or many different projects.

In order to exemplify the use of norms, let's consider the example presented in Section 3 and describe a norm related to the code smell *Public Fields*, named *publicFields*, and another related to the correction used to solve this smell, named *encapsulateFields*. The norm *publicFields* states that situations where the number of public fields of a class X is greater than 0 are always prohibited. As discussed in the motivational example, the consequence of not correcting such smell (and violating the *publicFields* norm) is the decrease of the code *encapsulation* level, which impacts negatively the quality factors *flexibility* and *effectiveness*.

publicFields (prohibition,
exist (PublicFields),
PublicFields $>$ 0,
[decrease (flexibility),
decrease (effectiveness)])

The norm *encapsulateFields* states that the correction *Encapsulate Fields* must be applied when the norm *publicFields* is violated. According to Section 3, only two quality indexes, namely *encapsulation* and *messaging*, are affected by norm *encapsulatesField*. Theses indexes affect three quality factors, namely *reusability* (influenced by the *messaging* index), *flexibility* (influenced by the *encapsulation* index) and *effectiveness* (also influenced by the *encapsulation* index). Therefore, we are able to conclude that the norm *encapsulatesFields* impacts positively on the quality factors *flexibility*, *effectiveness* and *reusability*.

encapsulateFields (obligation,
violated (publicFields, PublicFields),
apply (encapsulateFields, PublicFields),
[increase (flexibility),
increase (effectiveness),
increase (reusability)])

The corrections defined by the obligation norms are applied to the code by the refactoring agent through the execution of a set of actions that changes the software code. Corrections are thus

Table 1
Quality model.

Quality factors	Quality index calculation
Reusability	−0.25∗Coupling + 0.25∗Cohesion + 0.5∗Messaging + 0.5∗Design Size
Flexibility	0.25∗Encapsulation − 0.25∗Coupling + 0.5∗Composition + 0.5∗Polymorphism
Extendibility	0.5∗Abstraction − 0.5∗Coupling + 0.5∗Inheritance + 0.5∗Polymorphism
Effectiveness	0.2∗Abstraction + 0.2∗Encapsulation + 0.2∗Composition + 0.2∗Inheritance + 0.2∗Polymorphism

Table 2
Quality measures.

Before applying refactoring	After applying refactoring
Reusability = 0.5∗messaging = 0	Reusability = 0.5∗(16/16) = 0.5
Flexibility = 0.25∗encapsulation = 0	Flexibility = 0.25∗(8/8) = 0.25
Extendibility = –	*Extendibility* = –
Effectiveness = 0.2∗encapsulation = 0	Effectiveness = 0.2∗(8/8) = 0.2

related to sequences of actions known in the literature as *plans* Bratman (1999). Each *plan* describes the actions that must be executed in order to apply a correction.

In the proposed platform, a *plan* has three components (Rao, 1996): (*Invocation condition*) that describes the correction that must be applied in order to fix a code smell; (*Context*) that describes the condition that must be satisfied before executing the plan; and, (*Body*) that states a sequence of actions that apply the correction related to the plan.

For example, let's consider that the obligation norm *encapsulateFields* has been activated. Therefore, a plan to perform such correction must be retrieved. Such plan has an invocation condition representing such correction, a context that verifies if the software can be changed and a body composed of actions that change the fields from public to private and actions that create *get* and *set* methods, as described below.

Invocation Condition : apply (encapsulateFields, PublicFields)
Context : permission (change, true)
Body : .changeToPrivate (PublicFields);
.createGetsSets (PublicFields).

In order to confirm that correction is increasing the software code quality, the software is measured before and after applying a correction. Table 2 shows the measures related to each quality factor calculated before and after applying the correction. We can conclude that the software quality will be improved if the correction is applied. Note that we have not measured the extendibility of the code since it is not modified due to the application of *Encapsulate Field*.

In order to use our platform, presented in Section 5, it is necessary to define (i) the norms that will help the agent on the detection of code smells and on the determination of the corrections; (ii) the plans executed by the agent in order to apply the corrections. Since the proposed platform is an extension of the *Jason* platform, where the agents are described by using a programming language based on a restricted first-order language, we use the same notation to represent the norms and plans.

5. Refactoring platform

The agent-based refactoring platform being proposed extends the *Jason* platform (Bordini et al., 2007) to provide support to the development of a refactoring agent. In Section 5.1 we briefly recall the main components of the *Jason* platform and in Section 5.2 we present our extended platform.

5.1. Jason platform

Jason (Fig. 4, reproduced from Bordini et al. (2007)) is a platform that gives support to the creation of *Belief-Desire-Intention* (BDI) agents (Rao & Georgeff, 1991). Sets (of *beliefs*, *events*, *plans* and *intentions*) are represented as bases and rectangles are used to represent some of the functions involved in the execution of BDI agents.

In a nutshell, the *Jason* platform works as follows. An agent perceives information coming from: (i) the environment, (ii) from the execution of its own plans and (iii) in the form of messages sent by other agents. The perceived information is the input to the *Beliefs Update* function that is responsible for reviewing the *Beliefs* base by taking into account the current perception and the beliefs already stored in the base. The *Beliefs Update* function is also responsible for updating the set of events to be carried on by the *Event Selection* function. A single event is selected in the *Event Selection* function that is unified with triggering events in the invocation conditions of the plans by the *Unify Event* function generating a set of all relevant plans. The context of each relevant plan is verified according to the *Beliefs* base by the *Check Context* function selecting a set of applicable plans.

Next, the *Plans Selection* function selects a single applicable plan from the set of plans, which becomes the intended means for handling the selected event. The *Intention Selection* function selects one of the agent's intentions that is executed by the *Execute Intention* function. When all formula in the body of a plan has been executed, the whole plan is removed from the intention list. This ends a cycle of execution, and *Jason* starts all over again, checking the state of the environment after agents have acted upon it, generating the relevant events, and so forth.

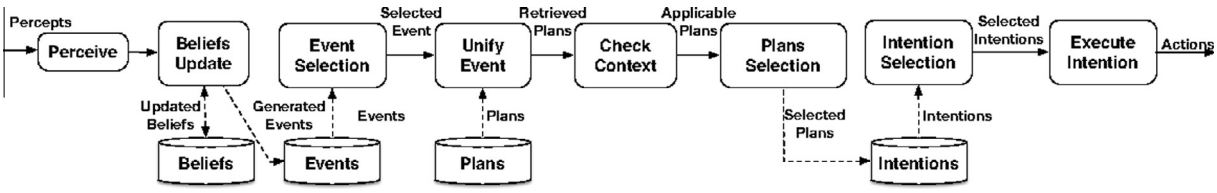

Fig. 4. *Jason* platform.

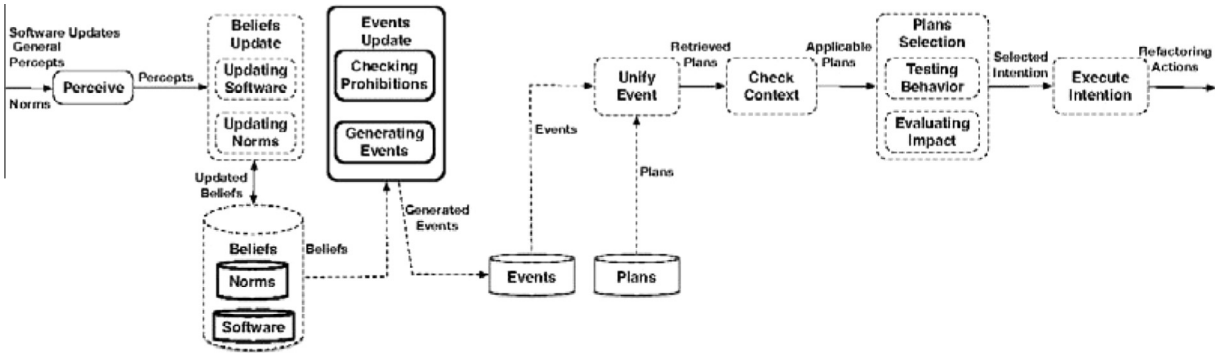

Fig. 5. *AutoRefactoring* platform.

5.2. AutoRefactoring platform

We have extended the *Jason* platform in order to be able to implement an agent that reasons about the refactoring requirements. The extended platform, called *AutoRefactoring* platform, is illustrated in Fig. 5. Four functions from the *Jason* platform were updated (*Beliefs Update*, *Unify Events*, *Plan Selection* and *Execute Intention*). A new function was added (*Events Update*) together with two new bases (*Software* and *Norms* bases). The added components are represented by bold lines in Fig. 5 and the updated components by dashed lines. In this section we will focus on discussing the updated and added functions.

5.2.1. Beliefs Update function

The *Beliefs Update* function updates the *Software* and the *Norms* bases by executing two new tasks. The *Updating Software* task uses version control systems (such as, git (Loelieger & McCullough, 2012) and svn (Pilato, Collings-Sussman, & Fitspatrick, 2009)) to update the current version of the software stored in the *Software* base. The *Updating Norms* task updates the set of norms stored in the *Norms* base according to the new norms specified by developers, as described in Section 4.

Example. In the scenario of the *Public Fields* refactoring, the perceived classes (*Person* and *Company*), shown in Fig. 1, are stored in the *Software* base by task *Updating Software*. In addition, if the norms *publicFields* and *encapsulateFields* are specified by a developer, and, furthermore, they do not exist in the *Norms* base, they are added by task *Updating Norms*.

5.2.2. Events Update function

The *Events Update* function, described in Algorithm 1, is responsible for two tasks. The *Checking Prohibitions* task identifies the existence of code smells by checking the prohibition norms have been activated and violated due to the updated *Beliefs* base (lines 2 to 8). As described in Section 4, prohibition norms regulate the existence of smells in the software code, therefore if the state regulated by a prohibition norm is satisfied, it indicates that such a norm was violated and a smell has been identified. The state regulated by prohibition norm is transmitted to existing code smells detection tools (Fontana et al., 2012) that verifies the state satisfaction.

The *Generating Events* task identifies the obligation norms whose circumstances have been satisfied and generates new events from the states regulated by the activated obligation norms (lines 9 to 13). Such events represent corrections to be applied in order to fix detected smells.

```
   Data: Events base represented by events
   Data: Beliefs base represented by beliefs
   Data: Norms base represented by norms
   Result: The Events base is updated
1  foreach norm n in norms do
2      if n is a PROHIBITION then
3          if The circumstance of n is satisfied due to the updated beliefs
           then
4              if The state regulated by n is confirmed by detection tools
               then
5                  A code smell has been identified;
6              end
7          end
8      end
9      if n is a OBLIGATION then
10         if The circumstance of n is satisfied due to the updated beliefs
           and identified code smells then
11             A new event is generated requesting the application of the
               correction regulated by the obligation norm;
12         end
13     end
14 end
```

Example. In the *Public Fields* refactoring example, the classes *Person* and *Company* have *public fields*, so the prohibition norm *publicFields* was violated. The obligation norm *encapsulateFields* is activated and a new event requesting the application of the correction *Encapsulate Fields* is generated.

5.2.3. Unify Events and Check Context

The *Unify Events* function retrieves all events and unifies them with the *invocation conditions* of the plans stored in the *Plans* base. Then, the *Check Context* function checks the plans whose context is satisfied; such plans are known as *applicable* plans. The Algorithm 2 describes this process.

Note that in *Jason* the events are retrieved by the *Event Selection* function and just one is selected and sent to the *Unify Event* function. Given that our refactoring platform will identify the best sequence of corrections to be applied (see the *Plan Selection* function in Section 5.2.4), it may not select just one event to be handled. Therefore, the *Event Selection* function has been removed and now the *Unify Event* function retrieves plans to handle all events stored in the *Events* base. The *Check Context* function works

as in Jason by checking if the context of each retrieved plan is according to the *Beliefs* base in order to select the applicable plans.

```
Data: Events base represented by events
Data: Beliefs base represented by beliefs
Data: Plans base represented by plans
Result: A set of applicable plans
foreach event e in events do
    foreach plan p in plans do
        if e can be unified with the plan invocation codition then
            if The p context is satisfied by the beliefs then
                Add p to the set of applicable plans
            end
        end
    end
end
```

Example. Let us consider that we have generated events representing the corrections *Encapsulate Fields* (that is used to fix the code smells *public fields*) and *Extract Class* (that is used to fix code smells *data clumps*) described in our motivational example. Then, a set of *applicable* plans to apply such corrections will be retrieved from the *Plans* base.

5.2.4. Plans Selection function

The *Plans Selection* function is responsible for selecting the sequence of corrections (or plans) that will provide the best code quality without changing the software observable behavior. In order to do so, it uses the search-based technique proposed by Ouni et al. (2012) to verify possible sequences of plans from the set of *applicable* ones. It discards the sequences that change the software observable behavior (see the *Testing Behavior* task), it evaluates the impact of each sequence of corrections on the code quality (see the *Evaluating Impact* task) and selects the sequence that provides the highest improvement on the software code quality.

5.2.4.1. Testing Behavior. The success of the application of a sequence of plans, representing the applications of corrections, is directly related to the fact that the observable behavior of the software must not change after applying the corrections. The *Testing Behavior* task uses the approach proposed in Soares et al. (2013a) where a set of tests is performed in order to check the behavior of the original software and the refactored one. If the execution of a sequence of plans does not preserve the software behavior, the sequence is discarded. Although the execution of tests is not able to assure that the software behavior was not changed, it increases the confidence on it. Therefore, a sequence of corrections is always tested before adopted. Algorithm 3 implements the *Testing Behavior* task.

```
Data: Software base represented by software
Data: A sequence of applicable plans to be tested
Result: This task indicates if the sequence of applicable plans must be
        discarded
Simulate the application of the sequence on the original software
if The software behavior is preserved after applying the plans sequence
then
    The plans sequence must not be discarded
else
    The plans sequence must be discarded
end
```

5.2.4.2. Evaluating Impact. As discussed in Section 3, the task of changing a software by correcting a smell can result in the increase or decrease of the software code quality. Therefore, the main goal of the *Evaluating Impact* task, described in Algorithm 4, is to evaluate the impact of each sequence of corrections on the code quality. More specifically, such an impact is evaluated by simulating the application of each sequence of corrections and measuring the gains and losses of violating the activated prohibition norms and for fulfilling the activated obligation norms according to the quality model captured by Table 1 in Section 4.

```
Data: Software base represented by software
Data: The Norms base represented by norms
Data: The sequence of corrections to be evaluated
Result: The impact of the sequence of corrections
Simulate the application of the sequence of corrections on the original
software
foreach norm n in norms do
    if n is activated then
        if The state regulated by n is satisfied due to the simulated
        software then
            if n is PROHIBITION then
                Evaluate the gains and loses of violating n
            end
            if n is OBLIGATION then
                Evaluate the gains and loses of fulfilling n
            end
        end
    end
end
```

Example. Let us consider once again our motivational example, where two corrections can be applied: *Extract Class* and *Encapsulate Fields*. The role of the *Plan Selection* function is to check which sequence of corrections preserve the software behavior and provides the highest quality evolution. Table 3 shows the quality evolution generated by following the four possible strategies: (i) apply only *Extract Class*; (ii) apply only the *Encapsulate Fields*; (iii) apply first *Extract Class* and then *Encapsulate Fields*; and (iv) apply first *Encapsulate Fields* and then *Extract Class*. We observe that the strategy that first executes *Encapsulate Fields* and then *Extract Class* provides the highest quality evolution (80%) when compared with other strategies. Assuming that we have checked in the *Testing Behavior* task that the observable behavior is preserved, such strategy will be the one selected to be applied.

5.2.5. Execute Intention

After the *Plans Selection* function selects the sequence of corrections to be applied, the function *Execute Intention* is executed in order to perform the selected sequence, as described in Algorithm 5. At this moment, the developer can be contacted to approve the

Table 3
Software quality evolution after applying a corrections sequence.

	Corrections sequence	Quality evolution (%)
1	Encapsulate Fields	65
2	Encapsulate Fields → Extract Class	80
3	Extract Class	57
4	Extract Class → Encapsulate Fields	79

intended corrections. Then, the actions that compose the body of the corrections are executed one by one. After the execution of all actions, the software is corrected and a commit is executed in order to update the software code.

```
Data: Software base represented by software
Data: The corrections sequence selected to be applied
Result: The corrected software
The developer can be contacted to approve the corrections sequence.
foreach correction c in sequence do
    foreach action a in c do
        Apply a to the software
    end
end
commit
```

6. Experiment settings

The goal of this study is to assess whether the *AutoRefactoring* platform is an appropriate tool to perform refactoring. We have analyzed five non-trivial open source projects using the *Goal, Question, Metrics (GQM)* assessment approach (Basili, Caldiera, & Rombach, 1994). This Section is organized as follows. Section 6.1 poses five research questions that drive our assessment. Section 6.2 presents the projects we have selected to participate in our experiment. Section 6.3 indicates the tools we have used when conducting the experiment and, finally, Section 6.4 presents some details about the machines used to run the experiment.

6.1. Planning

The general research question we would like to answer is:

RQ. Is our refactoring platform able to increase the quality of a software code without changing its observable behavior?
This general question was organized into 5 other questions with more specific scopes, as follows:
Question 1. Does our refactoring platform support code smells detection?
Question 2. Does our refactoring platform decrease the number of code smells?
Question 3. Does our refactoring platform increase the software code quality?
Question 4. Does our refactoring platform preserve the software observable behavior?
Question 5. Is the performance of our platform comparable to existing refactoring approaches?

Our intention with *Question 1* is to demonstrate that our platform is able to detect the code smells of a software by using different detection tools available in the literature (See Section 1). We use existing smells detection tools to check if the prohibited conditions described in the prohibition norms can be found in the code. To answer *Question 2*, we count the number of code smells in the original software and also in the refactored software. We demonstrate that our platform is able not only to detect the code smells (as in *Question 1*), but also to decrease the number of code smells by applying sequences of corrections. In order to answer *Question 3*, we use the quality model proposed by Raed et al. (2011) to evaluate the software code quality before and after performing a correction to demonstrate that the code quality increases (see Section 4). Regarding *Question 4*, we demonstrate that the software observable behavior has not changed after performing corrections. In other to do so, our platform uses tools available in the literature that test the software before and after corrections and compare its behavior. Finally, *Question 5* is answered by comparing the quality of the refactoring performed by our approach with the quality of the refactoring performed by other approaches.

Table 4
Subjects.

Project name	Project version	Application domain	Number of classes
Log4j	2.0	Loggin library	166
SweetHome3D	4.1.1	Interior design application	194
HSQLDB	2.3.1	Database engine	532
jEdit	4.2	Text editor	534
xerces	2.8	Parser	711

Table 5
Refactoring performance.

	Time
Log4j	1 min 7 s
SweetH3D	3 min 43 s
jEdit	4 min 3 s
Xerces	6 min 40 s
HSQLDB	9 min 16 s

6.2. Subject selection

We have analyzed 5 open source projects written in *Java* that have from 166 up to 711 classes. These projects come from different domains. The chosen projects are: *Log4j*, a software development tool, *SweetHome3D*, a interior design editor, *HSQLDB*, a database engine, *JEdit*, a text editor, and *Xerces*, a parser generator. In Table 4 we state the project name, the version used at the time of evaluation, its application domain and total number of classes used in its implementation.

6.3. Instrumentation

In order to be able to refactor the above mentioned projects, we have instantiated the *AutoRefactoring* platform five times. Each instantiation generated an agent to refactor a given project. The agents use several tools available in the platform to accomplish the four refactoring requirements listed in Section 1. They are organized depending on the requirement to be fulfilled, and therefore to answer the different research questions listed in Section 6.1. In this section we identify each tool and relate it to a different research question.

The *AutoRefactoring* Platform was implemented in Jason 1.3.10 and Eclipse Kepler.[12] To answer Question 1 and detect the smells, our platform used the following tools (Fontana et al., 2012): JDeodorant, Checkstyle and inCode. We have also used the Java Development Tools (JDT)[13] and Language Toolkit (LTK)[14] to implement the corrections to fix the detected smells, one of the steps related to *Question 2*. The model used to evaluate the software code quality, related to *Questions 3* and *5*, was implemented by using Chidamber and Kemerer Java Metrics (ckjm) (Spinellis, 2006). The tool *SafeRefactor* (Soares et al., 2013a) was used to verify the software behavior after performing corrections, as expected in *Question*

[12] Eclipse. http://www.eclipse.org.
[13] JDT. http://www.eclipse.org/jdt/.
[14] LTK. https://www.eclipse.org/articles/Article-LTK/ltk.html.

Table 6
Detected code smells.

	Log4j	Xerces	SweetHome3D	HSQLDB	jEdit
Number of detected smells	144	2079	569	3215	773
Public fields	82.64%	46.28%	3.16%	56.3%	54%
Feature envy	9.72%	42.04%	79.96%	42.55%	44.11%
Data clumps	7.64%	11.69%	16.87%	1.15%	1.94%

4. In addition, we have also used the tests provided by the selected projects in order to test them. Notice that other tools could have been used since our platform is flexible and does not depend on any particular tool.

6.4. Operation

We execute the evaluation on a MacBook Air 1.8 GHz Intel Core i7 4 GB, running Mac OS X 10.9. As a first step of our analysis, we ran all open source projects listed in Table 4. Next, we used our platform to refactor these projects by applying the corrections Extract Class, Encapsulate Fields and Move Method. At last but not least, we ran the projects again to assure that the application of the corrections did not introduce errors, according to Soares et al. (2013a). Then, we use the original and the refactored software as input to detection tools and to the ckjm to check the number of smells in each software and to measure the quality of the refactored software, respectively. To implement the detection of the smells, we use the default configuration provided by the detection tools (see Section 6.3).

Table 5 describes the time to perform the refactoring in each analyzed project.

7. Experiment results and discussion

In this section, we present the main results of the experiment outlined in Section 6. In Section 7.1 we answer the research questions listed in Section 6.1. Section 7.2 discusses some threats to validity our experiment.

7.1. Research questions

Next we answer and discuss the research questions.

7.1.1. Does our refactoring platform support code smells detection?

The agent implemented by using our platform is able to use different existing smells detection tools in order to check for code smells. In this study, the platform used the tools *JDeodorant*, *inCode* and *Checkstyle* (as mentioned in Section 6.3).

After updating its bases with the current version of the project to be refactored and with the norms defined by the developer, task realized by the *Belief Update* function (Section 5.2.1), the agent is prepared to identify the code smells. Such job is done by the *Checking Prohibitions* task that provides to the code smells detection tools the states regulated by the prohibition norms. As stated in Section 4, the states identified in the prohibition norms are the code smells that must be avoided. If the tools are able to identify such states, the agent figures out that a prohibition norm has been violated, i.e., a code smell has been detected and must be refactored.

Although a variety of code smells can be detected by using such tools, as described in Fontana et al. (2012), we focused our study on detecting the code smells *Public Fields*, *Feature Envy*[15] and *Data Clumps* since they are the ones that more frequently appear in the analyzed projects and the ones that have more impact on the quality of these projects.

In order to increase the confidence of the detected smells, we consider only smells identified at least by two tools. In our case study, they are as follows:

(*Public Fields*): JDeodorant and inCode;
(*Feature Envy*): JDeodorant and inCode;
(*Data Clumps*): JDeodorant and Checkstyle.

Table 6 shows the results for the identification of these smells in the analyzed projects. It presents the number of all detected smells in each project and the percentage of each smell type in each project. For instance, 144 code smells were detected in the project Log4j, 82.64% of the 144 detected smells are Public Fields, 9.72% are Feature Envy and 7.64% are Data Clumps. Such results demonstrate that the agent implemented by using our platform provides support for the detection of different code smells by using different tools.

Table 7
Code smells correction.

	Log4j	Xerces	SweetHome3D	HSQLDB	jEdit
Correction rate	80.56%	35.02%	10.19%	59.72%	62.74%

7.1.2. Does our refactoring platform decrease the number of code smells?

After detecting the code smell, the agent executes the *Generating Events* task to identify the obligation norms that have been activated. The states regulated by the obligation norms represent the corrections that must be applied in order to fix the detected smells. Such states generate events that are captured by the *Unify Event* function. This function unifies such events with the invocation conditions of the plans. The plans are the ones responsible to apply the corrections by executing the set of actions that changes the software code. The *Check Context* function selects the applicable plans following such unification. These are the plans able to apply the corrections.

Based on the set of applicable plans, the *Plans Selection* function is responsible for selecting the sequence of corrections (or plans) that will provide the best code quality. As stated in Section 5.2.4 the agent uses a search-based technique to verify possible sequences of plans, it discards the ones that change the software observable and it selects the sequence of plans that provides the highest improvement on the software code quality. The agent uses two tools (mentioned in 6.3) to implement and to apply three different corrections: *Move Method*, *Encapsulate Field* and *Extract Class* to correct the smells *Feature Envy*, *Public Fields* and *Data Clumps*, respectively.

After applying the corrections, the agent uses again the tools *JDeodorant*, *inCode* and *Checkstyle* in order to check if the number of smells has really decreased. As stated in Table 7 we demonstrate that the agents are able to decrease the number of code smells after applying the corrections. For instance, 144 code smells were

[15] Feature Envy occurs when existing a method that seems more interested in a class other than the one it actually is in Fowler et al. (1999).

Table 8
Quality evolution.

	Log4j (%)	Xerces (%)	SweetHome3D (%)	HSQLDB (%)	jEdit (%)
Reusability	12.81	7.46	29.73	17.14	56.59
Flexibility	−0.18	6.9	13.85	8.32	3.37
Extendibility	0.23	6.82	13.64	8.3	3.46
Effectiveness	6.71	4.23	0.01	1.14	4.12
Quality evolution	19.58	25.41	57.22	34.9	70.54

Table 9
Quality evolution average.

	Encapsulate Field	Move Method	Extract Class
Average rate	7.35%	32.33%	−14.61%

detected in the project *Log4j* and 80.56% of them were corrected by our refactoring platform.

7.1.3. Does our refactoring platform increases the software code quality?

As stated in Section 4, in order to measure the software code quality the agent instantiated by using our platform use the tool *CKJM* to measure the reusability, flexibility, extendibility and effectiveness of the code. Such job is performed by the *Evaluating Impact* task compares the values of such quality factors before and after the corrections application. Table 8 summarizes the quality evolution of the projects according to such measures and reveals that the applications of the corrections increase the projects quality from 19.58% (*Log4j*) to 70.54% (*JEdit*), an average of about 41.53% per project.

Although the agents were able to increase the quality of all analyzed projects by applying the corrections mentioned in Section 7.1.2, not all corrections impact positively in the code quality according to our quality model. As shown in Table 9, while the corrections Encapsulate Field and Move Method are responsible for evolving the code quality an average of 7.35% and 32.33% per project, respectively, the application of the Extract Class decreases the projects quality an average of −14.61% per project, according to our quality model.

The decreasing of the projects quality resulting from the *Extract Class* application takes the agent to avoid the correction of some *Data Clumps*. The drawback is that it decreases the rate of corrected code smells but the benefits is that it increases the projects code quality. Table 10 presents the rate of corrected smells in each analyzed project after applying the refactoring strategies used to achieve the quality evolutions shown in Table 8.

We observe from Table 10 that the agents of our platform corrects from 10.19% (SweetHome3D) up to 80.56% (Log4J) of the existing smells while considering the projects code quality.

7.1.4. Does our refactoring platform preserves software observable behavior?

In order to check the preservation of the projects behavior, each agent runs the tests provided by the project it is refactoring twice: before applying any correction and after applying the correction. In addition, the agent also uses the tool *SafeRefactor* to test the project. Such tests are executed by the *Testing Behavior* task.

After using the testes provided by the projects and also the tool *SafeRefactor* in each project, it was possible to conclude that the behavior of the refactored projects has not changed. Although the execution of tests does not assure that software behavior is preserved, they increase the confidence on behavior preservation.

Table 10
Comparison between quality evolution and correction rate.

	Log4j (%)	Xerces (%)	SweetHome3D (%)	HSQLDB (%)	jEdit (%)
Quality evolution	19.58	25.41	57.22	34.9	70.54
Correction rate	80.56	35.02	10.19	59.72	62.74

Table 11
Correction rates comparison.

	Correction rate	
	AutoRefactoring platform (%)	Existing approach (%)
Log4j	80.56	93.75
Xerces	35.02	37.37
SweetH3D	10.19	33.22
HSQLDB	59.72	60.84
jEdit	62.74	64.68
	mean = 49.65	*mean* = 57.97

7.1.5. Is the performance of our platform comparable to existing refactoring approaches?

Many approaches (Harman & Tratt, 2007; Kessentini et al., 2011; O'Keeffe & Cinnéide, 2008; Ouni et al., 2012, 2013) have been proposed to correct smells, however such works focus on applying corrections to fix code smells without concerning with the quality of the software code resulting of the implementation of such corrections.

This is a problem since not all corrections result in the increasing of the software code quality, as demonstrated before in Section 7.1.3. The application of the Extract Class decreases the quality of the project an average of 14.61% per project.

In order to compare our platform with others (Harman & Tratt, 2007; Kessentini et al., 2011; O'Keeffe & Cinnéide, 2008; Ouni et al., 2012, 2013) able to do corrections, we have measured the number of code smells corrected by each agent over the total of code smell detected by each agent in each project as well as the quality of the refactored project. Since the results presented by most recent refactoring approach (Ouni et al., 2013) outperform the results presented by Ouni et al. (2012), Kessentini et al. (2011), Harman and Tratt (2007) and O'Keeffe and Cinnéide (2008), we focus our study on the most recent one.

Table 11 presents the smells correction rates of our platform and the approach presented in Ouni et al. (2013). The approach presented in Ouni et al. (2013) has corrected more code smells than ours. However, when we use a t-test (Wohlin et al., 2012) we notice that such difference is not relevant. The t-test was used to test if the mean of corrections applied by the existing approach and our platform differ or not. By applying the t-test to the values described in Table 11, we obtained the *p-value* = 0.31. According to Wohlin et al. (2012), in order to represent a significant difference between two means, normally, the *p-value* should be lower than 0.05. Thus, we conclude that the mean of corrections applied by the existing approach is not significant greater than ours. Such conclusion confirms our hypothesis that the performance of our platform is equivalent to the performance of others.

Table 12 compares the quality evolution rates of the projects after executing our approach and after executing the approach presented in Ouni et al. (2013). The quality evolution of the projects increases more when our approach is used to refactoring the projects than when the other approach is used. In fact, after applying the t-test we noticed that the difference between the improvement generated by the use of our approach is significant when compared with the improvement generated by the use of the other approach. By applying the t-test to the values described in Table 12, we

Table 12
Quality evolution rates comparison.

	Quality evolution	
	AutoRefactoring platform (%)	Existing approach (%)
Log4j	19.58	16.66
Xerces	25.41	−3.63
SweetH3D	57.22	14.57
HSQLDB	34.9	26.52
jEdit	70.54	28.72
	mean = 41.53	*mean* = 16.57

obtained the p-value equal to 0.02, i.e., p-value < 0.05. Such value reinforces that the performance of our approach is not only equivalent to the performance of others but it is greater than the others.

Since the main goal of refactoring a software is to increase the quality of its code, we consider that the agents implemented by using our platform have successfully achieved their goals.

7.2. Threats to validity

In this section, we discuss some threats to validity.

7.2.1. Construct validity

Construct validity refers to whether our platform increases the software code quality without introducing behavioral changes. This way, two threats must be considered: how to define the code quality and how to preserve its behavior. In order to evaluate the quality of the projects we use the model described in Section 4. Note that such model can be replaced. We minimize the threat concerning to the software behavior by performing the test cases before and after applying the correction operations. Such test cases were defined from the ones available by the open source projects, besides this new tests were generated using the tool SafeRefactor.

7.2.2. Internal validity

The refactoring platform we propose solves code smells identified by existing detection tools. However, such tools can indicate wrong smells. In order to minimize this threat in our study, we consider just the smells identified by at least two detection tools.

7.2.3. External validity

We refactored 5 open source projects of different domains and sizes that range from 166 lines up to 711 classes. We select a UML modeling tool, database engine, multiagent framework, text editor, drawing software and interior design application. Moreover, we selected well-known Java open source projects used in practice. In this way, we alleviate this threat but we cannot generalize the results to Java projects with million lines of code such as *Android* and *Hibernate*.

8. Conclusion

In this paper, we present an agent-based platform, called *AutoRefactoring*[16] platform, to build an agent able to autonomously perform refactoring on the code of projects.

Our approach contributes to the state-of-the-art on code refactoring in two main perspectives. First of all, it is able to deal with all the refactoring requirements at once, namely code smell identification, corrections determination, quality improvement, and observable behavior preservation. *Practically, the platform provides to the developer that wants to apply refactoring on its projects the benefit of using only one approach that is able to put together all the steps of the refactoring process. Without using our proposed approach, the developers need to select and apply different tools that focus on specific parts of the refactoring process in order to refactoring a project*. Second, our approach focuses on applying refactoring in order to improve the software code quality. If the correction of a given code smells does not improve the quality of the code, such correction is not applied by the agent of our platform. Although according to Fowler et al. (1999) the improvement of the software code quality is the main goal of applying refactoring, many tools correct all the detected code smells without concerning with the quality of the refactoring project. Therefore, *the practical benefit of using our approach is that it guarantees that the quality of the refactoring project will be superior than the quality of the original project, even if some code smell are not corrected.*

We have evaluated our approach by considering only four quality factors taken from Raed et al. (2011). It is a limitation of our work since it limits the comparison with other techniques that do not take into account those quality factors. In addition, the agent instantiated by our platform is not able to suggest tests to test the code that was refactored. It does only uses the testes provided by the project it is refactoring and the tests executed by tools provided by the developer. Therefore, it may be the case that the agent is not able to test the refactored code. Moreover, another important limitation of our platform is that it is not able to correct the code smells generated when refactoring a project. It is not able to keeping refactoring the project till no new code smell is created.

In order to deal such limitations, we are extending the quality model used in this work by adding quality factors adopted in existing works (Kataoka et al., 2002). We are also building a module to the refactoring platform able to generate test cases to the refactored code by using a code formal specification in Alloy (Jackson, 2012). Finally, we are extending the agent-based approach to software change propagation described in Dam and Winikoff (2011) in order to deal with the generation of code smells after correcting a project.

[16] The *AutoRefactoring* platform is available at https://sites.google.com/a/ic.ufal.br/refactoringplatform/.

References

Alshayeb, M. (2011). The impact of refactoring on class and architecture stability. *Journal of Research Practice in Information Technology, 43*(4), 269–284.

Basili, V., Caldiera, G., & Rombach, D. H. (1994). *The goal question metric approach*. Wiley.

Bertran, I. M., Arcoverde, R., Garcia, A., Chavez, C., & von Staa, A. (2012). On the relevance of code anomalies for identifying architecture degradation symptoms. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering* (pp. 277–286).

Bordini, R. H., Hubner, J. F., & Wooldridge, M. (2007). *Programming multi-agent systems in agentspeak using Jason*. Wiley.

Bratman, M. E. (1999). *Intention, plans, and practical reason*. CSLI Publications.

Brown, W. J., Malveau, R. C., Brown, W. H., McCormick, H. W., & Mowbray, T. J. (1998). *Anti patterns: Refactoring software, architectures, and projects in crisis*. Wiley.

Dam, H. K., & Winikoff, M. (2011). An agent-oriented approach to change propagation in software maintenance. *Autonomous Agents and Multi-Agent Systems, 23*(3), 384–452.

Daniel, B., Dig, D., Garcia, K., & Marinov, D. (2007). Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 185–194).

da Silva, V. T. (2008). From the specification to the implementation of norms: An automatic approach to generate rules from norms to govern the behavior of agents. *Journal Autonomous Agents and Multi-Agent Systems, 17*(1), 113–155.

Fontana, F. A., Braione, P., & Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology, 11*(2), 5:1–38.

Foster, J. R. (1993). *Cost factors in software maintenance* (Ph.D. thesis). University of Durham.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.

Ge, X., Dubose, Q. L., & Murphy-hill, E. (2012). Reconciling manual and automatic refactoring. In *Proceedings of the 34th international conference on software engineering* (pp. 211–221).

Guéhéneuc, Y., & Sahraoui, H. (2011). A bayesian approach for the detection of code and design smells foutse. In *9th International conference on quality software*.

Harman, M., & Tratt, L. (2007). Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (pp. 1106–1113).

Jackson, D. (2012). *Software abstractions: Logic, language, and analysis*. The MIT Press.

Kataoka, Y., Imai, T., & Andou, H., et al. (2002). A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the international conference on software maintenance*.

Kessentini, M., Kessentini, W., & Erradi, A. (2011). Example-based design defects detection and correction. *Journal of Automated Software Engineering*.

Kim, T.-W., Kim, T.-G., & Seu, J.-H. (2013). Specification and automated detection of code smells using ocl. *International Journal of Software Engineering and Its Applications, 7*(4), 35.

Kolb, R., Muthing, D., Patzke, T., & Yamauchi, K. (2005). A case study in refactoring a legacy component for reuse in a product line. In *Proceedings of the 21st international conference on software maintenance* (pp. 369–378).

Lanza, M., & Marinescu, R. (2006). *Object-oriented metrics in practice*. Springer Verlag.

Liu, H., Yang, L., Niu, Z., Ma, Z., & Shao, W. (2009). 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* (pp. 265–268).

Loelieger, J., & McCullough, M. (2012). *Version control with git: Powerful tools and techniques for collaborative software development*. O'Reilly Media.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance* (pp. 350–359).

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering archive, 30*(2), 126–139.

Moha, N., Gueheneuc, Y., Duchien, L., & Meur, A. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering, 36*(1), 20–36.

Moser, R., Sillitti, A., Abrahamsson, P., & Succi, G. (2006). Does refactoring improve reusability? In *Proceedings of the 9th international conference on Reuse of Off-the-Shelf Components* (pp. 287–297).

Murphy-Hill, E., Parnin, C., & Black, A. P. (2011). How we refactor, and how we know it. *IEEE transactions on software engineering, 38*(1), 5–18.

O'Keeffe, M., & Cinnéide, M. (2008). Search-based refactoring: An empirical study. *Journal of Software Maintenance and Evolution: Research and Practice – Search Based Software Engineering [SBSE], 20*(5), 345–364.

Ouni, A., Kessentini, M., & Sahraoui, H. (2013). Search-based refactoring using recorded code changes. In *17th European conference on software maintenance and reengineering*.

Ouni, A., Kessentini, M., Sahraoui, H., & Hamdi, M. S. (2013). The use of development history in software refactoring using a multi-objective evolutionary algorithm. In *Proceeding of the fifteenth annual conference on genetic and evolutionary computation conference – GECCO '13*.

Ouni, A., Kessentini, M., Sahraoui, H., & Boukadoum, M. (2012). Maintainability defects detection and correction: A multi-objective approach. *Automated Software Engineering, 20*(1), 47–79.

Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T. (2007). Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering* (pp. 75–84).

Padilha, J., Pereira, J., Figueiredo, E., Almeida, J., Garcia, A., & Sant'Anna., C. (2014). On the effectiveness of concern metrics to detect code smells: An empirical study. In *Proceedings of the 26th international conference on advanced information systems engineering (CAiSE 2014)*.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., & Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *28th IEEE/ACM international conference on automated software engineering (ASE)*.

Pilato, C. M., Collings-Sussman, B., & Fitspatrick, B. W. (2009). *Version control with subversion*. O'Reilly Media.

Raed, S., Li, W., Swain, J., & Newman, T. (2011). An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering & Its Applications, 5*(4), 127.

Rao, A. S. (1996). Agentspeak(l): Bdi agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on modelling autonomous agents in a multi-agent world: Agents breaking away*.

Rao, A. S., & Georgeff, M. P. (1991). Modeling rational agents within a bdi-architecture. In *Proceedings of the 2nd international conference on principles of knowledge representation and reasoning*.

Sahraoui, H., Godin, R., & Miceli, T. (2000). Can metrics help to bridge the gap between the improvement of oo design quality and its automation. In *Proceedings of the international conference on software maintenance*.

Schafer, M., Ekman, T., & Moor, O. (2010). Specifying and implementing refactorings. In ACM SIGPLAN Notices - OOPSLA '10 (Vol. 45(10), pp. 286–301).

Seng, O., Stammel, J., & Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the genetic and evolutionary computation conference (GECCO'06)*.

Soares, G., Mongiovi, M., & Gheyi, R. (2011). Identifying overly strong conditions in refactoring implementations. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance* (pp. 173–182).

Soares, G., Gheyi, R., & Massoni, T. (2013a). Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering, 39*(2).

Soares, G., Gheyi, R., Murphy-hill, E., & Johnson, B. (2013b). Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software archive, 86*(4), 1006–1022.

Soares, G., Gheyi, R., Serey, D., & Massoni, T. (2010). Making program refactoring safer. *IEEE Software, 27*(4).

Spinellis, D. (2006). *Code quality: the open source perspective*. Addison-Wesley Professional.

Tourwe, T., & Mens, T. (2003). Identifying refactoring opportunities using logic meta programming. In *Seventh european conference on software maintenance and reengineering*.

van Gurp, J., Brinkkemper, S., & Bosch, J. (2005). Design preservation over subsequent releases of a software product: A case study of baan erp: Practice articles. *Journal of Software Maintenance and Evolution: Research and Practice, 17*(4), 277–306.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., & Wesslen, A. (2012). *Experimentation in software engineering*. Springer.