

# Data Mining for Performance Antipatterns in Component Based Systems Using Run-Time and Static Analysis

Trevor Parsons♣ and John Murphy\*

♣Performance Engineering Laboratory, Dublin City University, Ireland  
Phone: (353)-1-7007644 Email: parsonst@eeng.dcu.ie

\*Performance Engineering Laboratory, University College Dublin, Ireland  
Email: JMurphy@ucd.ie

***Abstract-*** *Current complex distributed enterprise systems with performance requirements can be difficult to design for even the most experienced system designers. Shrinking development cycles exacerbate the problem as developers are often compelled to treat performance as an afterthought, a matter of production tuning to be accomplished after the system has been coded, integrated, functionally tested and deployed. Consequently enterprise systems often suffer from severe performance problems. We propose a framework for automatically detecting and assessing the impact of poor performance design (performance antipatterns) in component based systems, using a combination of run time and static analysis. The framework consists of four modules (a monitoring module, a detection module, an assessment module and a visualization module). The framework borrows techniques from the field of Knowledge Discovery in Databases. We intend to instantiate the framework for the Enterprise Java Bean platform.*

***Keywords-*** *Components, Antipatterns, Data Mining, EJB, Dynamic Analysis*

## I. INTRODUCTION

Complex distributed enterprise systems with performance requirements can be difficult to design for even the most experienced system developers. There are multiple issues to contend with e.g. security, maintainability, reusability, transactional integrity, performance, etc., when designing such systems. Contextual component frameworks such as Enterprise Java Beans (EJB), Microsoft .Net and the CORBA Component Model (CCM) alleviate some of these issues by providing reusable standardized services and reliable runtime environments. Such platforms allow developers to concentrate their efforts on fulfilling application requirements rather than spending time developing the underlying infrastructure. Nevertheless,

enterprise systems still prove extremely complex and difficult to design, as these systems tend to be very large. As a result it can be difficult for developers to completely understand the performance impact of their design decisions. The use of Commercial Off The Shelf (COTS) components may exasperate the problem of completely understanding system behavior, since the internals of such components may be unknown. Also, while performance decisions should ideally be taken early in the development cycle, system designers squeezed by shrinking development cycles, are often compelled to treat performance as an afterthought, a matter of production tuning to be accomplished after the system has been coded, integrated, functionally tested and deployed. As a result of using COTS components, time-to-market constraints and the complexity of enterprise systems developers may not have a proper understanding of the system design from a performance perspective and consequently complex distributed enterprise systems often suffer from severe performance problems. The next section gives a brief introduction to the field of data mining and knowledge discovery in databases, followed by the detailing of our monitoring, detection, assessment and visualization modules. Section III discusses some preliminary results. Section IV looks at related work and Section V presents the current status and future work. Section VI gives our conclusions.

## II. PROPOSED SOLUTION

We propose a framework for automatically detecting and assessing the impact of poor performance design (performance antipatterns [1]) in component based systems using a combination of run time and static analysis, by applying techniques borrowed from the field of Knowledge Discovery in Databases (KDD). While current performance tools tend to generate massive amounts of data that can prove overwhelming when trying to locate the cause of performance bottlenecks, our approach makes sense of such

data, and through processes similar to those used in the field of KDD presents the discovered “knowledge” in readable, familiar formats, pin-pointing performance problems within the system and their underlying cause. Our solution is applied to a running system in order to detect, assess and visualize performance antipatterns that may be present in the system, and can be applied both during the development cycle and to systems in production. Therefore modern software development practices, that require a running implementation of the system at each iteration of the development cycle (e.g. XP), fit well with our approach. The framework consists of four modules: a monitoring module, a detection module, an assessment module and a visualization module (see Fig. 1).

### A. Knowledge Discovery in Databases

KDD [2] combines techniques from machine learning, pattern recognition, statistics, databases, and visualization to automatically extract concepts, concept interrelations, and patterns of interest from large databases. The basic task is to extract knowledge (or information) from lower level data (databases). The basic tools used to extract patterns from data are called Data Mining methods. The process surrounding the usage of these tools (including pre-processing, selection, and transformation of the data) and the interpretation of patterns into “knowledge” is the KDD process. KDD systems alleviate the problem of manually analyzing the large amounts of collected data which decision-makers face currently. Data mining is currently used for such tasks as exploratory data analysis, descriptive modeling, predictive modeling, pattern discovery and retrieval by content. We are currently interested in using data mining techniques from the field of pattern discovery. Agrawal et al. [3] introduced a class of regularities, association rules, and gave algorithms for finding such rules. Association Rule Mining (ARM) is the process of finding relationships among attributes in large databases.

The exemplar application of ARM is market basket analysis. However, ARM can be applied to any activity where a large amount of data is available and we suspect that there might be something interesting in that data. We intend to apply association rule mining to the task of detecting poor performance design.

### B. Monitoring Module

The goal of the monitoring module is to collect run time data for each component in the distributed system. The monitoring module collects the method calls (and their related performance data) made by each component. Method calls from different components are aggregated into their related business transactions. The aggregation of the method calls in this manner gives a transaction centric view of the system. Performance tools such as [4] and [5] can perform this task.

### C. Detection module

It is the responsibility of the detection module to perform the data mining on the runtime data and subsequently discover the performance antipatterns that exist within the running system. This module uses the run time information collected by the monitoring module as its input. We divide the detection module’s activities into two separate phases: the Association Rule Mining (ARM) phase, and the Antipattern Detection (AD) phase. ARM is performed in order to find interesting relationships and patterns within the runtime data collected. The relationships that are found during the ARM phase are expressions of the form  $X \Rightarrow Y$ , where  $X$  and  $Y$  are sets of items, or in this case, sets of method calls. The intuitive meaning of such an expression is that transactions of the database which contain  $X$  tend to contain  $Y$ .  $X$  is sometimes referred to as the *antecedent* of the rule and  $Y$  the *consequent*. The following is a formal statement [3] describing the ARM problem performed in the detection module: Let  $I = \{i_1, i_2, \dots, i_m\}$  be a set of literals, called items. Let  $D$  be a set of transactions, where each transaction  $T$  is an itemset such that  $T \subseteq I$ . Each transaction has an associated unique identifier, called its TID. A set of items  $X \subseteq I$  is called an itemset.  $T$  contains an itemset  $X$ , if  $X \subseteq T$ . An association rule is an implication of the form  $X \Rightarrow Y$ , where  $X \subset I$ ,  $Y \subset I$ , and  $X \cap Y = \emptyset$ . The rule  $X \Rightarrow Y$  holds in the transaction set  $D$  with confidence  $c$  if  $c\%$  of transactions in  $D$  that contain  $X$  also contain  $Y$ . The rule  $X \Rightarrow Y$  has support  $s$  in the transaction set  $D$  if  $s\%$  of transactions in  $D$  contain  $X \cup Y$ . The ARM phase of the detection module discovers all association rules of sufficient support and confidence. A number of ARM algorithms already exist in the literature [6]. Once the association rules are discovered, they are used as input into the AD phase. Since a minimum support level is set when performing rule discovery, rules that do not occur frequently are disregarded and discovery of insignificant occurring antipatterns can be avoided. On completion of the ARM phase, the AD phase of the detection module inspects

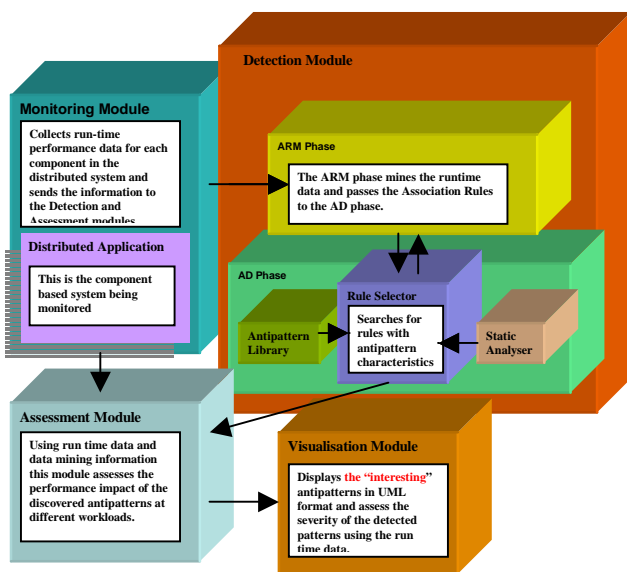


Fig. 1. Framework Overview

the discovered rules and uncovers the performance antipatterns within them. The AD phase is made up of three components: the Antipattern Library, the Static Analyzer and the Rule Selector. The Antipattern Library contains descriptions of the performance antipatterns that are to be detected by the system. The static analyzer obtains system specific information from the deployment descriptors of the system being analyzed. The rule selector inspects all rules to identify antipattern instances that may exist within them. System specific information is required by the Rule Selector so that it can identify the antipattern characteristics within the rules being analyzed. The required information can be obtained from the EJB deployment descriptors, which avoids the need to inspect the system source code. Deployment descriptor inspection is performed by the static analyzer and is advantageous since with the increasing use of COTS components, system source code may not be available in its entirety. For every performance antipattern in the Antipattern Library the Rule Selector inspects all rules discovered by the ARM phase. Rules which contain antipattern characteristics are marked as antipattern instances. In order to detect instances of certain antipatterns, rules at a range of support and confidence levels may be required. Therefore, for certain antipatterns the rule selector may request a further pass of the ARM phase.

#### D. Assessment module

Antipattern instances discovered by the detection module are assessed in light of the performance impact they have on the system. Performance data collected during runtime analysis, as well as the support and confidence levels (data mining information) associated with the discovered patterns, are utilized to compute the severity of each antipattern instance for the current system load and at different workloads. By analyzing support and confidence levels, the trends in antipattern occurrences can be revealed. Applying performance data to the discovered trends gives the assessment module the ability to detect and highlight how antipatterns affect the system at different loads. As a result, even at low loads the framework has the ability to detect antipatterns that may only appear at higher workloads. This is quite advantageous since load dependent antipatterns can often be overlooked during testing. In many cases the testing environment may not be able to simulate loads quite as large as those that would occur in the production environment. The assessment module using this information can calculate and suggest the optimal refactoring that should be made to maximize system performance pinpointing exactly where in the system the antipatterns are occurring. Antipatterns document recurring problems and mistakes made during software development. As well as documenting these problems, antipatterns also document their solutions. Performance antipatterns [1], in documenting the problem, list the performance principles [1] violated by the antipattern. Thus, antipatterns help to identify what is wrong with a software project and what can

be done to resolve these issues. Pin-pointing the existence of antipatterns with suggestive refactorings and explanations of the problem not only helps developers to locate and treat performance issues within the system, but also helps form a sense of performance intuition as the underlying performance principle [1] being violated by the antipattern is disclosed.

#### E. Visualization module

The visualization module of the framework presents the discovered antipatterns in a graphical format using familiar formats, such as UML, that developers can easily comprehend. The required information for visualization is obtained from the assessment module.

### III. RESULTS

In the following sections we provide a sample application containing two EJB performance antipatterns and describe how our detection module detects each pattern. Work is currently underway to automate the process described.

#### A. Antipattern A: Fine Grained Remote Calls

A major cause of poor performance in component based distributed systems is a large communication overhead between the distributed components of the system. In many cases this large communication overhead is a direct result of bad system design. Often if performance issues had been taken into consideration at the design stage, distributed communication between components could have been dramatically reduced. Related antipatterns creep up in EJB systems in many shapes and forms. The problem is that multiple fine grained invocations of remote session/entity beans add the overhead of multiple network calls. A typical example is the accessing of remote entity beans directly, rather than accessing them through a session facade.

#### B. Antipattern B: Aggressive Loading of Entities

Entity beans are persistent data components, i.e. they are an in memory Java representation of persistent data. Entity bean data is loaded into an in-memory entity bean instance by reading data stored in a database in order to manipulate the data within the Java Virtual Machine. The procedure of loading the data from the database can be a time consuming process especially when entity beans are large objects with numerous fields. The situation is not helped by the fact that beans may be involved in inter-bean relationships. Consequently, the loading of an instance of a single entity bean may result in the loading of numerous entity beans from the database, producing a large entity bean graph. This is especially wasteful if the case is such that only a few fields are actually needed. A common solution to this problem is to lazy load the fields that are not generally

needed. Loading a complete entity bean graph is known as aggressive loading and is a common EJB antipattern.

### C. Sample Application and Detection Process

In order to provide a proof of concept of our detection technique we have implemented a sample application with performance related design flaws. The application can be used to calculate employee's income tax and a company department's corporation tax. It consists of two EJB entity beans namely *Department* and *Employee* and an EJB session bean *Admin*. A sample transaction file produced by the monitoring module is shown in Fig. 2. and contains the five different business transactions that can be performed using the application. We have omitted the performance data from Fig. 2. as for this example we are concerned with detection of the antipatterns only, and do not assess their related performance impact. We apply our technique to the following list of transactions that were observed by the monitoring module over a particular time period: {*Transaction1*, *Transaction2*, *Transaction3*, *Transaction2*, *Transaction3*, *Transaction2*, *Transaction3*, *Transaction4*, *Transaction5*, *Transaction4*, *Transaction5*, *Transaction4*}. The transactions are a result of the system user looking up the *Department* entity bean in order to get the department *name*, *Id*, *wageBill* and a collection of *Employee* beans (transaction 1). For each *Employee*, employee details are obtained (transaction 2) and a session bean is used to calculate an employee's income tax (transaction 3). The user next looks up a number of *Department* beans and retrieves values for the department *name* and *wageBill* (transaction 4) followed by a call to the *Admin* session bean to calculate the department corporation tax (transaction5). The ARM phase of the detection module inspects the information provided by the monitoring module and discovers all frequently occurring rules. An initial minimum support value of 25% and a minimum confidence level of

```

<interaction name="Business Transaction 1">
  <call ejb="DepartmentHome" method="findByPrimaryKey" ></call>
  <call ejb="Department" method="getName" ></call>
  <call ejb="Department" method="getId" ></call>
  <call ejb="Department" method="getWageBill" ></call>
  <call ejb="Department" method="getEmployees" ></call>
</interaction>
<interaction name="Business Transaction 2">
  <call ejb="EmployeeHome" method="findByPrimaryKey" ></call>
  <call ejb="Employee" method="getFirstName" ></call>
  <call ejb="Employee" method="getLastName" ></call>
  <call ejb="Employee" method="getAddress" ></call>
  <call ejb="Employee" method="getTotalSales" ></call>
  <call ejb="Employee" method="getWeeklySales" ></call>
</interaction>
<interaction name="Business Transaction 3">
  <call ejb="AdminHome" method="create" ></call>
  <call ejb="Admin" method="calculateIncomeTax" ></call>
</interaction>
<interaction name="Business Transaction 4">
  <call ejb="DepartmentHome" method="findByPrimaryKey" ></call>
  <call ejb="Department" method="getName" ></call>
  <call ejb="Department" method="getWageBill" ></call>
</interaction>
<interaction name="Business Transaction 5">
  <call ejb="AdminHome" method="create" ></call>
  <call ejb="Admin" method="calculateCorporationTax" ></call>
</interaction>

```

Fig. 2. Sample Output from the Monitoring Module

30% is selected. The AD phase of the detection module inspects the rules discovered, searching for certain characteristics that may indicate the presence of antipatterns. The aim is to detect the presence of the two antipatterns described above. The detection of Antipattern A involves searching the discovered rules for rules of the form:  $X \Rightarrow Y$ , where  $X$  is an itemset that contains one remote call and  $Y$  is an itemset that contains one or more remote calls. Such antipattern descriptions are stored in the antipattern library. The detection module requires system specific information in order to be able to differentiate between local and remote calls. The system specific information can be acquired from the EJB system's XML deployment descriptors. Deployment descriptors provide information which allows one to determine whether beans are local or remote. The detection module uses the static analyzer to inspect the deployment descriptors and lists all remote beans, and their respective remote interfaces. The detection module, loaded with this information, inspects all rules and marks rules that satisfy the marking criteria for the particular antipattern being detected. The marking criteria for Antipattern A are:

1. If both the antecedent and consequent contain remote bean invocations (through the remote interface) the rule will be marked as a possible antipattern.
2. If two rules  $X$  and  $Y$  exist that contain remote calls in their antecedent and consequent, where rule  $X$  shares the same antecedent as rule  $Y$ , the consequent of rule  $X$  is a subset of the consequent of rule  $Y$ , and rule  $X$  has the same levels of confidence and support as rule  $Y$ , only rule  $Y$  is marked as a possible antipattern.

The second criterion above prevents the marking of multiple rules that point to the same antipattern or business transaction(s). Applying the marking criteria to our sample application our detection module marks the rules depicted in Table 1. Rules 1 and 2 relate to Business Transaction 4 and Business Transaction 1 (Fig. 2) and state that "100% of transactions that contain department.getName() also contain department.getWageBill()" (and vice versa). This rule is present in 33% of all transactions. Since these two remote calls are always made together the number of remote calls could be reduced using the Value Object (VO) pattern [10] to encapsulate the business data and transport it from the application tier to the client tier reducing the number of remote calls to one. Rules 3-7 relate to Business Transaction 2 (Fig. 2). Rule 3 declares that "100% of transactions that contain employee.getFirstName() also contain employee.getLastName(), employee.getAddress(), employee.getWeeklySales() and employee.getTotalSales". Rule 3 (rules 4-7 point to the same antipattern) is present in 25% of all transactions. Again Antipattern A has been detected. In this situation applying the VO pattern would reduce the number of remote calls from five calls to one. In cases where the consequent is large, a significant reduction in network traffic can be obtained through refactoring this antipattern. The detection of Antipattern B involves, detecting how often only a proportion of an entity bean's fields are accessed, detecting how often all of the fields of

Table1.

| Rule # | Antecedent              | Consequent   | S (%) | C (%) |
|--------|-------------------------|--|-------|-------|
| 1      | department.getName      | department.getWageBill   | 33    | 100   |
| 2      | department.getWageBill  | department.getName   | 33    | 100   |
| 3      | employee.getFirstName   | employee.getLastName, employee.getAddress, employee.getTotalSales, employee.getWeeklySales   |       |       |
| 4      | employee.getLastName    | employee.getFirstName, employee.getAddress, employee.getTotalSales, employee.getWeeklySales  | 25    | 100   |
| 5      | Employee.getAddress     | employee.getFirstName, employee.getLastName, employee.getTotalSales, employee.getWeeklySales | 25    | 100   |
| 6      | Employee.getTotalSales  | employee.getFirstName, employee.getLastName, employee.getAddress employee.getWeeklySales     | 25    | 100   |
| 7      | Employee.getWeeklySales | employee.getFirstName, employee.getLastName, employee.getAddress employee.getTotalSales      | 25    | 100   |

the same entity are accessed, and comparing these values to give an idea of what fields should be ideally lazy loaded. This requires searching the discovered rules for rules of the form:  $X \Rightarrow Y$  where  $X$  an itemset that contains one getter method relating to a particular entity bean persistent field, and  $Y$  is and itemset that contains zero or more getter methods that belong to the same entity bean. Again system specific information can be acquired from the EJB systems XML deployment descriptors. The static analyzer inspects every entity bean’s deployment descriptor and lists all persistent fields belonging to each entity bean. All rules are then inspected with the following marking criteria:

1. If the antecedent of a rule contains a persistent field getter method belonging to the particular entity, the rule is marked.
2. If two rules  $X$  and  $Y$  exist, where rule  $X$  shares the same antecedent as rule  $Y$ , the consequent of rule  $X$  is a subset of the consequent of rule  $Y$ , and rule  $X$  has the same levels of confidence and support as rule  $Y$ , only rule  $Y$  is marked as a possible antipattern.

Again the second criterion prevents the marking of multiple rules that point to the same antipattern or business transaction(s). From the marked rules we select any rules where the antecedent and consequent do not contain all the getter methods that belong to the particular entity bean. We

refer to these rules as the small rules. If a small rule is found the detection module then searches for rules that contain a higher number of persistent field getter methods belonging to the same entity bean. We refer to these rules as the large rules. A further pass of the ARM phase can be performed (and may be required) in order to find the large rules. If not found after further passes we can stop searching for the large rules, satisfied that the large rules do not occur above a certain level of support and confidence. The small rules are compared to the large rules in order to show the presence of Antipattern B. We applied this technique to our sample application. The rules depicted in Table 1 were marked when support and confidence were set to 25% and 30% respectively. Rules 1 and 2 are small rules and correspond to Business Transaction 4 and Business Transaction 1 and declare that “100% of transactions that contain department.getName() also contain department.getWageBill()” (and vice versa). This rule is present in 33% of all transactions. Only half of the persistent fields of the Department entity bean are covered by rules 1 and 2. The detection module now searches for the corresponding large rules. Since there are no corresponding large rules in Table 1, a second pass of the ARM phase of the detection module with lower support and confidence settings is required. For the second pass we set the confidence and support to 25% and 8% respectively.

Table2.

| Rule # | Antecedent               | Consequent   | S (%) | C (%) |
|--------|--------------------------|--|-------|-------|
| 1      | department.getName       | Department.getWageBill   | 33    | 100   |
| 2      | department.getWageBill   | Department.getName   | 33    | 100   |
| 3      | Employee.getFirstName    | employee.getLastName, employee.getAddress, employee.getTotalSales, employee.getWeeklySales   | 25    | 100   |
| 4      | Employee.getLastName     | employee.getFirstName, employee.getAddress, employee.getTotalSales, employee.getWeeklySales  | 25    | 100   |
| 5      | employee.getAddress      | employee.getFirstName, employee.getLastName, employee.getTotalSales, employee.getWeeklySales | 25    | 100   |
| 6      | employee.getTotalSales   | employee.getFirstName, employee.getLastName, employee.getAddress employee.getWeeklySales     | 25    | 100   |
| 7      | employee.getWeeklySales. | employee.getFirstName, employee.getLastName, employee.getAddress employee.getTotalSales      | 25    | 100   |
| 8      | department.getEmployees  | department.getName, department.getId, department.getWageBill                                 | 8     | 100   |
| 9      | department.getId         | department.getName, department.getWageBill department.getEmployees                           | 8     | 100   |
| 10     | department.getName       | department.getId, department.getWageBill, department.getEmployees                            | 8     | 25    |
| 11     | department.getWageBill   | department.getName, department.getId, department.getEmployees                                | 8     | 25    |

Applying the marking criteria again to the rules produced by the second pass of the ARM phase, the rules depicted in Table 2 are produced. Rules 10 and 11 (large rules) from Table 2 correspond to Rules 1 and 2 respectively, (small rules) from Table 1. A comparison of the rules highlights the presence of Antipattern B, since (from Rules 1 and 2) 100% of transactions that contain `department.getName()` also contain `department.getWageBill()` (and vice versa), however (from Rules 10 and 11, Table 2) only 25% of the transactions that contain either `department.getName()` or `department.getWageBill()` contain getter methods for all persistent fields. If no large rules were found with support of 8% and confidence of 25% we could have been satisfied that while 100% of transactions that contained `department.getName()` also contained `department.getWageBill()`, less than 25% of transactions that contained either of these two getter methods also contained all other persistent field getter methods. In this example scenario a performance improvement can be gained in 25% (33-8) of all transactions by lazy loading the *Id* and *Employees* fields, however 8% of the time a performance hit will be taken when the *Employees* and *Id* fields are requested and the information has to be obtained from the database. We intend applying this technique to detect a wide range of EJB performance antipatterns [9].

#### IV. RELATED WORK

Previous approaches to pattern detection have generally been focused on design recovery [7][8]. Earlier approaches [7] were restricted to static analysis techniques. With the dynamic nature of current component technologies static pattern detection is not sufficient. More recent approaches [8] have combined the advantages of static and dynamic analysis to overcome this problem and improve design pattern instance recognition. In general source code is used as the basis for the static analysis. However, with the increasing trend towards COTS components, the entire source code of a modern component based systems may not always be freely available. Our approach also uses a combination of static and dynamic data. However we overcome the issue of source code availability by extracting any static system information we require from the system deployment descriptors which are deployed with the system. Also, in contrast to [8] where the aim is to find all instances of a given pattern, our approach aims at finding the most frequently occurring instances of a given pattern, since such instances have a more substantial impact on system performance. The related approaches attempt to uncover design patterns in the system, as performance engineers, we are more concerned with the detection of performance antipatterns [1] within the system. While design patterns (and antipatterns) are generally concerned with software quality attributes such as reusability and maintainability, performance (anti)patterns, are solely focused on performance concerns. We are particularly interested in EJB performance antipatterns [9].

#### V. CURRENT STATUS & FUTURE WORK

A proof of concept of our detection module has been implemented. An implementation of the assessment and visualization modules is currently underway. Research into the area of data mining techniques is in progress with the aim of maximizing the efficiency of our detection module. Emphasis is also currently being placed on looking into the field of data mining in greater detail to see how it could be further applied to extract valuable information from runtime data. Finally we plan on applying our framework to a typically large e-commerce system.

#### VI. CONCLUSION

We propose a framework to automatically detect and assess the impact of poor performance design in component based systems. Our solution is applied to a running system in order to detect performance antipatterns that are present in the system. The framework consists of four modules a monitoring module, a detection module, an assessment module and a visualization module. We have presented an approach to detect the presence of performance antipatterns using techniques borrowed from the field of data mining and KDD. To the best of our knowledge we are the first to apply such techniques to the task of performance antipattern detection.

#### ACKNOWLEDGEMENTS

Our work is funded under the Research Innovation Fund and the Advanced Technology Research Programme from the Informatics Research Initiative of Enterprise Ireland.

#### REFERENCES

- [1] Smith, C. U. and Williams, L. G., "*Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*", Addison-Wesley, Boston, MA, 2002.
- [2] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. "The KDD process for extracting useful knowledge from volumes of data." *Communications of the ACM*, volume 39, issue 11, 1996.
- [3] Rakesh Agrawal, Tomasz Imielinski and Arun Swami. "Mining Association Rules between Sets of Items in Large Databases". *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.
- [4] <http://www.quest.com/performance>
- [5] <http://ejbperformance.org>
- [6] Jochen Hipp, Ulrich Güntzer, Gholamreza Nakhaeizadeh "Algorithms for Association Rule Mining A General Survey and Comparison". *SIGKDD Explorations*, 2000.
- [7] C. Kramer and L. Prechelt. "Design recovery by automated search for structural design patterns in object-oriented software". *Proceedings of the 3<sup>rd</sup> Working Conference on Reverse Engineering (WCRE)*, Monterey, CA, pages 208–215. November 1996.
- [8] Lothar Wendehals. "Improving Design Pattern Instance Recognition by Dynamic Analysis". *WODA, ICSE 2003*
- [9] Tate, B., Clark, M., Lee, B. and Linskey, P., "*Bitter EJB*", Greenwich, CT, Manning, 2003.
- [10] Deepak Alur, John Crupi and Dan Malks. "*Core J2EE Patterns: Best Practices and Design Strategies*." Prentice Hall / Sun Microsystems Press, 1st edition (June 26, 2001)