An Expert System for Correcting JEE Performance Anti-Patterns

Marco Crasso, Alejandro Zunino, Marcelo Campo and Leonardo Moreno

CONICET and ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 439682. Fax.: +54 (2293) 439681

Abstract. Java Enterprise Edition (JEE) middlewares have been successfully employed to build enterprise software systems, while achieving security, flexibility and reliability. However, achieving performance requirements through these middlewares has been identified as challenging. We propose a novel approach for automatically detecting performance problems in JEE-based applications and suggesting how to correct them. The idea is to automatically detect well known performance anti-patterns and suggest how to adapt concrete solutions to the performance problems of an application, from anti-pattern descriptions. Experimental evaluations show that after correcting the anti-patterns found in three JEE reference applications, their response times represent a 40%, 76% and 61% of the original response times.

1 Introduction

Typical enterprise software systems are expected to work under heavy load as fast as possible and scale to hundreds of thousands users, while presenting several design goals that are difficult to achieve, such as security, flexibility, maintainability, reliability and manageability. Java Enterprise Edition based middlewares have greatly reduced the effort for developing enterprise systems [1]. These middlewares provide a number of ready to use *enterprise services* for supporting typical enterprise requirements with little development effort, with respect to ad-hoc solutions. For example, through JEE components, namely EJB (Enterprise Java Beans), developers can build systems in which security, persistence and transaction integrity are provided by the middleware.

This is an outstanding approach for succeeding on the development of distributed enterprise systems. Nevertheless, one major consequence of the higher level of abstraction provided by these middlewares is the loss of control over the system that developers often experience. For example, EJB components can be executed in clusters of servers, where JEE, besides managing load balancing, hides the real location of each component to the developer. From a syntactical point of view, interactions between two remote

EJBs look just the same as between two plain Java objects in an application source code. However, a simple message between two EJBs may involve serialization of message parameters and network transfers. Accordingly, this hidden complexity makes very difficult, even for skilled designers, to completely understand the performance impacts of their decisions, and has been the cause of many performance problems [1,2].

Many approaches for diagnosing and solving performance problems in enterprise distributed systems have been proposed. One approach is to predict the performance of a system by modeling its components and interactions under different workloads [3,4]. This approach predicts the performance of a system at early stages of its life cycle, which commonly would be an advantage. However, typical enterprise systems often have ever changing requirements that should be implemented as fast as possible. Therefore, developers rely on development methods that promote short and frequent test-code-refactor cycles. This, in turn, alters the original application design and then performance simulations have to be redone. To further complicate this scenario, popular middlewares alter the original code of an application during compilation, deployment or execution by *injecting* middleware services into application components [5]. A major consequence of these technologies is that they not only alter the original code of the applications, but also their design, making performance modeling methods harder to adopt. Besides, "a quantitative prediction requires application-specific parameters, which can be obtained only from a substantial prototype implementation" [3].

Clearly, performance issues should be handled not only at design time, but also during the whole application life cycle. A common practice is to monitor a system performance during run-time to gather profiling information [6,7,8,9]. The resulting information is then analyzed by experts to find out possible courses of actions, which may involve 1) refactoring parts of the application being analyzed or 2) tuning the middleware. We aim at providing developers with assistance in the first type of actions. We propose JEETuningExpert, an expert system that besides detecting performance problems, as other approaches do [8,9,10], also proposes how to solve them. JEETuningExpert takes as input an existing JEE application source code and instruments it to obtain traces. Afterward, the application is executed normally and JEETuningExpert obtains logs that summarize run-time interactions between application components. JEETuningExpert uses a rule-based system to analyze *anti-patterns* in the gathered logs. Anti-patterns are well known culprits of causing performance problems. Specifically, JEE anti-patterns have been studied and provided with a proven and repeatable remedy [11,12], providing us a sound theoretical foundation to build JEETuningExpert. From this analysis, JEETuningExpert detects a number of points in the application code that may cause

performance problems and suggests developers how to solve them. The idea is to adapt a recommendation from the solution associated with an anti-pattern. Although anti-pattern descriptions are very useful, we believe that these descriptions may further alleviate a developer's task if they are carefully confined to the domain of the application that he/she wants to enhance. We have validated the proposed approach by analyzing three well known JEE reference applications (Avitek Medical Records[1], the Java Pet Store[2] and the Libra Book Store[3]), showing that by following JEETuningExpert recommendations, a developer was able to speed up the applications by 59.92%, 23.06% and 38.87%, respectively. Besides, JEETuningExpert found all the supported anti-patterns present in the analyzed applications. Therefore, the main contributions of this paper are:

– a novel approach to detect and correct JEE performance anti-patterns,
– empirical evidence showing the feasibility of this approach.

The next section discusses the most relevant related work, Section 3 describes some JEE performance anti-patterns, Section 4 explains JEETuningExpert, then Section 5 presents the experimental evaluation of the approach and Section 6 concludes the paper.

2 Related Work

Below, we first discuss approaches based on models and simulations to prevent systems from having performance problems. We then analyze methods that rely on experts to determine whether a system has performance issues. Finally, we describe works that use automatic monitoring techniques to help developers on detecting performance issues.

Some authors have proposed to use the *queuing nets* formalism for building models, which allow performance engineers to represent the components of a system and then simulate their performance under different workloads [3,13]. Instead, [4] shows that by combining queuing nets with Petri nets, engineers may capture more aspects of a system. These approaches generally lead to a model describing the overall form of a system performance when two important conditions are satisfied. The first condition is that developers should have precise knowledge of the underlying software infrastructure, including the middleware, the operating system, etc., and the expected workloads. Another condition is that every decision made during design should be implemented as planned, otherwise the model would represent "another" system. Unfortunately, satisfying these conditions is not always possible, as explained in the previous section.

[1] Avitek Medical Records `https://avitek-medical.projects.dev2dev.bea.com/`
[2] Java Pet Store `http://java.sun.com/developer/releases/petstore/`
[3] Libra Book Store `http://j2ee-examples.sourceforge.net/index.html`

[6,7] proposed iterative methodologies for formally reviewing a system and correcting its performance problems. The methodologies start by gathering information about the run-time performance of the JEE application under evaluation. Then, the collected information is analyzed by experts in performance, who determine whether a part of the system represents a bottleneck and, in turn, should be refactored. Then, the experts must re-apply the method until all the problems have been eradicated.

COMPAS [10] is a tool for monitoring the performance of JEE-based applications. This tool intercepts the invocations to each EJB of the application under evaluation and finds out the response time per invoked method. Then, the developers of the application are in charge of detecting performance problems based on monitoring results. As manually analyzing these results might be cumbersome when the number of operations is large, COMPAS uses a mechanism for automatically filtering non-critical operations, which reports an operation only if its response time is historically greater than a threshold. JPManager [8] is a tool that follows a similar approach for monitoring enterprise applications and finding bottlenecks. Performance Anti-pattern Detection (PAD) [9] is a tool for automatically detecting JEE performance anti-patterns. PAD is based upon monitoring techniques and a rule engine to identify anti-patterns. The detection effectiveness of PAD was evaluated using 2 case studies [9].

One similarity between COMPAS, JPManager, PAD and our approach, is that they can be used with the methodologies described in [6,7] to automatically detect performance bottlenecks. The main difference between them, is that JEETuningExpert also provides means to correct detected performance problems. JEETuningExpert exploits existing anti-pattern descriptions twice: (1) for detecting potential performance problems based on reported anti-pattern “symptoms”, (2) for assisting developers to adapt a concrete solution to their performance problems from reported anti-pattern solutions.

3 Background

The JEE platform supplies developers with reusable standardized services for addressing non-functional requirements such as security, scalability, availability and transactional integrity, freeing developers from building the underlying middleware. EJBs are server-side components that rely on middleware services for allowing developers to easily build the business logic of their applications. An EJB inhabits the business tier and connects to databases, other beans or applications, in a way that is transparent for developers. Moreover, an EJB can be accessed by clients from the presentation tier, by means of its remote interface. In fact, a client accesses an EJB through the middleware, which is responsible for creating, replicating and locating specific beans.

Although JEE is a well proven and popular choice for developing distributed enterprise systems, it can be difficult, even for skilled designers, to completely understand the performance impacts of their decisions. Recently, both academia and industry have striven to detect recurring design patterns that produce a negative impact upon the performance of JEE-based applications. The literature presents catalogs of JEE performance anti-patterns, such as [11] and [12]. The current implementation of JEETuning-Expert supports four anti-patterns, which are described through the rest of this section.

3.1 Anti-pattern: Round-tripping

Remote EJBs enable other distributed components to access to them as if they were deployed at the same host. The straightest way for retrieving the details of a bean is through its “getters” methods, nevertheless this requires multiple fine grained remote calls. For example, on the left side of Fig. 1 a remote client makes many invocations over a network for obtaining some bean attributes. In the context of JEE-based applications, a careless use of EJB technology may render too many "round-trips" at the expense of the overall system performance. *Round-tripping* can be seen as a special case of the well known *Empty Semi Trucks* anti-pattern [11], which occurs when an excessive number of requests is required to perform a task causing inefficient use of available bandwidth.

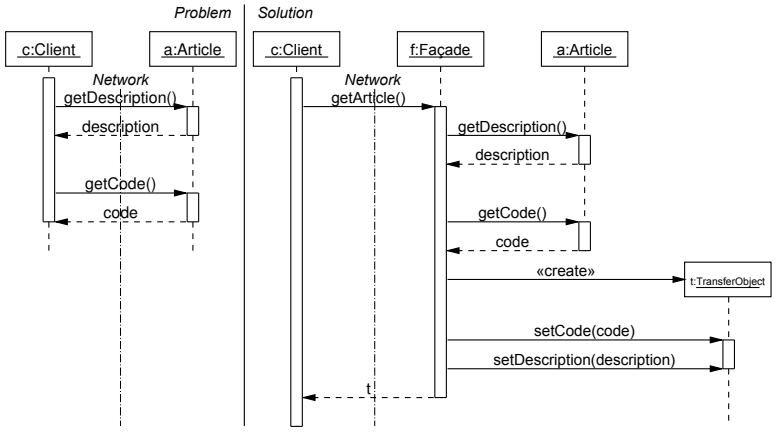

Fig. 1. Example of *Round-tripping* anti-pattern.

One alternative to eradicate the *Round-tripping* anti-pattern consists of grouping multiple fine grained remote calls together in one coarse-grained remote call. The right side of Fig. 1 shows a session Façade [5] component that provides a coarse-grained

remote call for retrieving the details from a bean. In fact, this bean is represented by a TransferObject [5], which is an object view of the individual bean designed for being serialized and transmitted over a network. Under this solution, the number of remote calls is reduced to 1, which improves performance if local communication is efficient.

3.2 Anti-pattern: Multiple EJB accesses per request

The previous anti-pattern replaces many read-only remote accesses with only one access. The *Multiple EJB accesses per request* anti-pattern is considered a special case of round-trip, which requires that, at least, one access is intended for modifying the bean. Figure 2 extends the *Round-tripping* anti-pattern example by replacing the *"getCode"* remote call with a call for modifying the bean (*"setCode*"). A refactored solution for this case consists of a session Façade that offers a service for updating the bean and returning its details simultaneously. In consequence, if the client uses this service then the number of remote calls will be reduced to 1, as shown in the right side of Fig. 2.

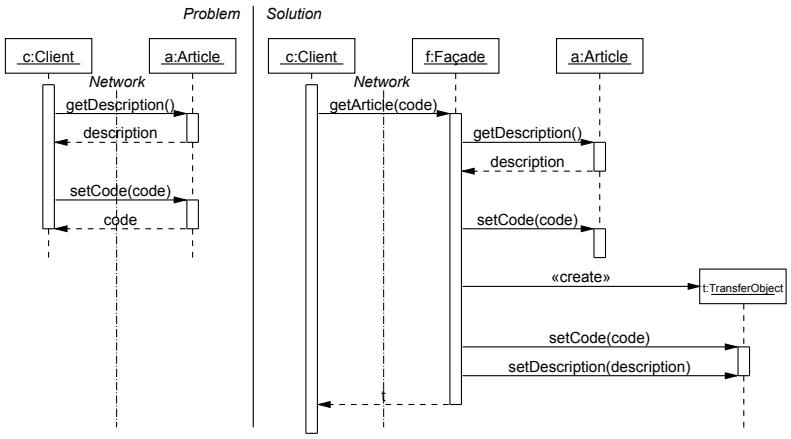

Fig. 2. Example of multiple EJB accesses per request.

3.3 Anti-pattern: Redundant JNDI lookups

Another case of communication overhead occurs when there are redundant accesses to a JNDI (Java Naming and Directory Interface) server. A JNDI server allows clients to look up available EJBs, data sources and connection factories, from anywhere in the network. The left side of Fig. 3 depicts a client that looks up a target service twice,

thus 2 remote calls are made. One solution for this anti-pattern is to supply clients with a local component, named Service Locator [5,11], which has a cache of previous searched services. This refactorization reduces the number of remote calls to 1 per service. In the right side of Fig. 3, a Service Locator resolves the second lookup process locally, i.e., without neither requiring serialization of messages nor network transfers.

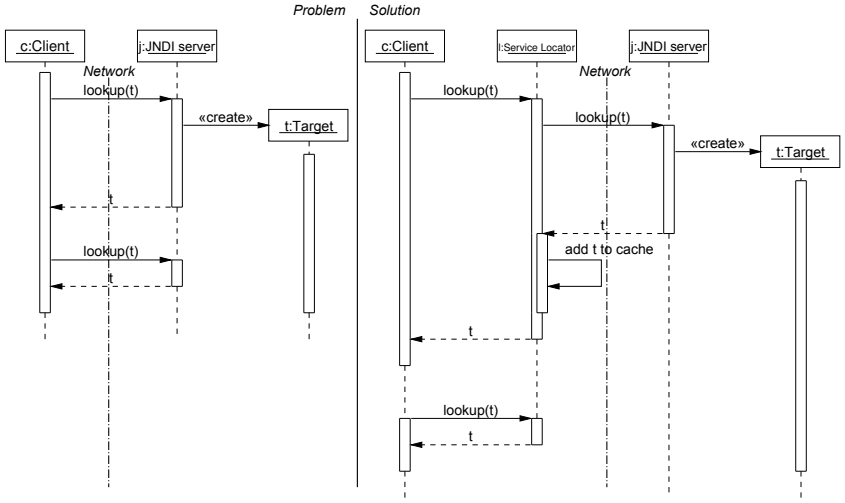

Fig. 3. Example of redundant JNDI lookups anti-pattern.

3.4 Anti-pattern: Sending excessive amount of objects

Finally, another case of communication overhead occurs when a huge volume of data is transmitted to the clients. Commonly, Web applications provide facilities for searching and browsing large query result sets. This, besides requiring a long time to transmit the result sets over the network, demands a long time from clients to process that sets afterward. In consequence, these facilities are expensive when large result sets are transmitted from the server-side to the client-side. Moreover, in the context of JEE applications, this situation presents another problem. To obtain a list of values developers usually implement EJB finder methods. A finder method returns a collection of remote beans stored into a database. Then, to retrieve the details of each bean, many round-trips are necessary. Graphically, the left side of Fig. 4 presents this problem.

The Value List Handler [5] design pattern provides a more efficient way to iterate a large list of remote beans. The iterator typically accesses a local ordered collection

of objects, representing a sub-range of the large list, thus providing the client with a result set whose size should not demand high processing time. Similarly, for avoiding communication overheads the size of the sub-range is chosen according to the network characteristics. Finally, round-trips are eradicated by returning a list of TransferObjects instead of a list of beans. Figure 4 shows this solution.

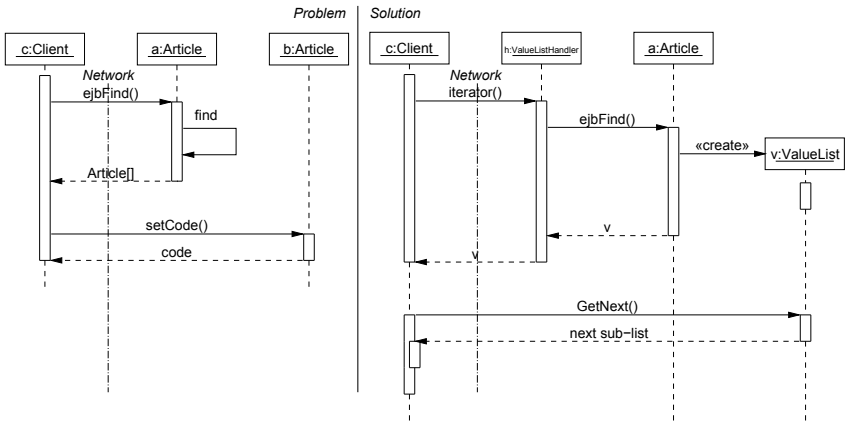

Fig. 4. Example of Value List Handler anti-pattern.

4 JEETuningExpert

In general, an expert system is a computer program, with a set of rules, that analyzes information about a specific class of problems, and recommends courses of user action to correct them, acting similarly to a human expert. JEETuningExpert is an expert system for assisting JEE developers in improving the performance of their applications. The underlying approach bases upon representing symptoms of known performance problems as rules, gathering information from an application, analyzing whether these rules hold for the collected information and reporting solutions to the problems found. Broadly, JEETuningExpert gathers traces of execution by monitoring interactions between application components. Then, JEETuningExpert searches for performance anti-patterns within the gathered information. Finally, it reports the anti-patterns found along with a description of how to incorporate their associated solutions into the application under evaluation. The rest of this section explains the main activities of the approach.

4.1 Information gathering

We have analyzed which information is essential to detect and correct the supported anti-patterns. JEE anti-patterns stem from three causes: unnecessary remote calls, repeated JNDI lookups and excessive amount of returned objects [11]. To analyze whether an application suffers from these symptoms and correct it afterward, it is necessary to keep a trace of the interactions between JEE components during each user's request. Specifically, JEETuningExpert observes: (1) the user's request information (an URL), (2) the invoked EJB data (the name of its Java class, the name of the invoked method, whether it is a read-only method, the number and type of returned values), (3) the invoker data (the name of its class, the line number in the class code where the invocation occurs), and (4) the JNDI lookup data (the name of the service requested through the JNDI server).

To gather this information, JEETuningExpert uses a non-intrusive mechanism for Java, called *Dynamic Proxies* [5]. A Dynamic Proxy is an object that decorates another object, called target, for intercepting all the method invocations that are sent to the target. JEETuningExpert only intercepts method invocations of EJBs and JNDI lookups, by decorating EJBs that implement remote interfaces and JNDI server instances. During the interception of a method, JEETuningExpert keeps trace of each remote invocation and its originator within a stack-trace, which is a snapshot of the Java stack-trace. This stack-trace associates each invocation with the class name of its originator and the line number where the invocation occurs. Finally, the output of this activity is a log containing records of "Invocation" and "JndiLookup" instances that occur during a user's session. Instances of "Invocation" and "JndiLookup" classes represent gathered information. Alternatively, a collection of logs may be stored on disk by using an XML format that allows developers to load the Java representation of the records into memory at any time. As we have developed this activity of the approach by means of portable libraries and standard Java APIs, it can be executed with any JEE application server.

4.2 Information analysis

To automatize the detection of anti-patterns within an application, the idea is to represent each individual record of a log as a fact and the anti-patterns as conjunctions, a.k.a. rules, in propositional and first order logic. Thereafter, rules are matched against the known facts, firing a rule if its corresponding anti-pattern occurs.

JEETuningExpert uses Drools[4], a declarative language and a rules engine based on the Rete algorithm, for translating users' session logs and anti-patterns into logic

[4] Drools `http://www.jboss.org/drools/`

clauses. The Rete algorithm is widely used to implement matching functionality within pattern-matching engines to support forward chaining and inferencing. Drools allows translating a Java representation of a log into a knowledge-base in a straight manner, by converting Java instances in facts. Then, instances of “Invocation” and “JndiLookup” classes are translated into facts. Moreover, Drools rule definition language enables the specification of anti-patterns. To illustrate this, we present a part of the rule for detecting whether there are redundant accesses to a JNDI server.

```
rule "Redundant JNDI lookup. Code SLC."                                    1
when                                                                       2
    $lookup: JndiLookup(                                                   3
        $resourceName: requestedResourceName,                              4
        $stackTrace: shortStackTrace                                       5
    ) exists JndiLookup(                                                   6
        requestedResourceName == $resourceName,                            7
        this != $lookup )                                                  8
    not Antipattern(                                                       9
        id == ($resourceName + $stackTrace),                               10
        type == "SLC" )                                                    11
```

This rule will be triggered if two JNDI requests for the same resource take place during the same user’s session. Using Drools, this rule will fire if there are two “JndiLookup” instances with the same “requestedResourceName” attribute. The rule starts by finding any instance that stands for a JNDI lookup fact in the knowledge base (lines 3-6). The found instance along with the name of the requested resource are stored in variables (lines 3 and 4), for comparing them with other lookups afterward. Subsequently, the rule looks for *another* JNDI lookup (lines 6-8) that embraces the same resource (line 7). Additionally, in line 5 a variable stores the information related to the invoker, which will be used to suggest how to remedy this anti-pattern.

We have specified the other anti-patterns similarly. Concretely, the rule for Round-tripping anti-pattern will hold if there are two or more instances of “Invocation” with the same “stackTrace” and “resourceName” attributes, that is the same client component makes many invocations over a network for obtaining some attributes from a bean. As *Multiple EJB accesses per request* anti-pattern is a special case of *Round-tripping* [11], its corresponding rule additionally checks whether at least one invocation is for updating the remote EJB. Finally, to detect Sending excessive amount of objects anti-pattern, a rule checks whether the number of returned values is above a tolerance threshold.

4.3 Problem-Solution reporting

To help developers in correcting design problems that might cause an inadequate performance of their applications, we propose to carefully confine the solution related to

an anti-pattern to the application under evaluation. For example, the solution related to redundant JNDI lookups requires to known which components ask for a same service many times, to incorporate a Service Locator there. JEETuningExpert outputs the class name/s and line number/s, where the Service Locator should be incorporated, as shown in a concrete problem-solution report for one of the case studies of Section 5:

```
Antipattern{                                                                    12
    type = SLC,                                                                 13
    solution = Add a Service Locator with cache for accessing to "RecordSessionEJB.   14
        RecordSessionHome"
    from com.bea.medrec.utils.EJBHomeFactory.lookupHome (EJBHomeFactory.java:111) }   15
```

JEETuningExpert uses the consequence of a rule for adapting solutions. The consequence of a rule is a set of facts that will be "executed" if the rule holds true. The consequence associated with the rule presented in Section 4.2 is:

```
then insert(new Antipattern(                                                    16
    ($resourceName + $stackTrace), "SLC",                                       17
      "Add a Service Locator with cache for accessing                           18
      to \"" + $resourceName + "\" from " + $stackTrace) );                     19
end                                                                             20
```

A consequence starts with the reserved word "then" (line 16). Particularly, this consequence adds to the knowledge base a fact standing for an occurrence of the *Redundant JNDI lookups* anti-pattern. This fact is represented as a new instance of class *Antipattern* (line 16). This object contains an unique identifier, a type (line 17) and a description of how to correct the problem (lines 18-19). This *ad-hoc* solution is built on the invoker data that were collected through the stack-trace (see line 5 of the rule in Section 4.2). To avoid redundant reports of a same anti-pattern occurrence, a rule fires only the first time a problem is detected. Thereafter, a rule will fail because a fact verifies whether the same anti-pattern has been previously detected (line 9-11).

For correcting *Round-tripping* and *Multiple EJB accesses per request* anti-patterns, JEETuningExpert states where to add a session Façade, as explained in Section 3.1. It also suggests which components should be refactored to supply clients with a Value List Handler, thus remedying the *Sending excessive amount of objects* anti-pattern.

5 Experimental Results

The evaluation consisted of two parts. First, we evaluated how JEETuningExpert performed for detecting anti-patterns with: Avitek Medical Records (MedRec), the Java Pet Store and the Libra Book Store. In the experiments JEETuningExpert found all the

anti-patterns of the evaluated applications. Second, we assessed the performance improvements that were gained from the refactorizations suggested by JEETuningExpert and the overhead introduced by its information gathering activity. The tuned applications have empirically shown to be faster than their original counterparts.

5.1 Anti-pattern detection effectiveness

We compared a manually built list of anti-patterns with the list of anti-patterns that were automatically detected by JEETuningExpert. The former list was built by an experienced JEE software developer who revised available documentation and source code of the case studies exhaustively. To build the latter list, three user's sessions, in which the user accesses to all the available functions of the systems, were captured and, in turn, reproduced with the original applications plus the dynamic proxies. Initially, the overall detection effectiveness of JEETuningExpert was 90%, as summarized in the fourth column of Table 1. All anti-patterns in Libra were found, but JEETuningExpert failed in detecting one anti-pattern in MedRec and another in Pet Store. These results stem from two peculiarities found in the captured sessions for MedRec and Pet Store. After we dealt with these peculiarities, the missing anti-patterns were detected by our tool.

Table 1. Summary of the anti-pattern detection effectiveness.

Case Study	Anti-patterns	False positives	Initially detected	Finally detected
MedRec	7	0	6 (85%)	7 (100%)
Pet store	9	0	8 (88%)	9 (100%)
Libra	4	0	4 (100%)	4 (100%)
Total	20	0	18 (90%)	20 (100%)

With respect to MedRec, the undetected anti-pattern occurs when executing "ApprovePatientRequestAction", a functionality requiring administrator privileges. As the monitored user had not administrator privileges, this functionality was not captured, so that JEETuningExpert had not the information needed to be aware of the anti-pattern. To correct this problem we monitored the application when an administrator invoked ApprovePatientRequestAction and supplied the rule engine with the new execution log. Accordingly, JEETuningExpert properly detected the anti-pattern.

With respect to Pet Store, JEETuningExpert failed to detect an occurrence of the *Sending excessive amount of objects* anti-pattern. The rule for detecting this kind of anti-

pattern builds on a pre-defined maximum number of exchangeable objects. We initially used a value for this threshold that was bigger than the quantity of exchangeable objects occurring in the monitored session. Then, we adjusted the threshold to the real number of instances and loaded the rule engine with the modified rule. Alternatively, we could have monitored a different user session, in which the user exchanges more objects, and then supplied the tool with this new session. However, we decided to change the threshold in the corresponding rule rather than re-capture the user's session, because this is a more realistic way to get this done in real world scenarios.

5.2 Impact evaluation: benefits and penalties

We have assessed the response time for each executed request during a captured user's session of the applications. The user's session comprised 32 requests for MedRec, 40 requests for the Pet Store and 39 for Libra. Moreover, we have measured response times when the dynamic proxies were gathering information and after a developer incorporated the refactorizations suggested by JEETuningExpert. To mitigate any noise introduced by external conditions, each user's session was reproduced 100 times with each application version (original application, original application with dynamic proxies and tuned application). To perform a fair comparison, we used the same hardware/software environment for deploying all the versions of an application. It is worth noting that, as each application has been deployed on a different application server, namely Weblogic, Sun Java Application Server and JBoss, our tool has been tested on these alternatives as well. Afterward, we averaged response times over the 100 reproductions from each request. Figure 5 depicts a summary of the results, in which a bar stands for the sum of the averaged response times per request, in milliseconds. The numbers within bars represent these values as well. From left to right, the first bar within each group depicts the performance of an original application, while the second represents its tuned variant. Additionally, the figure shows above each bar standing for the original version of an application, the overhead introduced by the information gathering activity.

To sum up, the dynamic proxies have empirically shown to increase the response times, on average, of MedRec, Pet Store and Libra, by 49.25%, 19.67% and 27.65%, respectively. In some cases it may by undesirable to impose such an overhead to an application. Therefore, JEETuningExpert should be executed during periods of time, when normal users do not use the systems, or on test environments. On the other hand, the sum of averaged response times for MedRec represents a 40.08% of the sum of the original response times. Likewise, 76.94% for Pet Store and 61.13% for Libra. This

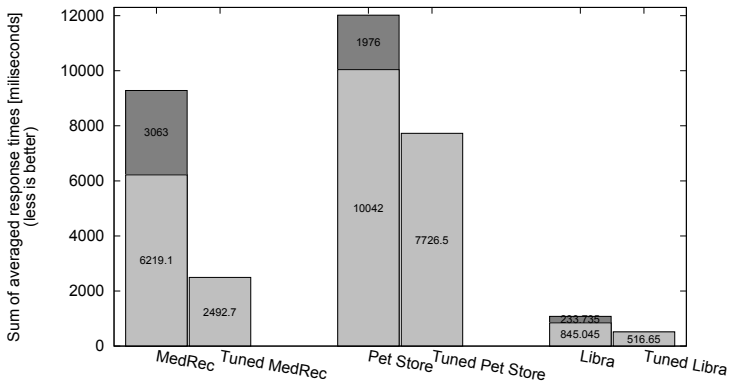

Fig. 5. Quantitative benefits and penalties of using JEETuningExpert.

means that the tuned variants are 59.92%, 23.06% and 38.87% faster than their original counterparts, respectively.

6 Conclusions

We have presented JEETuningExpert, an approach to detect performance anti-patterns within JEE applications and propose how to solve them. Broadly, it imitates a performance engineer, who gathers information about the performance of an application, analyzes it and presents possible courses of actions to correct found problems.

We have used three well known applications to evaluate the effectiveness of JEETuningExpert to detect anti-patterns and to measure the performance improvements after eradicating them. The evaluations have shown that the extent to which users' sessions are monitored, impacts on the effectiveness for detecting anti-patterns. In this sense, after augmenting the extent of the monitored sessions, our approach has identified all the anti-patterns that had been manually detected by an experienced JEE developer. In addition, we have empirically showed that the recommendations made by JEETuningExpert allow the same developer to decrease the response times of the three case studies between 23.06% to 59.92%. Although these results can not be generalized, as our approach relies on removing well known performance anti-patterns, it is reasonable to expect at least a small performance advantage when applying the recommendations.

A limitation of JEETuningExpert's current implementation is that it supports four performance anti-patterns. Although the supported anti-patterns have been identified as very frequent [11,12] and our experiments have empirically strengthened their significance, we are incorporating more anti-patterns to the knowledge base. Another limita-

tion is that an application might suffer from a performance overhead, when the information gathering activity is being carried on. However, our experiments light that, by automatically reproducing captured users' sessions, JEETuningExpert detects present anti-patterns anyway. In other words, the supported anti-patterns were successfully detected, even though the information gathering activity was carried out with captured user sessions. In fact, this is a more realistic way to perform a performance evaluation of a system in real world scenarios, because solving found problems usually requires refactoring the source code of the system, and its re-deployment. Obviously, in some cases it would not be desired, or allowed, to re-deploy a system while normal users are accessing it. Finally, we are developing a plug-in for the Eclipse SDK for guiding programmers when applying the recommendations made by JEETuningExpert.

References

1. Altendorf, E., Hohman, M., Zabicki, R.: Using J2EE on a large, Web-based project. IEEE Software **19** (2002) 81–89
2. Cecchet, E., Marguerite, J., Zwaenepoel, W.: Performance and scalability of EJB applications. ACM SIGPLAN Notices **37** (2002) 246–261
3. Liu, Y., Fekete, A., Gorton, I.: Design-level performance prediction of component-based applications. IEEE Transactions on Software Engineering **31** (2005) 928–941
4. Kounev, S.: Performance modeling and evaluation of distributed component-based systems using queueing petri nets. IEEE Transactions on Software Engineering **32** (2006) 486–502
5. Metsker, S.J., Wake, W.C.: Design Patterns In Java. Addison Wesley (2006)
6. Chow, K., Morin, R., Shiv, K.: Enterprise Java performance: Best practices. Intel Technology Journal **7** (2003) 32–46
7. Williams, L.G., Smith, C.U.: PASA: a method for the performance assessment of software architectures. In: 3^{rd} International Workshop on Software and Performance. (2002)
8. Guo, J., Liao, Y., Parviz, B.: A performance validation tool for J2EE applications. In: 13^{th} Annual IEEE International Symposium on Engineering of Computer Based Systems. (2006)
9. Parsons, T., Murphy, J.: Detecting performance antipatterns in component based enterprise systems. Journal of Object Technology **7** (2008) 55–90
10. Mos, A., Murphy, J.: Compas: Adaptive performance monitoring of component-based systems. In: 26^{th} International Conference on Software Engineering. (2004) 35 – 40
11. Smith, C.U., Williams, L.G.: More new software antipatterns: Even more ways to shoot yourself in the foot. In: 29^{th} Int. Computer Measurement Group Conference. (2003)
12. Dudney, B., Asbury, S., Krozak, J., Wittkopf, K.: J2EE AntiPatterns. Wiley (2003)
13. Chen, S., Liu, Y., Gorton, I., Liu, A.: Performance prediction of component-based applications. Journal of Systems and Software **74** (2005) 35–43