JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 7, No. 3, March–April 2008

Detecting Performance Antipatterns in Component Based Enterprise Systems

Trevor Parsons and John Murphy, Performance Engineering Lab, School of Computer Science and Informatics, University College Dublin, Ireland

We introduce an approach for automatic detection of performance antipatterns. The approach is based on a number of advanced monitoring and analysis techniques. The advanced analysis is used to identify relationships and patterns in the monitored data. This information is subsequently used to reconstruct a design model of the underlying system, which is loaded into a rule engine in order to identify predefined antipatterns. We give results of applying this approach to identify a number of antipatterns in two JEE applications. Finally, this work also categorises JEE antipatterns into categories based on the data needed to detect them.

1 INTRODUCTION

Today's enterprise applications are becoming more and more complex and have been moving towards distributed architectures made up of a heterogeneous collection of servers (see figure 1). Each server can in turn be made up of a large number of software components that interact to service different user requests. Component based enterprise frameworks [1] (such as JEE or .Net for example) alleviate the burden of developers that need to construct such systems, by providing system level services (e.g. security, transactions etc.). Thus, developers no longer have to worry about building the underlying infrastructure of these systems and can instead concentrate their efforts on developing functional requirements. However, in order to meet non-functional requirements (such as performance requirements for example), developers are still required to have an understanding as to what is actually happening "under the hood" of the application. Unfortunately due to the complexity of such systems developers very often find it difficult to obtain a complete understanding of the entire system behaviour. Consequently, it is common that they make incorrect decisions during development, that lead to design flaws in the application. Such flaws can lead to problems such as poor system performance, maintainability issues, reliability issues, etc. Evidence of this problem can be seen in recent surveys [2] which suggest that a high percentage of enterprise applications fail to meet their performance requirements on time or within budget.

Current development and testing tools fail to help developers address this lack of understanding in relation to complex enterprise systems. For example, most of today's performance monitoring tools merely profile the running system and present vast amounts of relatively unstructured data to the tool user. The amount of data

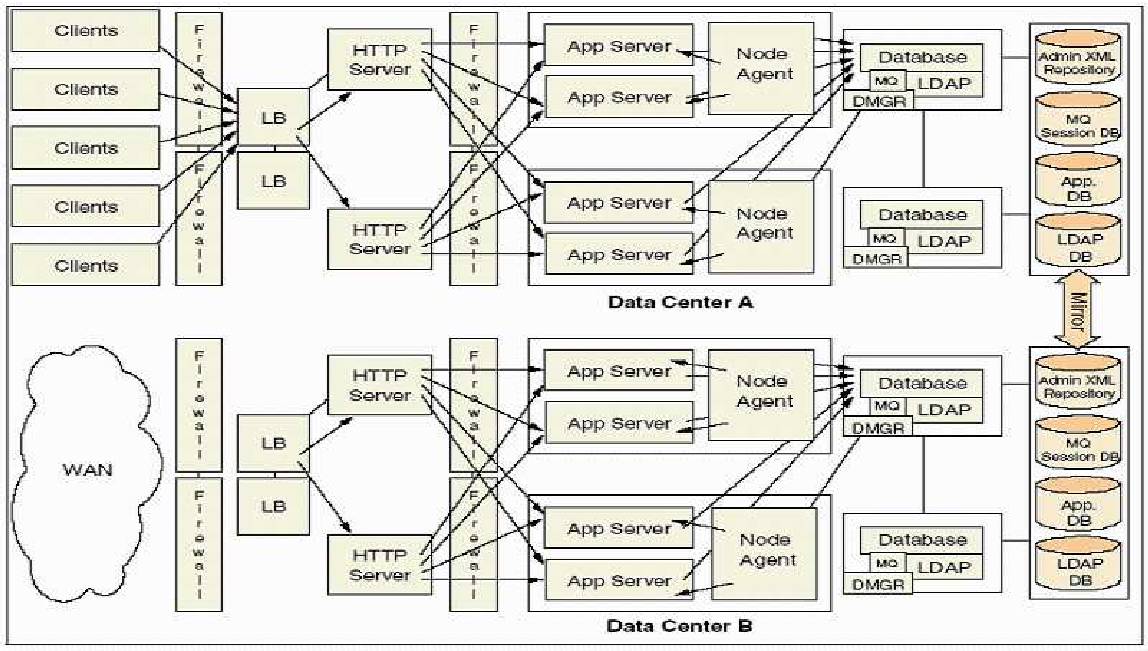


Figure 1: Typical Enterprise Architecture [3]

produced when monitoring even simple single user systems can be quite large. When monitoring large scale multi user systems made up of a myriad of software components, the amount of data produced can be overwhelming. Consequently developers or system testers often find it difficult to make sense of this information. In such circumstances it can be extremely difficult and time consuming to identify performance issues in an inefficient system.

For enterprise applications there are a number of common design mistakes that consistently occur causing undesirable results [4] [5]. In fact the same design flaw can often manifest itself in different ways across various parts of the application. A large number of these well known problems have been documented as software antipatterns [4] [5] [6] [7] [8]. Similar to software patterns [9], which document best practices (often described as a proven solution to a problem in a context[1]) in software development, antipatterns document common bad practices. However as well as documenting the bad practice, antipatterns also give the corresponding solution to the problem. Thus, they can be used to help developers identify problems within their system. Furthermore they can be used to easily rectify these issues, by applying the corresponding documented solution[2].

This first contribution of this paper is an *approach to automatically identify performance antipatterns within enterprise applications built on top of component based enterprise systems.* Our approach extracts the run-time system design from data collected during monitoring by applying a number of advanced analysis techniques.

[1]For a more precise definition of a software pattern see [10]
[2]The literature [11] gives a more complete overview of patterns and antipatterns

A systems run-time design can be defined as an instantiation (or instantiations) of a systems design decisions which have been made during development [11]. A run-time design model captures structural and behavioural information from an executing system. It contains information on the components that are executed, as well as the dependencies and patterns of communication between components that occur at run-time [11]. Using advanced analysis techniques we summarise the run-time data and identify relationships and patterns that might suggest potential antipatterns in the system. The information extracted from the monitoring data can be represented in a run-time design model of the system. This model is loaded into a rule engine or knowledge base which, (using predefined rules) can identify potential (well known) performance antipatterns that exist in the system. Any detected antipatterns are subsequently presented to the user along with specific data on the antipattern instance. This approach takes the burden away from developers having to sift through large volumes of data, in order to understand issues in their systems, and instead automates this process.

The second contribution of this work is *a study of JEE performance antipatterns*. We show a hierarchy of performance antipatterns, from high level language independent antipatterns, to technology specific ones. We also show that a high percentage of antipatterns related to enterprise technologies are performance related. We further categorise the JEE performance antipatterns into a number of groups. We focus on two of these groups in particular (design antipatterns and deployment antipatterns) and further categorise the antipatterns within them into groups based on the data needed to detect them.

The remainder of this paper is structured as follows: Section 2 discusses the limitations of current performance tools and states why we believe there is a need for more advanced analysis of the data that is collected from monitoring enterprise applications. Section 3 gives an overview of our approach and a categorisation of the different antipattern types that exist. Section 4 gives details on monitoring techniques that are used to extract information from component based enterprise systems such that antipattern detection can be applied. In section 5 we discuss a number of analysis techniques used to identify relationships and patterns in the run-time data. This information can be used to reconstruct the run-time design of the system. In this section we also outline a number of advanced analysis techniques that can be applied to summarise the data produced during monitoring. A detection mechanism (based on a rule engine approach) is outlined in section 6. In this section we also group the antipatterns that we can detect into a number of different categories (based on the data required to detect them). Section 7 presents our results from applying our Performance Antipattern Detection (PAD) tool to a number of component based enterprise applications. In this section we show how we successfully detected a number of performance antipatterns in these applications. The applications tested include a sample application and a real enterprise application from IBM. Related work and our conclusions are discussed in sections 8 and 9 respectively.

2 LIMITATIONS OF CURRENT PERFORMANCE TOOLS

Current performance tools for component based enterprise systems are quite limited, insofar as they tend to profile running systems and present vast amounts of low-level data to the tool user. Most of today's Java profilers for instance work by monitoring at the JVM level. This is achieved by interfacing with the JVM through a standard mechanism (e.g. the Java Virtual Machine Profiler Interface [12] or Java Virtual Machine Tools Interface [13]). This allows the profiler to collected information such as memory/CPU consumption on any class loaded by the JVM. The information collected is then presented to the user. However, for enterprise systems the number of classes loaded by the JVM can be in the order of thousands. The classes can generally be broken down into the following categories; application level classes (written by the application developers), middleware classes (corresponding to container services) and lower level java library classes. A major issue is that, while developers are generally interested in obtaining information on their own code, it can be very difficult for developers to distinguish their code from lower level middleware and library calls. Another issue with such tools is that they tend to present the information to the user in basic formats. For example they often present lists of the different objects created, the number of instances, related CPU and memory consumption etc. Although, from this type of information developers can determine the most resource intensive/common objects in the system, it can be difficult to determine the cause of a performance issue without also understanding the run-time context of these objects (i.e. the sequence of events that lead to a particular object being instantiated/called). Commercial profilers (e.g. [14]) often present object graphs showing parent child relationships between objects in the system. However it can be still difficult to trace the ordered sequence of events that lead to particular problems in the system (since these graphs do not maintain the order of calls). Consequently in conjunction with using such profiling tools developers are very often required to spend much time tracing through reams of source code to identify issues in their applications.

The most significant problem with the current tools is that they only give a small indication of where potential problems exist in the system, since they fail to give a sufficient run-time context and also fail to perform any meaningful analysis on the data collected. There is a real need for more advanced performance tools that do not merely collect low level data on a running system. Instead these tools should collect data at the correct level of abstraction that the developers work at (e.g. component level for JEE systems), while at the same time they need to provide a sufficient run-time context for the data collected (e.g. run-time paths [11], dynamic call traces [15]) such that problems can be easily identified and assessed. Furthermore it is also desirable that more advanced analysis be applied to the data collected to highlight potential problems in the system automatically, such that developers do not have to waste time correlating large volumes of information.

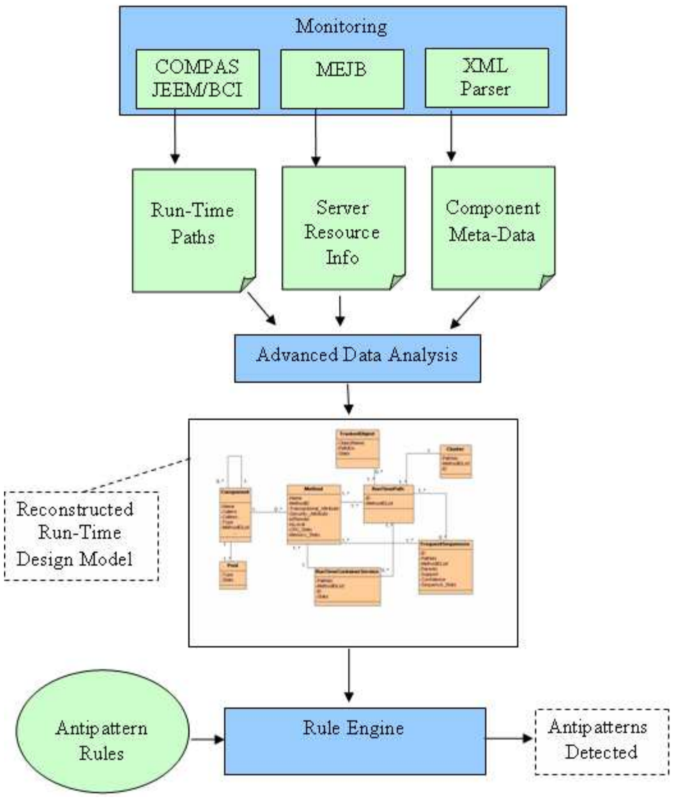


Figure 2: PAD Tool Overview

3 OVERVIEW OF PERFORMANCE ANTIPATTERN DETECTION TOOL

In light of the limitations of current performance tools we propose an approach to automatically identify performance antipatterns in component based enterprise systems [16]. This approach has been realised in the PAD tool (see figure 2). Our approach improves on current tools, taking the onus away from developers having to sift through large volumes of data by performing analysis on the data collected, and automatically identifying potential issues in the system. The tool consists of three main modules: a monitoring module, an analysis module and a detection module. The monitoring module (section 4) is responsible for collecting run-time information on the different components that make up an enterprise application. The PAD monitoring module is end-to-end i.e. it collects data on all (server side) tiers that make up the enterprise application. The monitoring approaches allow for (a) identification of component relationships, (b) identification of communication patterns between components, (c) tracking of objects (and how they are used) across components, (d) the collection of component performance metrics, (e) the collection of component meta data (e.g. container services used, bean type etc.) and (f) the collection of information on server resources. The monitoring is performed at the component level and the techniques used are portable across different middleware implementations since they make use of standard JEE mechanisms. Thus they are

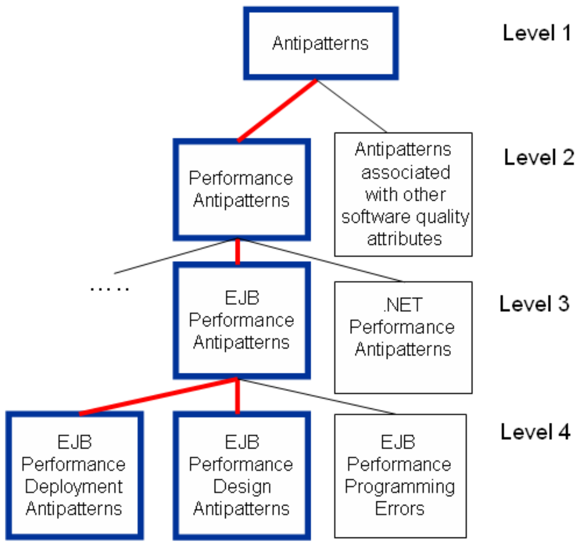

Antipatterns
Level 1
Performance Antipatterns
Antipatterns associated with other software quality attributes
Level 2
.....
EJB Performance Antipatterns
.NET Performance Antipatterns
Level 3
EJB Performance Deployment Antipatterns
EJB Performance Design Antipatterns
EJB Performance Programming Errors
Level 4

Figure 3: Hierarchy of Antipatterns

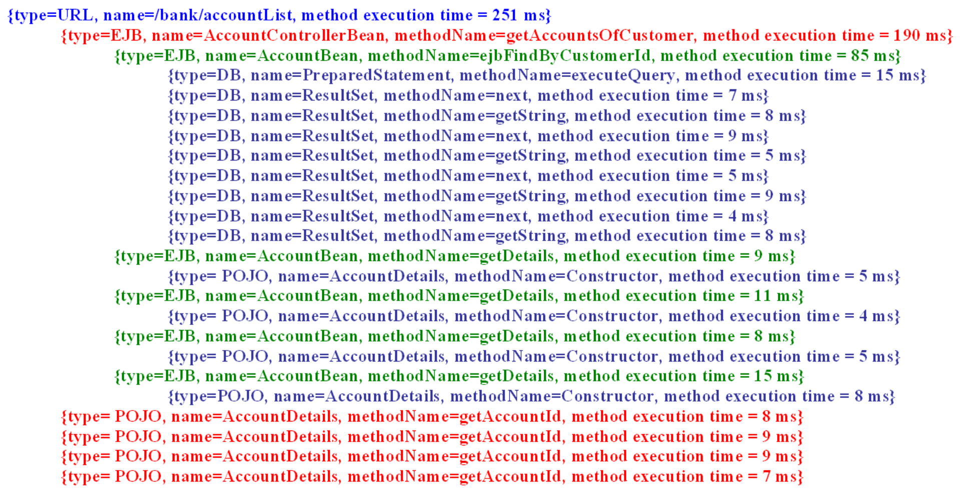
suitable for truly heterogeneous systems made up of servers from different software vendors. The data collected during monitoring is passed to the analysis module where the design of the application is automatically reconstructed. During analysis (see section 5) a number of techniques are applied to extract the relationships from the monitored data that reflect interesting aspects of the system design. Furthermore, an effort is also required during analysis to reduce and summarise the large volume of data produced during monitoring. In section 5 we discuss a number of techniques that can be utilised for these purposes. The output from the analysis module is a run-time design model of the system (see figure 8) which captures the relationships extracted from the monitored data. This model can be loaded into a rule engine, representing the detection module, in the form of rule engine specific *facts*. Rules can be input into the rule engine which describe the antipatterns that we want to detect in the model. Rules are specified so that the rule's conditions verify the occurrence of a certain antipattern. Subsequently, when existing facts match a rule's condition clauses, the rule action is fired indicating the antipattern detection (see section 6). In the following subsection we categorise the different types of antipatterns that can exist for enterprise applications. In particular we focus on the antipatterns detected by the PAD tool, i.e. performance design and deployment antipatterns for component based enterprise technologies.

Antipattern Overview

Figure 3 gives an antipattern hierarchy diagram. At the top of the diagram we have high level technology independent software antipatterns. Brown et al. [7] introduced a number of such antipatterns concerned with a range of software quality attributes (such as re-usability, maintainability, performance etc.). More recently Smith and Williams [6] introduced general performance antipatterns which solely focus on performance concerns (i.e. level 2). The performance antipatterns presented in the literature [6] are high level and language-independent antipatterns. They describe situations where a sub-optimal performance design choice has been made. Instances of the antipatterns documented by Smith and Williams, however, occur throughout different technologies. Many of these problems are especially common in enterprise technologies where performance is often a major concern (level 3). JEE antipatterns have been presented in [4] and [5]. The literature [4] concentrates on EJB antipatterns, while [5] lists antipatterns concerned with a number of aspects of the JEE technology (i.e. Servlets, JSP, EJB and Web Services). We have analysed the antipatterns from both sources. From a total of 43 antipatterns documented in [4] we have identified 34 (79%) of them to be performance related antipatterns (since they can have a significant impact on system performance). From a total of 52 antipatterns documented in the literature [5] we identified 28 performance related antipatterns (54%). The high proportion of antipatterns from [4] and [5], that are related to performance, is further evidence that performance is an important software quality attribute for enterprise systems, and that poor performance design is common in such systems.

We further divided all JEE performance antipatterns identified into 3 different categories (level 4): (a) Performance programming errors, (b) performance design antipatterns and (c) performance deployment antipatterns. Performance programming errors (a) can be defined as common mistakes made by developers that result in degraded system performance. They yield no design trade-off and always have a negative impact on performance. Examples include memory leaks, deadlocks, improper cleanup of resources such as database connections, etc. Generally developers are unaware of the presence of performance programming errors in the system. The Rotting Session Garbage antipattern [4] is an example of a performance programming error that is often made by developers using the EJB technology. This antipattern occurs when a client fails to explicitly remove a stateful session bean when finished using it. The orphaned bean will continue to live in the application server using up system resources until it times out. Until then, the EJB container must manage the bean, which can involve the relatively expensive job of passivating the bean to make room for other active beans. In many situations fixing programming errors alone will not improve the overall system performance such that requirements are met. Often it is the case that the system design requires modification. Performance design (b) and deployment (c) antipatterns can be defined as instances of sub-optimal design or sub-optimal deployment settings that exist within the application i.e. situations where an inefficient design choice has been taken. In such situations an alternative more efficient deign choice exists. Developers are often aware of having made these choices, but can be unaware of their consequences. Performance design and deployment antipatterns can be used to identify and resolve these situations since they document both the sub-optimal design and its corresponding optimal solution.

We are interested in both design and deployment antipatterns since, with component based frameworks such as JEE, many decisions that were inherent in the design and coding of applications in the past, have been abstracted out into the deploy-



```
{type=URL, name=/bank/accountList, method execution time = 251 ms}
      {type=EJB, name=AccountControllerBean, methodName=getAccountsOfCustomer, method execution time = 190 ms}
            {type=EJB, name=AccountBean, methodName=ejbFindByCustomerId, method execution time = 85 ms}
                  {type=DB, name=PreparedStatement, methodName=executeQuery, method execution time = 15 ms}
                  {type=DB, name=ResultSet, methodName=next, method execution time = 7 ms}
                  {type=DB, name=ResultSet, methodName=getString, method execution time = 8 ms}
                  {type=DB, name=ResultSet, methodName=next, method execution time = 9 ms}
                  {type=DB, name=ResultSet, methodName=getString, method execution time = 5 ms}
                  {type=DB, name=ResultSet, methodName=next, method execution time = 5 ms}
                  {type=DB, name=ResultSet, methodName=getString, method execution time = 9 ms}
                  {type=DB, name=ResultSet, methodName=next, method execution time = 4 ms}
                  {type=DB, name=ResultSet, methodName=getString, method execution time = 8 ms}
            {type=EJB, name=AccountBean, methodName=getDetails, method execution time = 9 ms}
                  {type= POJO, name=AccountDetails, methodName=Constructor, method execution time = 5 ms}
            {type=EJB, name=AccountBean, methodName=getDetails, method execution time = 11 ms}
                  {type= POJO, name=AccountDetails, methodName=Constructor, method execution time = 4 ms}
            {type=EJB, name=AccountBean, methodName=getDetails, method execution time = 8 ms}
                  {type= POJO, name=AccountDetails, methodName=Constructor, method execution time = 5 ms}
            {type=EJB, name=AccountBean, methodName=getDetails, method execution time = 15 ms}
                  {type=POJO, name=AccountDetails, methodName=Constructor, method execution time = 8 ms}
      {type= POJO, name=AccountDetails, methodName=getAccountId, method execution time = 8 ms}
      {type= POJO, name=AccountDetails, methodName=getAccountId, method execution time = 9 ms}
      {type= POJO, name=AccountDetails, methodName=getAccountId, method execution time = 9 ms}
      {type= POJO, name=AccountDetails, methodName=getAccountId, method execution time = 7 ms}
```

Figure 4: Example Run-Time-Path

ment settings of the application. With EJB for example the granularity and type of transactions can be specified in the XML deployment descriptors of the application. As a result, when making deployment time configuration settings, different design trade-offs that can significantly impact performance must be considered.

In section 6 we give the categories of performance design and deployment antipatterns that our PAD tool can currently detect. In this section we categorise the antipatterns further into groups related to the data required to detect them.

4 MONITORING

Our monitoring module is responsible for collecting information on the system under test such that a detailed representation of the system can be recreated and potential antipatterns identified. In particular we obtain information on component relationships, component communication patterns, component resource usage, component object usage, component meta data and server resource usage. Using this information we can extract the required relationships to reconstruct the run-time design for performance antipattern detection. Next we detail the different techniques required to capture this information in a portable manner. Our monitoring approaches are applied to a running application and do not require any analysis of the source code.

Capturing Component Interactions, Communication Patterns and Performance Metrics

In order to be able to deduce the run-time design from an application we need to identify the relationships that exist between the different components that make up the system. These relationships can be captured by recording run-time paths [17]

[illegible]

[11]. Run-time paths capture the ordered sequence of events that execute to service a user request. Figure 4 gives a run-time path from a sample JEE application. A diagrammatic representation of this path is given in figure 5. It shows the different components (from a number of different application tiers) that get called to service a particular user action. Run-time paths clearly capture the different component relationships. However, since run-time paths maintain the order of calls between components they also capture communication patterns between the components. Such communication patterns can be analysed to identify inefficient communication between components. Furthermore run-time paths can also contain performance metrics (such as CPU and memory usage) on the component methods that make up the path as well as arguments passed between components and return types. Performance metrics can be essential for identifying if particular relationships in the system are truly affecting the system performance, while information on arguments and return types can be useful for object tracking.

As part of the PAD tool we have recently implemented the COMPAS Java End-to-End Monitoring (JEEM) tool [18], which has the ability to collect component level end-to-end run-time paths from JEE systems. The tool does not require the source code of the application to be available and is completely portable across different middleware implementations. COMPAS JEEM works by injecting a proxy layer in front of the application components through standard JEE mechanisms. The proxy layer contains logic which maintains the order of calls along the different user requests. A major advantage of this tool is that it can collect all the different run-time paths invoked when the system is loaded with multiple simultaneous users. One drawback of the tool is that it requires the system under test to be redeployed during the instrumentation process. A recent extension of the tool, COMPAS Byte Code Instrumentation (BCI), overcomes this problem by using the JVMTI to dynamically instrument the application at run-time [19]. Thus no redeployment of the system is required.

Tracking Objects Across Components

Objects can also be tracked along run-time paths to allow for analysis of their usage in the system. Such analysis can lead to identification of inefficient object usage. Figure 5 shows an example run-time path which tracks the lifecycle of instances of the AccountDetails data transfer object.

The COMPAS BCI tool has recently been extended to track selected objects across run-time paths [19]. Tool users can select particular classes to be tracked. When an object of the selected class is created it is identified along with its corresponding position in the run-time path. Whenever another method (other than the constructor, or creator method for EJBs) is accessed this is also logged. Thus we can identify where objects are created along the run-time paths and at what points they are accessed. We can effectively see how objects are created, used and passed along the run-time path. Figure 5 shows where instances of the AccountDetails

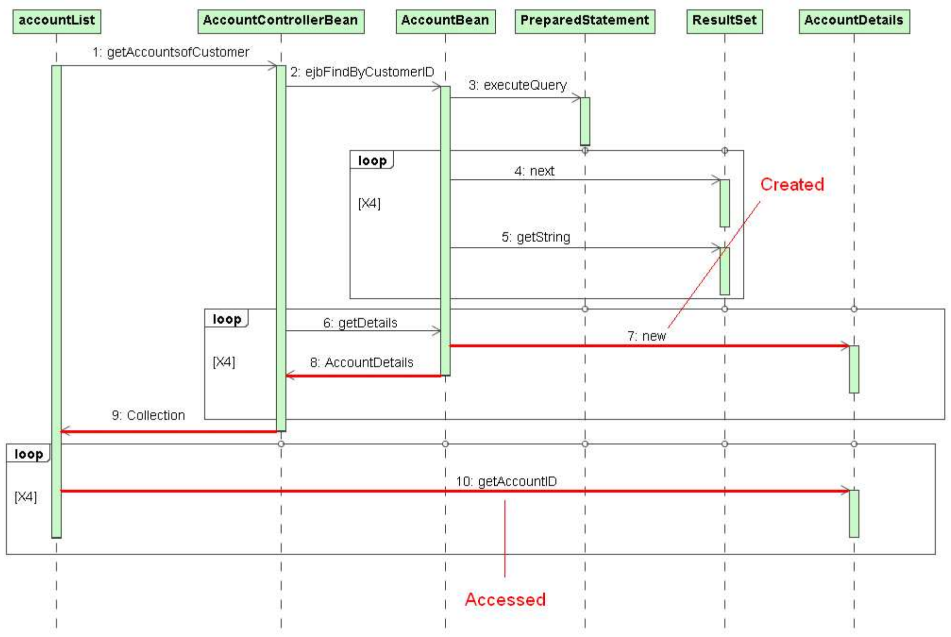


Figure 5: Run-Time-Path with Tracked Object, as a Sequence Diagram

object are created and accessed along a JEE call path. The object was created by an entity bean in the application tier and passed to the web tier where a single method was accessed. This information is required to identify a range of common antipatterns (for example to identify variations of the excessive dynamic object allocation antipattern [6] which manifests itself in a number of different antipatterns in enterprise applications).

Component Meta data

Component based enterprise frameworks are particularly suited for antipattern detection since (a) they specify a component model which defines how the different components types in the model should be used (e.g. using entity beans for persistence) and (b) they generally contain meta data on the components that make up the system. EJB meta data contains structural and functional information on each of the EJB's deployed in the system. For example, information on the EJB component type (i.e. is the bean a stateless session bean, a stateful session bean, an entity bean or a message driven bean). The meta data also contains information on the container services required by the bean's business methods (e.g. whether the bean requires security checks, whether the bean should be invoked in a transactional context, whether the bean can be accessed remotely etc.). This meta data is contained in the XML deployment descriptors that are used to configure the application during deployment. Thus the meta data can be obtained without having to access the source code of the application. The data can be used to reason about the behaviour of certain components. For example, if from our run-time profiling we see that a

[illegible]

particular component is frequently communicating with a database, from the meta data we can check the component type. If this component turns out to be a stateful session bean we could flag this as a potential antipattern, since stateful session beans are not designed to frequently access persistent data (as outlined in the component model specified by the EJB specification[20]). The fact that component based enterprise frameworks specify how certain component types should behave allows us to automatically identify this type of unusual behaviour. Without this information automatic antipattern detection is more difficult. For example, if instead we were monitoring an application made up of plain old Java objects (POJO's) with no information describing how we expect the objects to behave, it would be difficult to flag unusual behaviour. In such situations domain or application specific information could instead be supplied by the application developers. The PAD tool extracts the component meta data from the XML deployment files of the JEE applications automatically using an XML parsing library.

Monitoring Server Resource Usage

In enterprise frameworks such as JEE the different components that make up the application interact with the underlying middleware. The state of the server resources that service these components can significantly impact the overall system performance (e.g. thread pools, database connection pools, object pools etc.). According to the JEE Management specification application servers are required to expose this data through a Java Management Extensions (JMX) interface [21]. Consequently it can be recorded at runtime using a Management EJB (MEJB) [22].

5 ANALYSIS

In this section we discuss how we automatically extract the different relationships (that make up a reconstructed run-time design model of the system) from the monitored data. In particular we detail how we automatically identify inter component relationships, inter component communication patterns and object usage patterns. In addition, we show how run-time container services can be reconstructed and added to the design model. In this section we also discuss how our analysis approach summarises and reduces the amount of data produced during monitoring using clustering and statistical analysis techniques. The summarised data can be used to further enhance the design model. Finally, at the end of this section, we present the reconstructed design model, and the information that it captures.

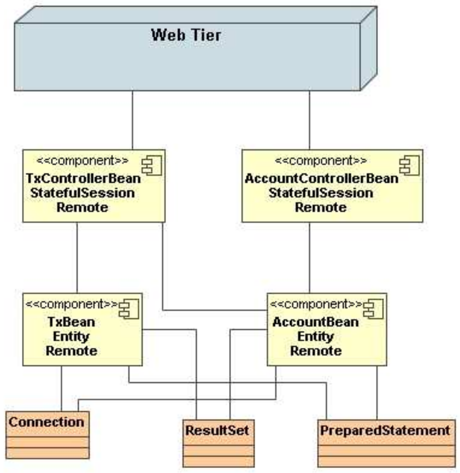


Figure 6: Class Diagram created from Run-Time Path Analysis

Automatically Extracting Component Relationships and Object Usage Patterns

Run-time paths (figures 4 and 5) capture the run-time design of the application (i.e. they capture the design of the instrumented application that is executed during monitoring). As shown in figure 5 run-time paths can be easily represented in a graphical format. Figure 5 shows a run-time path converted to a UML sequence diagram which captures the relationships between the different components for a given user action. Run-time paths are represented at a code level by a tree like data structure. A root node represents the first component in the path, which has an ordered list of callees. Each callee is itself a node which can also have an ordered list of callees. The *RunTimePath* data structure can be traversed to identify the different component relationships that exist within it. The *RunTimePath* data structure is traversed by visiting each node in a preorder fashion. By analysing all run-time paths collected we can build up a collection of all the (run-time) component relationships that exist for the application. This information can be represented in a UML class diagram which shows the overall system architecture (see figure 6). During analysis instances of a *Component* data structure are created which contain this information.

Object usage patterns can also be identified by traversing the run-time paths (produced using COMPAS BCI, see section 4). For each object type that we track, we can mark any points along the path where an instance of this object is (a) created and (b) accessed. This information can be stored in a *TrackedObject* data structure, which contains information on the object type, a list of the call paths where it has been created and accessed, a corresponding list of the object methods accessed in each path and (depending on the granularity of the information required) the points/positions along the path at which the objects were accessed. A diagrammatic representation of this information is shown in figure 5.

[illegible]

Reconstructing Run-time Container Services

Component based enterprise frameworks provide services to components at run-time (e.g. checking and ensuring that component boundary constraints are being met). In EJB such boundary constraints include security restrictions and transactional isolation checks. For example, an EJB component method may have the following transactional attribute: (transaction) *Required* (i.e. the method will participate in the client's transaction, or, if a client transaction is not supplied, a new transaction will be started for that method). Such attributes are defined during deployment, (specified in the XML deployment descriptor) and do not change during run-time. By analysing the different component attributes, along with run-time paths, it is possible to reconstruct the different container checks as they occur along the paths. For example, by analysing the transactional attributes of each component method along a particular run-time path, one can easily reconstruct the transactional boundaries (i.e. where transactions begin and end) along the path. This information can be used by developers to easily identify how the container services are being used by their application. Since a high proportion of the application run-time can be spent executing such container services [23] it is important that the services are utilised in an efficient manner. Inefficient use of such services can lead to well known antipatterns (e.g. the large transaction antipattern [5]). A *RunTimeContainerService* data structure is created during analysis which contains information in relation to the reconstructed services. The information includes the service type, the path id in which the service occurred and the service start and end points along the path, as well as the methods that make use of the service.

Automatically Identifying Communication Patterns

What is not clear from the class diagram in figure 6 is the communication patterns or frequency of calls between the different components in the system. This type of information is often required when trying to identify particular performance issues in the application. It is important to be able to identify parts of the application where there are high or unusual levels of communication between components as knowledge of such patterns can lead to opportunities for optimizations (see section 7). By applying techniques from the field of data mining we can automatically identify such patterns in the run-time paths.

Frequent Sequence Mining

Data mining techniques such as frequent itemset mining [24] have been traditionally applied to market basket analysis to identify relationships between items that tend to occur together in consumers' shopping baskets. Consumers' baskets can be represented as unordered transactions of items, and items that consistently occur

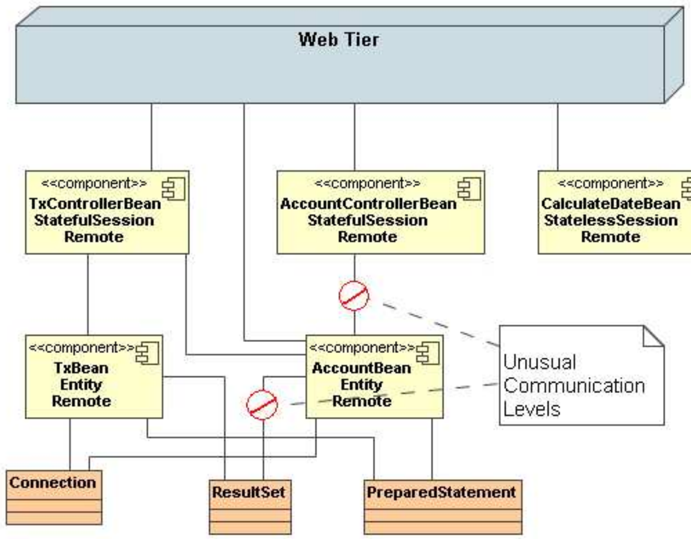

Web Tier
<<component>> TxControllerBean StatefulSession Remote
<<component>> AccountControllerBean StatefulSession Remote
<<component>> CalculateDateBean StatelessSession Remote
<<component>> TxBean Entity Remote
<<component>> AccountBean Entity Remote
Unusual Communication Levels
Connection
ResultSet
PreparedStatement

Figure 7: Class Diagram of a modified version of Dukes Bank With Communication Patterns Highlighted

together across the transactions can be identified[3]. This allows for improved marketing campaigns and product placement strategies. Similarly this type of analysis can be applied to run-time paths to identify patterns of method calls that consistently occur. As with consumers' shopping baskets, run-time paths can be represented as transactions in a transactional database. Unlike shopping baskets, which do not maintain an order on the items within them, run-time paths contain the ordered sequence of events. Frequent itemset mining does not respect this order and thus the patterns it identifies are unordered patterns.

Frequent sequence mining (FSM) [26] is a direct generalisation of frequent itemset mining and is more suitable for the analysis of run-time paths since it respects the order within them. We have recently applied FSM to identify frequently occurring sequences of method calls that occur across run-time paths [27]. The mining process can identify the most common sequences of method calls within an ordered transactional database. It has been shown how this technique can be utilised to identify frequently repeating loops within run-time paths [27]. In situations where the run-time paths are augmented with performance metrics, we have shown that the mining process can be weighted to take these metrics into account, and thus identify the most resource intensive frequent sequences (e.g. resource intensive loops) within the data [27]. It has also been shown that identifying such patterns in run-time paths allows for quick (manual) identification of design flaws in large enterprise applications [27]. In section 7 we show how this information can also be used for automatic identification of performance antipatterns. Figure 7 shows a UML class diagram

[3]Transactional data in the data mining context refers to a database of transactional records. For example a database of different customer shopping transactions on a given day (known as market basket data) or a database of individuals banking transactions. It is important not to confuse a transaction in the data mining context with the meaning of a transaction in the JEE context [25].

[illegible]

enhanced with information pertaining to identified sequences within the run-time paths. A *FrequentSequence* data structure is created during analysis which contains information relating to the frequent sequences identified in the run-time paths. The data structure contains the path id's of the sequence (i.e. in what paths the sequence occurs), the items (i.e. the component methods) that make up the sequence, the parents of the sequence, the support of the sequence (i.e. how often the sequence occurs) and the sequence confidence [28] (i.e. the accuracy of the sequence). The support of the sequence can be broken down further to reflect how often it occurs in the different run-time paths.

Data Reduction

The amount of data produced during monitoring large scale enterprise applications is often too large for easy human analysis. Thus it is required that the data be reduced and summarised such that it can be presented to developers and system testers in a more meaningful format. Summarising the data also allows for the data to be more easily integrated with a run-time design model.

Clustering Run-time Paths

Run-time paths collected from enterprise applications can often be too long or too many for easy human analysis. Considering the number of software components and sub components in an typical enterprise application (this can be in the order of hundreds), the number of paths that can be taken through the system is generally quite large. Similarly the length of a run-time path corresponding to a particular user action can also be very long (since a large number of component method calls may be needed to service a user request). The issue of path lengths is somewhat addressed through frequent sequence mining, since repeating sequences in a given path can be identified and represented in a more concise manner (see the more concise representation of loops in figure 5). To address the issue of having a large number of unique paths we can apply another data mining technique called clustering [28]. Clustering is the classification of similar objects into different groups, or more precisely, the partitioning of a data set into subsets (clusters), so that the data in each subset share some common traits.

Although there can be a large number of unique paths through the system many of these paths can be very similar and may be constructed of many of the same sub-paths. Using very basic clustering analysis we can reduce the number of paths significantly into common path groups. In section 7 we show how this can be achieved for run-time paths collected from monitoring a JEE system, when we cluster paths together that (a) call the same component methods and (b) make use of the same components. During analysis we create a *Cluster* data structure that contains all run-time paths that belong to a particular cluster. This data structure also contains information relating to the clustering criteria (e.g. for (a) above the data structure contains the list of component methods that are called by the run-time paths in a particular cluster). The cluster data structure can be added to the design model. This allows a developer to see (a summary of) the different paths that are taken through the system (e.g. at a method or component level), without the need to analyse all run-time paths recorded.

Statistical Analysis

Statistical analysis is used to summarise resource usage and server resource information. For example for each method we can get the average methods response time/CPU usage/memory usage, maximum and minimum values, as well as the standard deviation. The same analysis can be applied to queues for server resources, e.g. object pool sizes, database connection pools etc. The data structures created during analysis can be enhanced with statistical values for the component methods/server resources that they contain. The statistical values are calculated by analysing the performance metrics collected with the run-time paths and by analysing the server resource usage information that is collected.

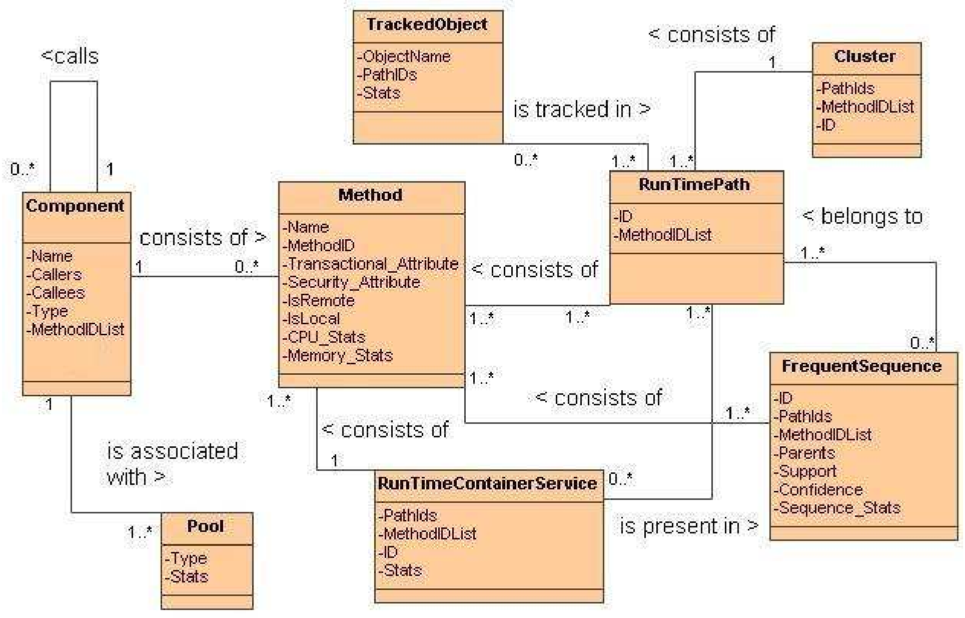


Figure 8: Run-Time Design Meta Model

The Reconstructed Design Model

The output from the analysis module is a detailed design model that captures the relationships and patterns identified during analysis, as well as the reconstructed container services, path clusters and statistical information. We call this a run-time design model since it captures a snapshot of system design decisions that are realised as the system executes. Figure 8 gives a diagram of the different data structures that are contained in the design model. Figure 8 contains 8 entities with the following relationships: A *Component* entity can call zero or more other *Components*, is made up of zero or more component *Methods* and can be associated with one or more object *Pools*. A *RunTimePath* entity is made up of one or more component *Method* calls. The *RunTimePath* can contain zero or more *FrequentSequences* (made up of a sequence of one or more *Methods*), the *FrequentSequences* can in turn belong to 1 or more *RunTimePaths*. Likewise a *RunTimePath* can contain zero or more *TrackedObjects* and *TrackedObjects* can be tracked in one or more *RunTimePaths*. A *RunTimePath* can belong to a single *Cluster* and a *Cluster* is made up of one or more *RunTimePaths*. Finally, a *RunTimeContainerService* consists of one or more *Method* calls within particular service boundary constraints (determined by meta-data) and can be present in one or more *RunTimePaths*.

The extracted design model gives two main views into the system. A transactional view and a hierarchical view. The transactional (*RunTimePath*) view of the system shows the main paths (*Clusters*) through the system, the most frequent communication patterns along these paths and how particular (*Tracked*) data objects are being created and used along the transactions. From the transactional design view one can also determine the container level services that are used along the different transactions. A more abstract hierarchical view of the system can also be obtained from the design model, by analysing the *Component* details and the component callers and callees. This shows the different component types and relationships that make up the system. The model also shows how the components are being made available by the underlying container through the different container pools.

6 DETECTION

The detection module is responsible for identifying performance design and deployment antipatterns in the extracted design model. Detection can be achieved through the use of a knowledge base or rule engine. Our prototype PAD tool makes use of the JESS rule engine [29] for this purpose. JESS is a Java based rule engine, whereby information can be loaded into the rule engine in the form of JESS facts. Rules can be defined to match a number of facts in the knowledge base. If a rule matches the required facts, the rule fires. When a rule fires, particular tasks can be performed. Such tasks are outlined in the rule definition. The information captured in the extracted design model can be easily loaded into the rule engine by converting the instances of the model entities to JESS facts. JESS will accept java objects as input and automatically convert them to JESS facts. During analysis instances of the model entities are created in the form of java objects. The objects created are loaded into JESS by the detection module. To detect antipatterns in the design, rules that describe the antipatterns must be written and loaded into the rule engine.

Antipatterns Categories

In this subsection we categorise the antipatterns, that we detect, based on the similarities in the type of data used to detect them. In the following subsection we give examples of rules that can be used to detect antipatterns from a number of the different categories.

1. **Antipatterns Across or Within Run-Time Paths:** In order to detect the antipatterns in this category, information is required on how often particular components or services occur within the same, or across the different run-time paths. For example, a large number of session beans occurring across the different run-time paths may signify the existence of the Sessions-A-Plenty antipattern [5] (i.e. the overzealous use of session beans, even when they are not required). Another example of an antipattern in this category is the Large/Small transaction [5] antipattern whereby an inefficient transaction size is set resulting in either very large long living transactions or many inefficient short living transactions.

2. **Inter-Component Relationship Antipatterns:** The inter-component relationship antipatterns can be identified by analysing the relationships that exist between the different components in the system. Where inappropriate relationships exist antipatterns can be detected. A typical example might be the Customers-In-The-Kitchen Antipattern [4] where web tier components directly access persistent objects (e.g. Entity Beans). Other antipatterns that can be detected by analysing inter-component relationships include the Needless Session Antipattern (described in the following subsection), the Transparent Facade Antipattern [5] and the Bloated Session Antipattern [5].

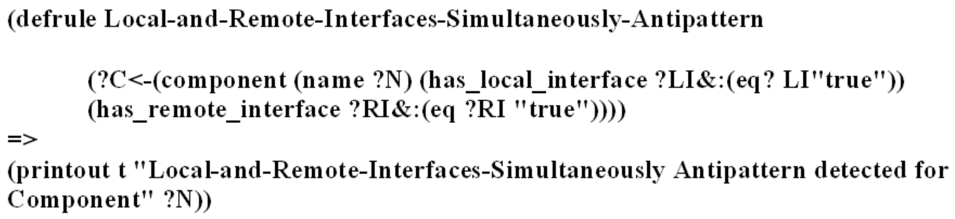
3. **Antipatterns Related to Component Communication Patterns:** This category of antipatterns can be identified by analysing the communication patterns between particular component types. For example a high level of fine grained chattiness between remote components (i.e. the Fine Grained Remote Calls or Face Off [4] Antipattern). Another example might be an unusually bulky or high amount of communication between the business tier and the database which could signify the existence of an Application Filter/Join Antipattern [4]. Other examples include the Eager Iterator Antipattern [4].

4. **Data Tracking Antipatterns:** The data tracking category of antipatterns can be detected by analysing how data objects are created and used across particular run-time paths. A typical example of this antipattern is an unused data object, whereby a data object is created and passed along the run-time path, but the information which it contains is never accessed or used. Another variation on this antipattern is called aggressive loading of entity beans, whereby an entity bean is loaded but only a small proportion of the persistent data it contains is ever accessed [30].

[illegible]

5. **Pooling Antipatterns:** Antipatterns in this category can be detected by analysing the related pool size and queue information. For example, to determine if an inefficient pool size has been specified we need to consider the pool size, the number of object instances being used on average and the average queue length. Examples of these antipatterns include inefficient configuration of any of the container pools (thread pools, session bean pools, entity bean pools, database connection pools etc.) [30].

6. **Intra-Component Antipatterns:** Intra-component antipatterns can be detected by analysing internal component details. Examples include the Simultaneous Remote and Local Interfaces Antipattern [4] or the Rusty Keys antipattern [4]. In both these cases the antipattern can be identified by analysing the component meta data.



```
(defrule Local-and-Remote-Interfaces-Simultaneously-Antipattern

        (?C<-(component (name ?N) (has_local_interface ?LI&:(eq? LI"true"))
        (has_remote_interface ?RI&:(eq ?RI "true"))))
=>
(printout t "Local-and-Remote-Interfaces-Simultaneously Antipattern detected for
Component" ?N))
```

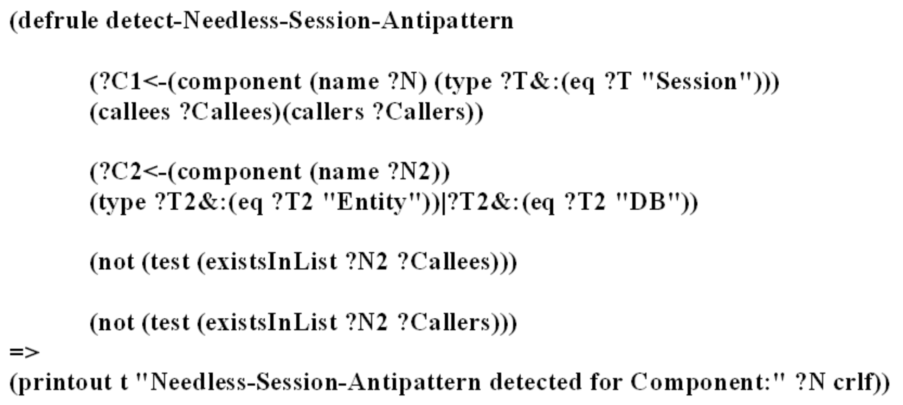
Figure 9: Rule to Detect Simultaneous Interfaces Antipattern

Example Rules

Next we give examples of how we specified antipattern rules for a number of different antipatterns from the categories above. The rules given have been used to detect instances of antipatterns in JEE applications as shown in section 7. JESS rules are written in a Lisp like syntax [31]. A rule has two parts separated by the following sign: =>. The left hand side (LHS) of the rule consists of patterns that match facts. The right hand side (RHS) gives the functions to be executed when the pattern on the LHS is matched. The RHS of the rules shown in this section consists of a function call to the *printAPSolution* function. This function prints the antipattern description, solution and corresponding contextual information for the particular antipattern detected.

The rule in figure 9 describes an antipattern from the intra-component antipatterns category. The antipattern described is the Local and Remote Interfaces Simultaneously antipattern, whereby a component exposes its business methods through both local and remote interfaces. The detection of this antipattern is quite simple since it requires the matching of only one fact i.e. is there a component fact that has the value "true" for both attributes "has_local_interface" and "has_remote_intreface".

The rule shown in figure 10 is from the inter-component relationship antipatterns category. It describes a situation where a session bean has been used but was not



```
(defrule detect-Needless-Session-Antipattern

        (?C1<-(component (name ?N) (type ?T&:(eq ?T "Session")))
        (callees ?Callees)(callers ?Callers))

        (?C2<-(component (name ?N2))
        (type ?T2&:(eq ?T2 "Entity"))|?T2&:(eq ?T2 "DB"))

        (not (test (existsInList ?N2 ?Callees)))

        (not (test (existsInList ?N2 ?Callers)))
=>
(printout t "Needless-Session-Antipattern detected for Component:" ?N crlf))
```

Figure 10: Rule to Detect Needless Session Antipattern

required. In general a session bean is generally only required if there is interaction with the persistent tier (e.g. entity beans or the database components) or if other container services are required. Otherwise a plain old java objects, which is less resource intensive, can be used. To identify this antipattern we try to identify session beans that exist but do not have any relationships with entity or database tier components. Further checks can be made to identify the use of container services. However, we have found that in many situations container services are used by sessions when not required (e.g. setting transaction attributes to "Required" by default), so instead in the rule below we check only for (persistent) component relationships. The rule in figure 10 (a) checks for a session bean component, C1 that has a list of caller and callee components, (b) checks for a second component C2, that is either an entity bean or a database component and (c) checks if C2 is a caller or callee of C1. JESS allows for the use of user defined functions which can be used to provide more complex functionality to the rules in a concise manner [31]. The *existsInList* function in figure 10 is a user defined function which checks a list (argument 2) for a particular value (argument 1). Without the use of such functions the rules can become overly complex and difficult to both write and comprehend. The PAD tool provides a number of user defined JESS functions to allow for the easy construction of rules.

The final rule example given in this section (figure 11) is a rule from the antipattern category concerned with component communication. In this rule we identify a relationship between a session bean and an entity bean in the form of a frequently repeating sequence (which may be present in the case of the Application Filter antipattern, for example). If this relationship exists, the average resource consumption of the frequent sequence is calculated and is flagged if it is above a user defined threshold. The calculation of the resources consumed is performed by the Jess user defined *flagHighResourceConsumption* function, which is passed the list of methods in the sequence and the support of the sequence. The function refers to a user defined configuration file which specifies the acceptable threshold values. Alternatively if performance metrics are unavailable the frequency of the sequence alone can be used to identify the antipattern.

[illegible]

```
(defrule detect-Bulky-Session-Entity-Communication

        (?C1<-(component (name ?N1)(type ?T1&:(eq ?T1 "Session")) (callees
        ?Callees)))

        (?C2<-(component (name ?N2)(type ?T2&:(eq ?T2 "Entity")))

        (?FS<-(frequentSequence(children ?Ch) (parents ?Ps)
        (methodIdLists ?Ms) (support ?S)))

        (test(existsInList ?N1 ?Ch))

        (test(existsInList ?N2 ?Ps))

        (test(flagHighResourceConsumption "frequentSequence" ?Ms ?S)
=>
(printout t "Bulky- Communication Detected when" ?N1 " calls " ?N2 crlf))
```

Figure 11: Rule to Detect Bulky Database Communication

7 RESULTS

In this section we show how the PAD tool was applied to two JEE applications to identify a number of performance design and deployment antipatterns. The first of these applications is a sample application from Sun Microsystems called Duke's Bank [32] which is freely available for download. Sun have used the Duke's Bank application to showcase the JEE technology. The other application we tested is a beta version of the IBM Workplace application, which is a real large scale enterprise system [33]. Antipatterns from all the categories outlined in section 6 have been detected. For each antipattern detected we give a brief description of the antipattern and the antipattern category. We also give the related information (PAD output) which is presented to the tool user upon detection. Using this information the tool user can easily determine the severity of the problem and a decision can be made as to whether refactoring is required or not. We do not show performance improvements that can be attained by refactoring the antipatterns detected since these improvements have already been well documented [4] [5]. Also performance improvements can be very much application specific and vary greatly depending on the severity of the antipattern.

PAD Tool User Modes and Data Reduction Results

The PAD tool can be used in two different monitoring modes. Either single user mode or multi user mode. Single user means that there is only one user in the system during monitoring (e.g. a single developer testing the application). Multi user mode means that the system is loaded with multiple simultaneous users. Antipatterns from categories 1,2,3,4 and 6 can be detected in single user mode. All antipatterns can be detected in multi-user mode and in fact this mode is required to detect the antipatterns in category 5. An added advantage of using multi user mode is that accurate performance metrics can be given to the tool user on detection of an antipattern. Such metrics can be used by the tool user to quickly assess the impact of the detected pattern. Performance metrics can also be collected during single user mode, however they are less reliable since the system is most likely being used in a less realistic manner.

There are two main drawbacks of using multi user mode however. Firstly, it requires a load to be generated on the system. In most cases this requires the creation of an automated test environment (e.g. using load generation tools). Secondly, a large amount of data is produced when monitoring enterprise systems under load. In particular, our monitoring module produces a large amount of run-time path information. This issue can be addressed however by applying the clustering and statistical analysis techniques outlined in section 5. To show the effectiveness of these data reduction techniques we have applied them to data collected from a JEE application under load. For this test we loaded the Duke's Bank sample e-commerce application with 40 users for a five minute period. Each user logged onto the system, browsed their account information, and deposited funds onto different accounts. In total each user performed 8 different user actions. A total of 1081 run-time paths were collected during this period. To reduce the data produced we clustered the paths (a) by the component methods that were invoked in each path and (b) by the different components that were invoked in each path. After applying clustering criteria (a) we grouped the paths into 11 different path clusters. That is, our cluster analysis reduced the 1081 paths recorded to 11 (component-method level) paths through the system. In this instance statistical analysis can be applied to the component methods contained in each cluster to give a summary of the performance metrics associated with the call paths in each cluster. Applying the single user mode approach to the same user actions results in 11 distinct call paths. Our results show that (in this instance) applying clustering analysis to data collected in multi user mode can effectively reduce the number of distinct path clusters to the number of different paths observed in single user mode. The path clusters in multi user mode in fact contain more useful information than the paths collected in single user mode, since they give more realistic performance metrics for each method that is invoked in the path. Applying clustering criteria (b) to the 1081 paths resulted in 8 path clusters. That is, at the more abstract component level, there were 8 different paths through the system.

Antipatterns Detected in the Duke's Bank Application

The first application we applied the PAD tool to, in order to identify performance design and deployment antipatterns, was Duke's Bank. Duke's Bank is an online banking application. When a user logs in to the Duke's Bank application he/she can perform the following actions: log on, view a list of accounts, view an individual account's details, withdraw or lodge cash, transfer cash from one account to another

[illegible]

or finally log off. For our tests each of the different (8) user actions was performed. COMPAS BCI was used to collect run-time paths, related performance information and to perform object tracking. Duke's Bank was deployed on the JBoss application server (version 3.2.7) with a MySQL database (version 4.0.2) as the backend. Our MEJB monitoring tool was used to interface with the application server to collect information on the server resources. For multi user mode (which was required to identify the Incorrect Pool Size antipattern) the open source Apache JMeter load generation tool was used to load the application. Two versions of dukes bank were tested, the original version and a modified version with a number of antipatterns added. The original dukes bank application consists of 6 EJBs (4 of these were invoked during the tests, see figure 6). We also modified the original version of Duke's Bank to add a number of antipatterns to be detected by the PAD tool such that antipatterns from all categories discussed in section 6 were present (see figure 7 for a class diagram of the modified version of dukes bank). The antipatterns introduced are described in detail below. In total 3 antipatterns were detected in the original version of Duke's Bank by the PAD tool:

- **Conversational Baggage Antipattern** [4] **(category 1)**: This antipattern describes a situation where stateful sessions beans are being used but are not necessarily required. Stateful session beans maintain state on the application server between client requests and should be used only when there is a clear need to do so. Stateless session beans will scale better than stateful session beans and should be used when state does not need to be maintained. Detection of this antipattern involves flagging the creation of unusually high numbers of stateful session beans across the run-time paths. A potential instance of this antipattern was flagged by the PAD tool when Dukes Bank was analysed as the number of stateful session beans was above the user defined threshold. This threshold was set to zero for the dukes bank application since there is no noticeable state maintained from one user action to the next. When the potential antipattern was detected PAD showed us that stateful session beans were used in 100% of the run-time paths. On closer inspection of the application (source code) we saw that indeed the stateful sessions could have been replaced with stateless sessions to improve scalability.

- **Fine Grained Remote Calls (also known as the Face Off antipattern** [4]**) (category 3)**: This antipattern describes a situation where a number fine grained calls are consistently made from a client to the same remote bean. This results in a high number of expensive remote calls. A better approach is (if possible) to make a more coarse grained call that performs the combined tasks of the fine-grained calls. Performing a single coarse grained call instead of a number of fine grained calls will reduce network latency and thus the overall time required for the user action to execute. An instance of this antipattern was identified by the PAD tool. The tool identified a frequent sequence which contained fine grained remote calls. In fact it identified that the remote methods AccountBean.getType and AccountBean.getBalance appeared together 100% of the time they were called (i.e. the sequence had a confidence value [28] of 100%). Thus it would be more efficient to combine these calls into a single coarse grained remote call. This antipattern is far from severe in this instance, however the identification of this antipattern shows that the tool can indeed identify antipatterns in this category. The rule to identify this antipattern can in fact be modified with a user defined threshold to only flag more severe instances.

- **Remote Calls Locally (also known as Ubiquitous Distribution** [4]**) (category 2)**: This antipattern describes a situation where beans that run in the same JVM as the client are called through their remote interfaces. In this situation the client has to perform an expensive remote call even though the bean is local. While some containers can optimize in this situation, this optimization is not a standard feature of JEE. A better approach would be to write the beans with local interfaces (instead of remote, provided the beans are not also called from remote clients) such that they can be accessed in a more efficient manner. The PAD tool identified this relationship between components. In fact all (4) beans invoked in the Duke's Bank application are accessed through remote interfaces even though their clients are calling them from within the same JVM.

Next we describe the antipatterns that were added to Duke's Bank and the information given by the PAD tool when they were detected:

- **Accessing Entities Directly** [5]**/Customers In The Kitchen** [4] **Antipattern (category 2)**: The application was modified such that components in the web tier were directly accessing entity bean components (see figure 7). This antipattern can cause a number of different performance issues as documented in the antipattern description (e.g. problems with transaction management). Furthermore it can create maintainability issues since it mixes presentation and business logic. The PAD tool identifies the inappropriate component relationships and flags them as antipattern instances. The component identified was the accountList.jsp which was modified to call the AccountBean (entity) directly.

- **Needless Session Antipattern (category 2)**: Another antipattern added was the needless session antipattern. This antipattern is described in section 6 and outlines a situation where a session bean is used but not required. In the Duke's Bank application we added a session bean that was being used to calculated the current time and date. This function could have been easily carried out by a POJO which would have been a more efficient solution. The antipattern was detected by the PAD tool which reported that the session bean (CalculateDateBean) had no relationships with any persistent components (e.g. entity beans or database components) and thus was a potential needless session bean.

[illegible]

- **Application Filter Antipattern** [4] **(category 3)**: The application filter antipattern describes a situation where large amounts of data are retrieved from the database tier and filtered in the application tier. However, databases are designed to process filters a lot faster can be done in the application tier. Thus filters should, where possible, be implemented in the database tier. We modified the Duke's Bank application to include an application filter. The dukes bank application can retrieve account information for a customer that has logged in. When this is performed the application performs a search for the accounts that are owned by the particular user. We created an application filter in this situation by modifying the SQL statement which filtered the information (using the following SQL statement `"select account_id from customer_database where customer_id = ?"`) to instead retrieve *all* accounts and send them to the application tier for filtering. In the application tier filtering of the data was performed by checking the details of each account to see if it matched the id of the customer. We applied the PAD tool to the modified version of Dukes Bank. The tool successfully identified the application filter that had been added to the application. In fact the PAD tool identified that by modifying this filtering 7/11 of run-time paths were affected. The tool identified two frequent sequences of communication: (1) between a session bean (AccountControllerBean) and an entity bean (AccountBean) (see figure 11) and (2) between the same entity bean (AccountBean) and the database in these paths. The first sequence of size 1 (AccountBean.getDetails) occurred 1180 times across the different run-time paths. The second sequence of size 2 (ResultSet.next, ResultSet.getString) occurred 6336 times across the run-time paths. It was evident from this information that an application filter existed in the application, especially when considering that this amount of activity was occurring in single user mode.

- **Unused Data Object/Aggressive Loading** [30] **(category 4)**: A second antipattern was identified when the PAD tool was applied to the dukes bank application modified with an application filter antipattern. The antipattern detected was the unused data object antipattern/aggressive loading antipattern. This antipattern describes a situation whereby information is loaded from the database, but the data (or at least a high percentage of the data) is never used. A solution to this problem can be to refactor the application such that information is not retrieved if it is never actually required. Often however the information may be required a small percentage of the time. In such circumstances the information can be lazy loaded when it is required. This antipattern occurred in the application filter above. During filtering a check is performed on each account to see if its owner id matched that of the (logged in) customer's. To obtain each account's owner id, the application loads the account information from the database into an entity bean. An AccountDetails Data Transfer Object (DTO) [34] is then created by the entity bean with the account information. Finally during the filtering the DTO is accessed to obtain the account owner id. Rather than loading all the account information into the entity bean, and subsequently into the DTO, a lazy loading approach can be used to load only the information that it generally needed (i.e. the account owner id). If the remaining (unloaded) information is required it can be loaded later. This antipattern will in fact be removed if the application filter is pushed into the database tier as suggested in the antipattern solution for the application filter. However there may be situations where it may not be possible to easily remove the application filter and where lazy loading can be applied. Lazy loading can in fact be applied in any situation where only a small proportion of the entity bean fields are ever accessed. The PAD tool provided the following information when this antipattern was identified: The AccountDetails Object was created on average 1886 times across the 7 run-time paths with a maximum value per path of 2400 times and a minimum value of 1200 times. The object has 8 accessor methods. Each method is given below with the corresponding average number of times it was accessed per run-time path: getCustomerID's 1886 times, getType 1.4 times, getDescription 3 times, getBalance 2.6 times, getAccountId 8.1 times, getCreditLine 2.3 times, getBeginBalance 0 times and getBeginBalanceCreditLine 0 times. From the PAD tool output it is evident that (if the application filter could not be removed) it would be beneficial from a performance perspective to apply lazy loading for account bean and DTO in this application to all fields except for the CustomerIds field.

- **Incorrect Thread Pool Size (category 5)**: The final antipattern added to the dukes bank application was a deployment antipattern. We redeployed the application and modified the thread pool size to 10 (from the default 1,000,000). To detect this antipattern we ran the tests in multi user load. We loaded the application with 40 users (over 10 seconds) and monitored the system for a 5 minute period. We specified the maximum passivation level as 4 (10% of user load) in (user configuration files associated with) our antipattern rules. The PAD tool detected the incorrect thread pool size antipattern and presented the following information to the tool user: The AccountControllerBean instance cache passivation levels exceeded the specified user threshold of 4. The average passivation level of the measured period was 15. In this situation increasing the size of the instance cache is recommended.

Next we discuss the issue of false positives and negatives detected by the PAD tool when applied to Duke's Bank. In the strictest sense no false positives were found during the tests i.e. the tool did not identify antipatterns instances that were not present in the system. However with performance related antipatterns we are more concerned with identifying antipatterns that have an impact on the overall system performance. It is likely that the fine grained remote calls antipattern instance would not have a significant impact on the system performance and thus might be considered a false positive in this instance. However, by modifying the user defined threshold associated with the rule to detect this antipattern we can filter out instances with a low performance impact. Our aim was to show that instances

[illegible]

of this antipattern can be identified by our tool and thus we set the threshold value such that even insignificant instances (from a performance perspective) were also identified. Similarly the remote calls locally antipattern may not have a performance impact in application servers that can optimize remote calls that are made to local components. However, again our aim was to show that this antipattern can be identified using our tool.

By studying the Duke's Bank documentation [32] and source code we were confident that our tests did not produce false negatives i.e. there were no antipatterns in the application, which were defined in our antipattern library that we did not detect.

Antipatterns Detected in the IBM Workplace Application - Beta Version

The second system tested was an early beta version of the IBM Workplace Application [33]. IBM Workplace is a collaborative enterprise application built on the JEE technology. In total 76 EJBs were instrumented, 38 of these were only ever invoked during the test runs (17 entity beans with Container Managed Persistence and 21 Session beans). The test run consisted of invoking 25 different user actions. Monitoring was performed using the COMPAS JEEM tool (COMPAS BCI was unavailable at the time of testing). As a result no object tracking information was obtained. Also performance metrics were not collected during these tests due to our limited access to the system. All tests were carried out in single user mode. The IBM Workplace application was running on the IBM WebSphere application server (version 5.x). The database used was IBM's DB2. Using the PAD tool we identified antipatterns from four of the different categories outlined in section 6:

- **Local and Remote Interfaces Simultaneously (category 6)**: This antipattern occurs when a bean exposes both local and remote interfaces. There are a number of issues associated with this antipattern including exception handling issues, security issues and performance problems. From the 76 beans instrumented the PAD tool identified 30 (session) beans that exposed both local and remote interfaces. Many of the beans identified were not invoked during the test run, however the required information for antipattern detection in this instance was available in the beans meta data. On identification of this antipattern we contacted the development team who acknowledged that this was indeed an antipattern that had been removed in a later release due to security concerns.

- **Unusual/Bulky Session-Entity Communication (category 3)**: The second antipattern type identified by the PAD tool was in relation to database communication. The tool identified unusually high communication levels between session and (persistent) entity beans (see rule in figure 11). In fact as a result of the 25 user actions, a frequent sequence of entity bean calls (from a session bean) of size 24 occurred 58 times. In fact in one particular run-time path this sequence occurred 33 times. The 24 calls in the sequence were all to the same method which suggested that the sequence was in fact a loop of size 24. From this data it seemed that there was potentially an application filter causing this issue. Again we contacted the development team responsible for this code. The development team had identified this antipattern in a later release and had rectified it by pushing the filtering into the database tier as suggested by the application filter antipattern solution.

- **Transactions-A-Plenty/Incorrect Transaction size [5](category 1)**: Another antipattern detected by the PAD tool was the transactions-a-plenty antipattern. The tool identified that for one particular use case a very high number of transactions were being created. In fact the tool identified that 131 transactions were created in a single run-time path. In fact for every session bean method call a transaction was being initiated. On further inspection we discovered that the issue was the fact that the session beans methods' transactional settings were being set to "Requires New" by default. The development team addressed this issue by editing the transactional settings for the beans in the deployment descriptors to initiate transactions only where new transactions were actually required.
- **Bloated Session Bean Antipattern [5](category 2)**: The final antipattern type detected was the bloated session bean antipattern. This antipattern is similar to the well known God class antipattern [7]. It describes a situation in EJB systems where a session bean has become too bulky. Such session beans generally implement methods that operate against a large number of abstractions. For example one method may check the balance of an account while another may approve an order, and yet another may apply a payment to an invoice. Such sessions can create memory and latency issues since their creation and management can come with much overhead due to their bulky nature. The PAD tool identified two potential bloated session beans in the IBM Workplace application. The first potential instance of this antipattern was a session bean which had 8 entity bean relationships, 6 session bean relationships and was invoked in 11 of the 25 user actions executed. It also contained a high number of business methods (47). The second instance of this antipattern was a session bean which had 7 entity relationships 1 session relationship and was invoked in 6 of the 25 user actions. This session had 14 business methods. In general the rule of thumb is that there should be a 1 to 1 relationship between session and entity beans [5]. From the information (above), presented by the PAD tool upon identification of these potential antipattern instances, it seemed that these session beans were in fact bloated session beans. Unfortunately we were unable to contact the developers originally responsible for this code. However we did discover that this code was removed from later releases of the application which indicated that it was indeed problematic.

No false positives were identified when we applied the PAD tool to the IBM workplace application and in fact all antipatterns identified were addressed in the subsequent versions of the application which suggested they were indeed problematic pieces of code. Unfortunately we could not assess whether false negatives were identified in the application as we did not have access to the complete system source code or documentation.

8 RELATED WORK

Antipatterns have been previously documented and categorised in a range of different literature [7] [6] [8] [4] [5]. Technology independent software antipatterns have been previously documented in the literature [7] and [6]. Smith and Williams [6] focus on technology independent performance antipatterns in particular. Technology specific antipatterns have been documented in [8], [4] and [5]. The literature [4] and [5] both categorise their antipatterns according to the related component types which are effected. For example, antipatterns related to entity beans, antipatterns related to session beans etc. In contrast we have taken technology specific (JEE) performance antipatterns and categorised them according to the data required to detect them. A similar approach has previously been taken by Reimer et al., who have categorised programming errors based on the algorithms used to detect them [35]. Our antipattern categorisation also differentiates between performance design antipatterns, performance deployment antipatterns and performance programming errors. Similarly, Moha and Gueheneuc [36] provide a taxonomy in which they attempt to clarify the difference between errors, antipatterns, design defects and code smells. In their analysis they define code smells as intra-class defects, design defects as inter-class defects. Our antipattern categorisation is at a higher more abstract component level. Hallal et. al. [37] provide a library of antipatterns for multi-threaded java applications. They also distinguish between errors and design antipatterns. They classify the mutli-threaded antipatterns, that they present, following a categorization which reflects the effect of the antipatterns on the outcome of the tested program.

There has been much work in the area of reverse engineering applications to extract the application design. Many of these approaches (e.g. [38] [39] [40]) work by analysing the source code of the application (or the bytecode) and create static models of the system. A drawback of using static models is that they contain all potential relationships that may exist in the system. For performance analysis many of these potential relationships may never be relevant. The PTIDEJ tool [41] makes use of both static and dynamic models to construct detailed class diagrams that contain inheritance, instantiation, use, association, aggregation, and composition relationships. Our reverse engineering approach works at a higher (component) level of abstraction and contains run-time relationships only. Chen et al. [42] had previously shown how such dynamic component relationships (or models) can be extracted from run-time paths for the purpose of problem determination using the pinpoint tool. Our monitoring approach is an extension of the pinpoint tracing tool [18]. In contrast to pinpoint it is completely portable and also provides for object tracking [19]. Briand et. al have previously presented work on reverse engineering sequence diagrams from distributed [43] and multi-threaded [44] java applications. Their approach, similar to our analysis module, is based on populating instances of a meta model with information collected by monitoring a running system. The literature [45] similarly presents an approach for architecture recovery using runtime analysis. With this approach a run-time engine takes a mapping specification and monitoring events (from a running system) as input, and subsequently produces architecture events. A drawback of this approach is that it requires an engineer to create the mapping specification between the low level events recorded and the architecture events that are produced. An alternative approach for the identification of run-time relationships has been suggested by Agarwal et al. [46]. They use a data mining approach to extract resource dependencies from monitoring data. Their approach relies on the assumption that most system vendors provide a degree of built in instrumentation for monitoring. A major drawback of this approach however is that it is statistical and not exact, and at higher loads the number of false dependencies increase significantly.

Data mining techniques have been previously applied to run-time paths for the purpose of problem determination using clustering and statistical analysis to correlate the failure of requests to the components most likely to have caused them [42]. We make use of clustering for the purposes of data reduction. Clustering has previously been used in a wide range of fields. A comprehensive survey of current clustering techniques can be found in the literature [47]. Similarly frequent itemset mining algorithms have been used in many different domains [28]. However, we believe we are the first to apply FSM to run-time paths to identify communication patterns in enterprise applications.

There has been much research in the area of detecting low level programming errors or bugs in software systems (e.g. [35] [48] [49] [50] [51] [52]). Current performance tools also focus on this area of programming errors and provide views that assist in the identification of memory leaks and deadlocks (e.g. [14]). Problems detected should ideally be annotated with descriptions of the issue detected as well as a solution that can be applied to alleviate the problem. For example the Smart Analysis Error Reduction Tool (SABER) [35] used for programming error detection provides supporting information which explains why the code is defective. It also provides contextual path and data flow information which can explain how the defect occurred. Our work concentrates on higher level instances of inefficient design [53] [16]. The detection of higher level design antipatterns has not been so widely addressed. A recent commercial tool, eoSense [54] has been developed to identify general JEE antipatterns. This tool identifies a number of JEE antipatterns and presents the user with possible solutions. This tool, similar to our approach, extracts a run-time model from a running system [55] and has the ability to identify performance related issues (e.g. bulky or excessive communication). However, the tool has been designed to be used in single user mode and thus does not perform data reduction when monitoring applications under load.

[illegible]

While antipattern detection is a relatively recent research topic, there has already been a significant effort in the area of detecting software design patterns. Most of these detection approaches have relied on static analysis [56] [57], or a combination of static and dynamic analysis [58] [59]. Using static analysis is unsuitable for detection or design recovery in large enterprise systems since the number of potential relationships can be extremely large if there are a large number of components in the system. A further advantage of dynamic analysis over static analysis is that it allows for the collection of performance metrics. Our approach uses run-time data and does not perform static analysis on the source code (or bytecode) of the application. A more indepth discussion on related work is given in the literature [11].

9 CONCLUSIONS

In this paper we outline an approach for the automatic detection of performance design and deployment antipatterns. We discuss a number of advanced monitoring and analysis techniques that are required for our antipattern detection approach. Furthermore we categorise the antipatterns we detect into groups, according to the data required to detect them. We show how the approach can be applied to enterprise applications using our PAD tool. Using the tool we identified a number of antipatterns in both a sample application from Sun and a real enterprise application from IBM as presented in our results section. We also show how monitoring information collected from a system under load can be reduced using data reduction techniques.

As part of our future work we intend to automatically assess the performance impact of the detected antipatterns such that developers can concentrate their efforts on refactoring the antipatterns with the highest performance impact. It is also expected to apply this approach to alternative component frameworks.

10 ACKNOWLEDGEMENTS

The authors would like to thank IBM and in particular Patrick O'Sullivan and Simon Pizzoli for their help with this work and for allowing us access to their systems. Our work is funded under the Commercialisation Fund from the Informatics Research Initiative of Enterprise Ireland.

REFERENCES

[1] Szyperski C., Gruntz D. and Murer S.: "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, November, 2002.

[2] Noel J., ”J2EE Lessons Learned”, SoftwareMag.com, http://www.softwaremag.com/L.cfm?doc=2006-01/2006-01j2ee, accessed February, 2008.

[3] Roehm B. , Csepregi-Horvath B. , Gao P., Hikade T., Holecy M., Hyland T., Satoh N., Rana R. and Wang. H. ”IBM WebSphere V5.1 Performance, Scalability, and High Availability WebSphere Handbook Series”, June, 2004, http://www.ibm.com/redbooks, accessed February, 2008.

[4] Tate B., Clarke M., Lee B. and Linskey P.: “Bitter EJB”, Manning, 2003.

[5] Dudney B. et al.: ”J2EE Antipatterns”, Wiley, 2003.

[6] Smith C. U. and Williams. L. *”Performance Solutions”*. Addison Wesley, 2002.

[7] Brown W. J., Malveau R. C. and Mowbray T. J.:”AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis”, Wiley, 1998.

[8] Tate B.:”Bitter Java”, Manning Publications Co., 2002.

[9] Gamma E. and Helm R. and Johnson R. and Vlissides J.: ”Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.

[10] The Hillside Group, Pattern Definitions, http://www.hillside.net/patterns/definition.html, accessed February, 2008.

[11] Parsons T., ”Automatic Detection of Performance Design and Deployment Antipatterns in Component Based Enterprise Systems”. Ph.D. Thesis, 2007, University College Dublin.

[12] The Java Virtual Machine Profiler Interface, http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html, accessed February, 2008.

[13] The Java Virtual Machine Tools Interface, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html, accessed February, 2008.

[14] Quest Software, JProbe profiler, http://www.quest.com/jprobe/, accessed February, 2008.

[15] Jerding D.F., Stasko J.T. and Ball T. ”Visualizing Interactions in Program Executions”. In the proceedings of the International Conference on Software Engineering, 1997.

[illegible]

[16] Parsons T. and Murphy J.: "The 2nd International Middleware Doctoral Symposium: Detecting Performance Antipatterns in Component-Based Enterprise Systems", IEEE Distributed Systems Online, vol. 7, no. 3, March, 2006.

[17] Chen M., Kiciman E., Accardi A., Fox A. and Brewer E.: "Using runtime paths for macro analysis", Proc. 9th Workshop on Hot Topics in Operating Systems, Lihue, HI, USA, May 2003.

[18] Parsons T., Mos A. and Murphy M.,:"Non-Intrusive End to End Runtime Path Tracing for J2EE Systems", IEE Proceedings Software, August, 2006.

[19] Bergin J. and Murphy L., "Reducing runtime complexity of long-running application services via dynamic profiling and dynamic bytecode adaptation for improved quality of service", Proceedings of the 2007 workshop on Automating service quality, 2007, Atlanta, Georgia, USA.

[20] The Enterprise Java Bean Specification, http://java.sun.com/products/ejb/docs.html, accessed February, 2008.

[21] The J2EE Management Specification, http://www.jcp.org/en/jsr/detail?id=77, accessed February, 2008.

[22] The Java Management Extensions technology, http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/, accessed February, 2008.

[23] Ammons, G., Choi, J.D., Gupta, M. and Swamy,N: "Finding and Removing Performance Bottlenecks in Large Systems", In Proceedings of ECOOP, 2004.

[24] Agrawal R., Mannila H., Srikant R., Toivonen H. and Verkamo A.I.: "Fast discovery of association rules". In Advances in Knowledge Discovery and Data Mining, 1996.

[25] E. Roman, Scott W. Ambler and Tyler Jewell, *"Mastering Enterprise JavaBeans"*, second edition, J.Wiley and Sons, USA and Canada, 2002.

[26] Agrawal R. and Srikant R.: "Mining sequential patterns". In P. S. Yu and A. L. P. Chen, editors, Proceedings 11th International Conference in Data Engineering, 1995.

[27] Parsons T., Murphy M. and O'Sulivan, P.: "Applying Frequent Sequence Mining to Identify Design Flaws in Enterprise Software Systems", In Proceedings 5th International Conference on Machine Learning and Data Mining (poster track), Leipzig, Germany, 2007.

[28] Hand D., Mannila, H., and Smyth P.: "Principles of Data Mining". MIT Press, 2001.

[29] JESS, http://www.jessrules.com/jess/index.shtml, accessed February, 2008.

[30] Precise Java, http://www.precisejava.com/, accessed February, 2008.

[31] E. Friedman-Hill. *Jess in Action*. Manning Publications, July, 2003.

[32] http://java.sun.com/javaee/5/docs/tutorial/doc/, accessed February, 2008.

[33] http://www.ibm.com/software/workplace, accessed February, 2008.

[34] Alur D., Crupi J. and Malks D.: "Core J2EE Patterns: Best Practices and Design Strategies", Prentice Hall, Sun Microsystems Press, 2001.

[35] Reimer, D., et al.: "SABER: Smart Analysis Based Error Reduction", Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis, 2004.

[36] Moha N. and Gueheneuc Y.G.: "On the Automatic Detection and Correction of Design Defects". In Serge Demeyer, Kim Mens, Roel Wuyts, and Stephane Ducasse, editors, Proceedings of the 6th ECOOP workshop on Object-Oriented Reengineering, July 2005.

[37] Hallal H. H., Alikacem E., Tunney W. P., Boroday S., and Petrenko A.. *"Antipattern-Based Detection of Deficiencies in Java Multithreaded Software"*, Proceedings of the Quality Software, Fourth International Conference on (QSIC'04), IEEE Computer Society, USA, 2004.

[38] Murphy G.C., Notkin D., and Sullivan K.. *"Software Reflexion Models: Bridging the Gap between Source and High-Level Models"*. Proceedings SIGSOFT Symposium on Foundations of Software Engineering, ACM Press, New York, 1995.

[39] Jackson, D. and Waingold, A. "Lightweight extraction of object models from bytecode". In David Garlan and Jeff Kramer, editors, Proceedings of the 21st International Conference on Software Engineering, May, 1999.

[40] Korn, J., Chen, Y.F., and Koutsofios, E.: "Chava: Reverse engineering and tracking of Java applets". In Proceedings of the 6th Working Conference on Reverse Engineering, IEEE Computer Society Press, 1999.

[41] Gueheneuc, Y.G.: "A Reverse Engineering Tool for Precise Class Diagrams". In Janice Singer and Hanan Lutfiyya, editors, Proceedings of the 14th IBM Centers for Advanced Studies Conference, ACM Press, October, 2004.

[illegible]

[42] Chen M., Kiciman E., Fratkin E., Fox A. and Brewer E.: ”Pinpoint: Problem Determination in Large, Dynamic, Internet Services”, Proc. Int. Conf. on Dependable Systems and Networks (IPDS Track), Washington, D.C., June, 2002.

[43] Briand L.C., Labiche Y. and Leduc J., *”Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software.”* IEEE Transactions on Software Engineering, vol. 32, no. 9, September, 2006.

[44] Briand L.C., Labiche Y. and Leduc J., *”Towards the Reverse Engineering of UML Sequence Diagrams for Distributed, Multi-threaded Java Software”*. Technical Report SCE-04-04, Carleton Univ., http://www.sce.carleton.ca/Squall, September, 2004.

[45] Schmerl B., Aldrich J., Garlan D., Kazman R., and Yan H., *”Discovering Architectures from Running Systems”*. IEEE Transactions on Software Engineering, July, 2006.

[46] Agarwal M. K., Gupta M., Kar G., Neogi A. and Sailer A.:”Mining Activity Data for Dynamic Dependency Discovery in e-Business Systems”, IEEE eTransactionson Network and Service Management Journal, Vol.1 No.2, September, 2004.

[47] Berkhin. P., *”Survey of clustering data mining techniques”*. Technical report, Accrue Software, San Jose, CA, 2002.

[48] Hovemeyer D. and Pugh W., ”Finding bugs is easy”, SIGPLAN Notices, vol. 39, no. 12, ACM Press, New York, NY, USA, 2004.

[49] Johnson, S., ”Lint, a C program checker”. In UNIX Programmer’s Supplementary Documents Volume 1 (PS1), April, 1986.

[50] Evans, D., ”Static Detection of Dynamic Memory Errors”. In Proc. of PLDI, May, 1996.

[51] Detlefs, D. L., ”An overview of the extended static checking system”. SIGSOFT Proceedings of the First Workshop on Formal Methods in Software Practice, January, 1996.

[52] Ball, T. and Rajamani, S. K., ”The SLAM project: Debugging system software via static analysis”. In Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, January, 2002.

[53] Parsons, T., ”A Framework for Detecting, Assessing and Visualizing Performance Antipatterns in Component Based Systems”. First Place at ACM SIGPLAN Student Research Competition at The 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, October, 2004.

[54] Eologic, Eosense, http://www.eologic.com/eosense.shtml, accessed February, 2008.

[55] West A. and Cruickshank G., "Derived Model Analysis: Detecting J2EE Problems Before They Happen", http://dev2dev.bea.com/pub/a/2007/07/derived-model-analysis.html, accessed February, 2008.

[56] Keller, R. et al., "Pattern-based reverse-engineering of design components". In Proceedings of the International Conference on Software Engineering, 1999.

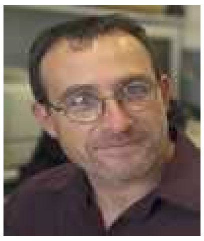
[57] [4] Kramer C. and Prechelt L., "Design recovery by automated search for structural design patterns in object-oriented software". Proc. of the 3rd Working Conference on Reverse Engineering (WCRE), Monterey, CA, November, 1996.

[58] Heuzeroth, D., Holl, T. and Lowe, W., "Combining Static and Dynamic Analyses to Detect Interaction Patterns", Proceedings of the Sixth International Conference on Integrated Design and Process Technology (IDPT), June, 2002.

[59] Wendehals L., "Improving Design Pattern Instance Recognition by Dynamic Analysis". WODA, ICSE, 2003.

ABOUT THE AUTHORS

Trevor Parsons is a post doctoral researcher in the School of Computer Science and Informatics in University College Dublin and a member of the Performance Engineering Laboratory. Contact him at trevor.parsons@ucd.ie. See also http://pel.ucd.ie/tparsons/



John Murphy is a senior lecturer in the School of Computer Science and Informatics in University College Dublin and a member of the Performance Engineering Laboratory. Contact him at j.murphy@ucd.ie. See also http://www.cs.ucd.ie/staff/jmurphy/home/