Identification of Design Roles for the Assessment of Design Quality in Enterprise Applications

Cristina Marinescu
LOOSE Research Group
"Politehnica" University of Timişoara, Romania
cristina@cs.utt.ro

Abstract

*The software industry is increasingly confronted with the issues of understanding and maintaining a special type of object-oriented systems, namely enterprise applications (EA). In the recent years many specific rules and patterns for the design of such applications were proposed. These new specific principles of EA design define precise roles (patterns) for classes and methods, and then describe "good-design" rules in terms of such roles. Yet, these roles are rarely explicitly documented; therefore, due to their importance for an efficient understanding and assessment of EA design, they must be identified and localized in the source code based on their specificities. In this paper we define a suite of techniques for the identification and location of four such roles, all related to the data source layer of an EA. Using the knowledge about these roles we show how this can improve the accuracy of formerly defined techniques for detecting two well-known design problems (*i.e., Data Class *and* Feature Envy*), making them more applicable for the usage on enterprise systems. Based on an experimental study conducted on three EAs, we prove the feasibility of the approach, discuss its benefits and touch the issues that need to be addressed in the future.*

1. Introduction

In the recent years, as object-oriented systems became increasingly complex, a novel category of software systems emerged, namely *enterprise applications*. These systems are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data [11].

It is well known both in the theory and practice of software engineering that one of the major issues with large-scale, highly complex applications is that a poor design (*i.e.*, a design affected by design flaws) has a strong negative impact on quality attributes such as flexibility or maintainability [10]. This applies very much to enterprise applications.

From the point of view of design and implementation these systems can be regarded as object-oriented ones, usually consisting of three primary layers (see Figure 1): presentation, domain and data source [9]. Therefore, apparently, the assessment of design quality could be based on the principles, heuristics and best practice of object-oriented design (*e.g.*, [23, 27, 12]). But is this enough?

It is clear from the state-of-the-art literature [11, 14, 18, 25, 22] that, within an enterprise application, the presentation layer, the data source layer, and also their interaction with the domain layer, are governed by a novel set of principles and patterns which state more specifically what is "good design" for an enterprise system. Furthermore, comparing these specific design rules with the ones used for the design quality assessment of "regular" object-oriented systems we notice that sometimes they are even conflicting. For example, a *Data Transfer Object* (DTO) [11, 18] which in an enterprise application is a class carrying data between a client and a server will be always detected as a *Data Class* [10] design flaw. Thus, using a strict object-oriented perspective we are in danger of getting **incomplete** and **inaccurate** results. Consequently, as the number and the complexity of enterprise systems is increasing, the need to maintain and evolve these systems will require more and more specific means both to understand and to assess *completely* and *accurately* the quality of their design. In order to achieve this goal, the heuristics and patterns of enterprise applications design must be be taken into account. In the last ten years various valuable techniques have been defined for quality assessment in object-oriented systems [3, 8, 15, 19, 20, 21]. Yet, regarding them from the perspective of enterprise applications, almost all of these approaches are limited in two aspects:

- they rely exclusively on principles, heuristics and best practices of object-oriented design.
- they are based exclusively on a structural view of the

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

source code, without integrating any additional information (*e.g.*, the database schema).

In this paper we aim to lay the foundation for a new approach of understanding and assessing the design of enterprise applications. In order to do this, we first need to identify the classes and methods that fulfill *design roles* which are specific for enterprise systems, and which are described in literature in form of various patterns (*e.g.*, a class that acts as a *Data Transfer Object* [11]). In this context, we define a suite of automatic detection techniques for several *design roles*. The ability to identify such roles is a first step towards a specific understanding of an enterprise application's design. In the second part of the paper we show how the detection accuracy of two well-known design flaws (*i.e.*, Data Class, Feature Envy) can be improved by taking into account these identified design roles.

The paper is structured as follows: in Section 2 we present the main characteristics of an enterprise application and we introduce the notion of a *design role* attached to a design entity (*e.g.*, class, method). In Section 3 we introduce a suite of techniques for automatically identifying several design roles related to the *data source layer*. Next (Section 4), taking into account the design roles introduced in the previous section, we revise the detection method for two design flaws in order to make it more accurate for enterprise applications. This is followed by a brief description of the tool support (Section 5) that ensures the automation of the entire approach and by an experiment (Section 6) based on three case-studies. The conducted experiment is intended to reveal the applicability and the accuracy of the approach. The paper concludes with a discussion on related work (Section 7) and some final remarks towards the future work (Section 8).

2. Characteristics of Enterprise Applications

Usually, an enterprise application involves a lot of persistent data, its users manipulate the data concurrently and has a lot of user interface screens [11]. Within such an application it is desirable to be able to change the user interface or the persistency provider without affecting the rest of the application *i.e.*, the application's logic. Moreover, mixing the commands that ensure the persistency (usually the persistency is supplied by a database, mostly relational database) and application logic hampers the understanding and the testing of the application. This led to a multi-layered architecture, consisting of three primary layers namely data source, domain and presentation [9]. The responsibility of each layer is presented in Figure 1. As we can notice in Figure 1, in a well designed enterprise application there should be no dependency going from a lower-level layer to a higher-level one (*e.g.*, the data source and the domain layers should not depend on the presentation).

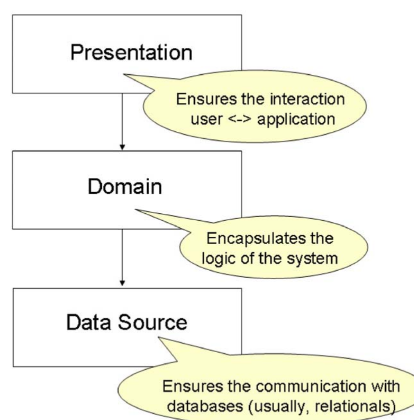

Figure 1. Responsibilities of each layer and their interdependencies in an well-designed enterprise application.

2.1. Design Entities, Roles and Layers

In an enterprise system each design entity (*e.g.*, class, method) belongs to a layer. Furthermore, Fowler in [11] identifies and describes in the form of patterns different types of classes and methods that have (or should have) responsibilities with respect to the structure of an enterprise application or, more specific, with respect to the design of a particular layer. In this context we define the notion of **design role** as follows:

Definition 2.1 (Design Role) A design role *is a specific responsibility that a design entity (*i.e.*, a class or a method) might have with respect to the design of a specific layer or to the collaboration between two layers within an enterprise application.* A design role *is reflected in the design entity by a suite of constraints related to its structure and/or the functionality that it provides.*

If a class or method has a particular *design role* then knowing it provides the engineer who maintains or evolves the system with additional semantical information about its place within the layer where it belongs. Not every entity will have an "enterprise-specific" role and if we know the role of an entity, we also know the layer it belongs to.

2.2. Design Roles and Quality Assessment

The knowledge about layers and design roles is very important both for understanding and for assessing the design of an enterprise application, at least for the following two reasons:

- Patterns of desirable and avoidable interactions in an EA are defined in terms of an entity's *affiliation* to a

IEEE COMPUTER SOCIETY

specific layer and/or to a specific role. For example, a class should not contain methods that belong to the presentation layer and other methods belonging to the data source layer. Or, a class that is a *Table Data Gateway* [11, 1] (that is its role) should not contain methods that belong to the domain layer.

- They help the reverse engineer of the system by providing him with *orientation points* about the interactions in the system in terms of layers and roles.

3. Extracting Roles in the Data Source Layer

As we presented the importance of layers and roles in the context of quality assessment of enterprise applications, an essential question arises: While these design roles are described in an informal manner, how can they be identified automatically in a given enterprise application? In this section we present a suite of *role-mining techniques* for several design roles that characterize the data source layer.

3.1. Roles in the Data Source Layer

Before describing the detection techniques for the identification of roles we first present the main specific roles that methods and classes may have within the *data source* layer of an enterprise application. Why do we focus on the *data source* layer? Apart from the space constraints of this paper, the reason resides in the fact that the aforementioned layer is usually responsible for assuring the proper bridging of two different paradigms *i.e.*, the relational database and the object-oriented model; and it is well known that this issue raises many understanding and design quality concerns [14].

Design Roles of Methods. Design roles are revealed especially by particular patterns of provided functionality and therefore methods are the lower level design entities that can have design roles. Thus, within the data source layer, a method might have one of the following four roles: to *retrieve* (R), *insert* (I), *modify* (M) or *delete* (D) information from tables of the relational database that ensures the persistency within the enterprise application. When a method perform one of the aforementioned operations on a table T, we say that the *method accesses table* T.

Design Roles of Classes. According the way a class is structured, different roles associated to a class within the data source layer have been arisen. A class that belongs to the data source layer might be a

- *Table Data Gateway (TDG)* [11][1] which encapsulates accesses to one or more database tables. The accessed tables from a class are the set of distinct tables accessed from its methods. One instance handles all the rows in the tables and holds all the SQL for accessing the tables.
- *Row Data Gateway (RDG)* [11] which encapsulates access to a single record in a data source table. There is one instance per row. RDG allows changing the structure of the database with fewer changes in the application than TDG.
- *Active Record (AR)* [11][2] which wraps a row in a database table, encapsulates the database access, and adds domain logic on that data. AR is suitable for a domain logic that is not too complex. The major drawback of this pattern is that it couples the domain logic with the database access.

3.2. Identifying Roles in the Data Source Layer

Starting from the previous definitions of the three roles of classes we identified a set of specific features based on which they can be automatically identified in the source code. A first mandatory condition for our identification rules is that each table be accessed by a single class. Thus, if a class *C* accesses a table that is also accessed by another class, then no role can be assigned to *C*. Apart from this precondition, the rest of the roles'identification process for a class is captured by the flowchart depicted in Figure 2.

Thus, if class *C* accesses one or more tables, is stateless (*i.e.*, it contains only static and/or final attributes) and all its public methods ensure the communication with the database (belong to the data source layer) then class *C* is a *Table Data Gateway*.

In a similar fashion we say that a class *C* is a *Row Data Gateway* if it accesses one single table, all its public methods belong to the data source layer, and for each column in the accessed table class *C* defines an attribute.

Eventually, if class *C* accesses one table, for each column in the accessed table defines an attribute and, beside public methods from the data source layer, it contains also public methods belonging to the domain layer, then class *C* is considered to be an *Active Record*.

3.3. Extracting Design Information

The description of the design roles presented in the previous section suggests, that in order to identify them in a repeatable and automatic manner in the code, we need to build a model that contains various types of design information. More precisely, based on the extraction's algorithm

1 a.k.a. Data Access Object [1], Domain Object Assembler [25]

2 a.k.a. Active Domain Object [25]

0-7695-2601-2/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

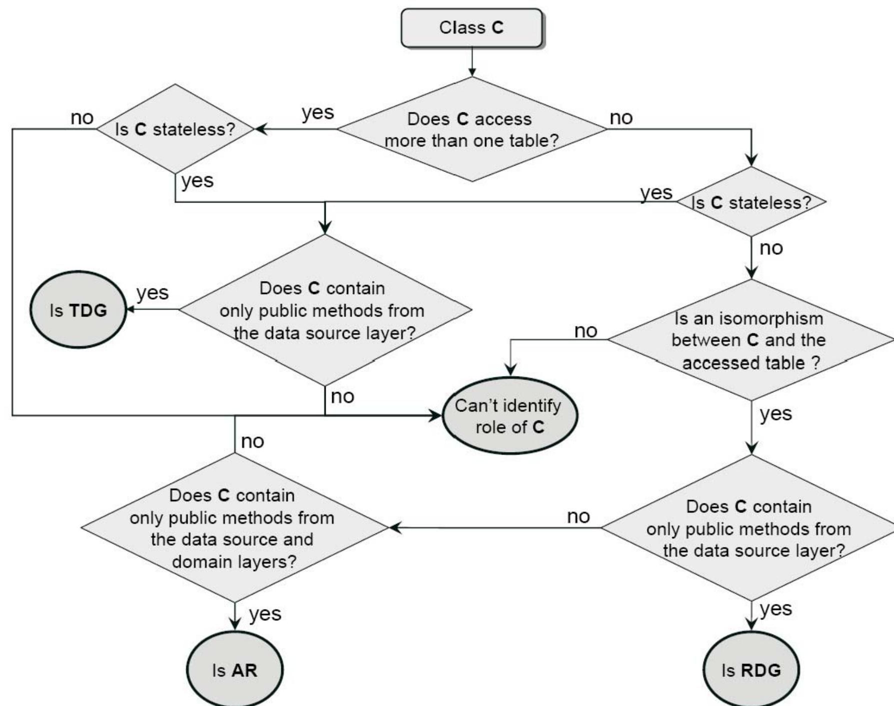

Figure 2. Identification of roles in the data source layer.

presented in Figure 2, we conclude that for the identification of the three aforementioned roles from the data source layer the following information is needed: the *attributes* and *methods* of a given class, the *columns* of a given database table, the *specific functionality* of a given method (*i.e.*, communication with the database, domain logic) and the *tables* from the database accessed by a given class.

This reveals an important aspect about the representation requirements for defining techniques for design understanding and/or quality assessment of enterprise applications: compared to previously defined techniques – employed for assessing common object-oriented design – these new analysis techniques for enterprise systems require an *enlarged set of design information* which is extractable not only from the *source code* of the enterprise system, but also from its *database schema*, as depicted in Figure 3.

Thus, besides the information about object-oriented entities (*e.g.*, classes, methods, variables) and their interactions (*e.g.*, accesses of variables, method calls, inheritance relations), for an enterprise application we also need to extract design information specific to its relational part (*e.g.*, tables, columns, primary and foreign keys). Figure 3 reveals also a simple yet highly important aspect: there is an interaction between the entities from the *object-oriented model* (especially methods) and the entities from the *relational model* (*e.g.*, the tables accessed by all the methods from a class).

Based on the generic remarks made in this section, we introduce in Section 5 our concrete implementation for building such an extended model. But for now our aim was just to emphasize the intrinsic and general need of an extended representation of an enterprise application's design, compared to the one used for common object-oriented systems.

3.4. Mapping of Design Entities to Layers

As mentioned in Section 2, the role of a design entity is defined with respect to the layer where it belongs. Thus, for each design entity we need to find the layer (or sometimes layers) where it belongs to. In order to do this, we take a simple approach to this issue, by taking into account the various usages of third-party libraries and/or frameworks that are specific either for the presentation or for the data source layer.

Mapping of Methods to Layers. The following rules determine the mapping of methods to one of the three layers of an enterprise application:

1. A method is considered to belong to the **data source layer** if it invokes one or more methods from a specific library that provides an API for accessing and processing data stored in a data source, usually a relational database (*e.g.*, `executeQuery()` method from the `java.sql` package).
2. A method is mapped to the **presentation layer** if it calls one or more methods from a specific third-party

0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

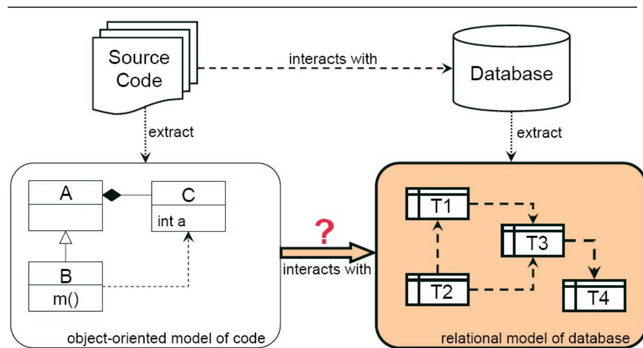

Figure 3. Design information must be extracted both from the code and the database schema.

library and/or framework that provides components for building the user interface (*e.g.*, calls methods from classes found in the `java.swing` package).

3. If none of the previous two rules applies for a method then the method is mapped to the **domain layer**.

Mapping of Classes to Layers. In order to map classes to layers we use the following set of rules:

1. A class containing one or more methods belonging to the **data source layer** will be mapped to the **data source layer**.
2. A class that contains one or more methods belonging to the **presentation layer** will be mapped to the **presentation layer**.
3. A class that is derived directly or indirectly from a specific third-party library and/or framework that provides components for building the user interface is mapped to the **presentation layer** (*e.g.*, a class that extends `JButton`).
4. A class that contains one or more methods belonging to the **domain layer** will be mapped to the **domain layer**.
5. A class that contains methods that belong to different layers will map the class to each of the layers to which its methods are mapped to. For example, if a class has methods belonging to the data source layer and to the domain layer, the class will be mapped both to the data source and to the domain layer.

Remark. In conformity to these rules, a method will always belong to a single layer, while the last rule for class mapping makes it possible for a class to be assigned to several layers. This mapping decision might look confusing, as we would expect for each design entity to belong to a single layer. But note that we extract (reverse engineer) these mappings from the source code, where the clear initial design intentions might have become "blurred" during implementation. Thus, we expect at least some of these classes that are mapped to multiple layers to reveal signs of design flaws; yet, this quality assessment of multiple mappings is beyond the scope of this paper.

4. Roles-Aware Detection of Design Flaws

As mentioned in the beginning of the paper, being aware of the various design roles of classes can help to improve the accuracy of detecting design problems. In this section we present a roles-aware enhancement for the detection of two well-known related design flaws.

4.1. Data Class versus Data Transfer Object

Data Classes [10] are dumb data holders that provide almost no functionality. The lack of functional methods may indicate that related data and behavior are not kept in one place; this makes it a strong signal for an improper data abstraction. But is a a *Data Class* always a design flaw? As we are going to see next, in an enterprise application, the answer is not obvious and it depends on layers and roles.

In order to explain, let us consider the following example. Considering an enterprise application that manages a library, let's assume that all books are stored in a table called *books* (see Figure 4), and we need to find out the information about a specific book, identified based on its ID.

```
create table books (
  ID int primary key, title varchar,
  author varchar, publisher varchar, year int)
```

Figure 4. Table books.

In order to retrieve the information, we create the *BookDataSource* class. As you notice in Figure 5 the class is designed to fulfill the *Table Data Gateway* role. For each column in the table a method that returns its corresponding value for a given *ID* was created (*e.g.*, method *getAuthor* returns the value stored in the column named *author*). Consequently, a class from the domain layer that needs all the information about a book would need to call 4 methods. It is clear that retrieving in this manner large amounts of data, by performing numerous fine-grained calls to the server will be simply a performance killer [18].

So, how should methods from a *Table Data Gateway* class retrieve information to its clients? A solution that is very often encountered in enterprise systems is the use of a

0-7695-2601-2/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

```
class BookDataSource {
 public String getAuthor(int id)
              throws Exception { ...
   String query;
   query = "SELECT author from books " +
           "WHERE ID=" + id;
   ResultSet rs = statement.executeQuery(query);
   return rs.getString("author"); }

 public String getTitle(int id)...
 public String getPublisher(int id)...
 public String getYear(int id)... }
```

Figure 5. Class BookDataSource.

Data Transfer Object (DTO) [3] [11, 18]. A DTO is the instance of a class with little more than a bunch of fields and afferent getters and setters for each of these fields. Such an object carries data between a client (the domain layer) and a server (the data source layer which encapsulates access to a database) in order to reduce the number of fine-grained method calls. In Figure 6 we see how the use of a DTO changes the *BookDataSource* example.

```
class Book {
 private String author, title, publisher;
 private int year;
 public void setAuthor(String author) {
  this.author = author; }
 public String getAuthor() {
  return author; }
 //other getters and setters methods
}
class BookDataSource {
 public Book getBook(int id) throws Exception {
   ...
   query = "SELECT * from books " +
           "WHERE ID=" + id;
   rs = statement.executeQuery(query);
   Book b = new Book();
   b.setAuthor(rs.getString("author"));
   b.setTitle(rs.getString("title"));
   b.setPublisher(rs.getString("publisher"));
   b.setYear(rs.getInt("year"));
   return b; }
}
```

Figure 6. Class BookDataSource Revised.

In an enterprise application a *Data Transfer Object* will always be detected as a design flaw (*i.e.*, as a *Data Class*). But reporting a class that fulfills the *Data Transfer Object* design role as a design flaw is a *false positive*; in other words an undesirable "detection noise". In order to improve the detection accuracy of *Data Classes* [20] for enterprise applications, classes that have a DTO design role must be removed from the list of suspects.

3 a.k.a. Value Object [1]

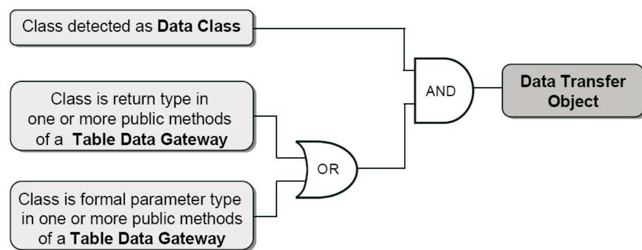

Figure 7. Detection of Data Transfer Object.

Based on the previous considerations, we define the following detection rule for a *Data Transfer Object* role: these are classes that fulfill the following conditions (see Figure 7):

1. the class is a *Data Class* in conformity with the detection rule described in [20].
2. the class appears as a **return type** in at least one public method of a class that has a *Table Data Gateway* design role or,
3. the class appears as **formal parameter type** in at least one public method of a *Table Data Gateway* class.

4.2. Enhanced Detection Rule for Feature Envy

Feature Envy [10] is another frequent design flaw that refers to methods that appear to be more interested in the attributes of another class than those of its own class. Oftentimes, this design flaw appears in methods which collaborate with *Data Classes*. In other words, *Feature Envy* usually is the sign of abnormality that appears on the clients side in case of an improper data encapsulation [20].

In [21] Marinescu proposes a metrics-based technique for automatically detecting methods affected by Feature Envy. The detection technique takes into account how many attributes are used by methods from other classes (directly or via getters and setters) compared to the usage of the attributes from its definition class. Based on the aforementioned detection rule, the `getBook` method listed in Figure 6 is apparently affected by *Feature Envy*, due to the fact that it accesses all the fields of class `Book` (via setter methods). But as discussed in this section, class `Book` is designed by intention as a data carrier and thus, the apparent *Feature Envy* we identified (method `getBook`) is harmless and should not be refactored.

Based on the design roles discussed so far in the paper we can now propose a more accurate detection rule for *Feature Envy*, in the context of enterprise applications. The new detection rule for Feature Envy will count usages of attributes from another class *only if the provider class has not*

IEEE COMPUTER SOCIETY

the design role of a Data Transfer Object. In Section 6 we will see that the impact of eliminating DTO classes from the list of data providers increases the accuracy of detecting *Feature Envy*.

5. Tool Support

In this section we present the tool called DATES [4] we have developed in order to automatize the introduced technique. DATES can be used for assessing enterprise applications written in Java where the persistency is provided by SQL relational databases, in particular MySQL and Microsoft Access. The communication with the relational database is performed by executing SQL commands as embedded strings from well-known methods as *executeQuery(String sql), executeUpdate(String sql)*. We have integrated DATES within the IPLASMA [17] environment.

The identification of roles within enterprise applications requires the following steps:

- Construct the *model of the system* based on MEMORIA [26] meta-model. At this phase, the model of the system is not enriched with specific information regarding the interaction between the object-oriented and relational paradigms found in enterprise systems.
- Load the DATES tool in the INSIDER [17] front-end. At this moment the model of the system is enriched with specific information of enterprise applications'design, both for the relational paradigm (tables, columns, primary and foreign keys – extracted from the schema of the database) as well as for the object-oriented one (*e.g.*, accesses to database tables from the bodies of methods within the data source layer)
- Run the *identification algorithm* upon the analyzed system.

In IPLASMA the detection techniques for identifying design flaws are implemented as Java classes. In order to find the entities affected by a particular design flaw we apply the corresponding detection technique on the model of the analyzed system. When DATES is loaded into the INSIDER front-end it will override the existing implementations of detections for finding entities affected by Data Class and Feature Envy design flaws in "regular" object-oriented applications with the ones suitable for enterprise applications presented in Section 4.

6. Evaluation of the Approach

In order to evaluate the approach we have conducted different experiments on a suite of enterprise applications. In this section we present the results obtained by applying the Tool Support described in Section 5 upon three enterprise applications. The size characteristics of the systems are summarized in Table 1.

System	Size(bytes)	Classes	Methods	Tables
KITTA	212,553	37	254	10
TRS	537,058	54	500	10
Payroll	577,323	129	657	12

Table 1. Characteristics of the case studies.

6.1. Identification of Roles

In this section we present the results of applying the algorithm presented in Figure 2 for the identification of roles upon the analyzed applications. In order to give a view about the size of the data source layer, we present in the first line of Table 2 the number of classes that were mapped to the data source layer according to the rules defined in Section 3.4. As a result, we identified a total of 12 classes that have precise design roles. 11 of these classes come from the **Payroll** case-study and are identified as being a *Table Data Gateway* (TDG). One further class from the **TRS** system is also mapped to the TDG role.

	KITTA	TRS	Payroll
Data Source Classes	9	10	15
Table Data Gateway	0	1	11
Row Data Gateway	0	0	0
Active Record	0 (2)	0	0

Table 2. Identified roles.

At first sight the surprising thing is that our approach did not identify any class as being a *Row Data Gateway* or *Active Record*. In order to check if this is due to the rather small size of the case-studies or if it is a conceptual problem, we performed an in-depth analysis, partially manually and partially supported by the IPLASMA [17] analysis environment. We analyzed in particular the classes from the data source layer with a special focus on those for which no mapping to a role was made. The findings and particularities of each of the three systems are discussed next.

The KITTA application. In this system one class from the 9 belonging to the data source layer belongs also to the domain and presentation layers, so it breaks the steady rule about the dependencies between the main layers. 5 classes

4 Design Analysis Tool for Enterprise Systems

0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

from the data source layer belong both to the data source and domain layers and access two or more tables, so definitely they do not have one of the presented roles. The rest of the remaining classes (3) look very much like *Active Records* but they were not identified because they break the condition regarding the isomorphic schema between the attributes of the class and the columns in the accessed table. At this point we decided to relax the condition about the isomorphic schema and consider a class as being an *Active Record* if between its attributes and the columns in the accessed table is a match greater than 80%. Using this more loose condition, 2 out of the 3 classes from the application were identified as *Active Records*.

The TRS application. There, applying the aforementioned relaxation of the rule for *Active Record*'s identification did not improve the number of classes mapped to roles. Apart from that, we found 2 classes that belong to all the three main layers and the other remaining 7 belonging both to the domain and data source layers. Again, some of the classes were accessing more than one table, while among those that access only one table the matching level between attributes and table columns was under 80%.

The Payroll application. In this application only 4 classes from the data source layer were not classified as having a role. Here we discovered that the pre-condition about the access of a table by a single class (see Section 3.2) is the reason why 2 further were not identified as being a *Table Data Gateway*.

Conclusive Remarks. The first conclusion is that, living in a less than perfect world, the identification rules must be less strict in order to also capture the cases where the particularities of a specific role are slightly altered.

The KITTA and TRS applications showed us that in some cases the basic rule for designing enterprise applications (*i.e.*, the separation between layers) is brutally broken. Thus, the identification rules work fine only if the application is conforming (at least intentionally) to the general design rules and practice defined for enterprise applications.

At the same time, the previous remarks raised the idea of detecting these specific design flaws using the same design information (of course, in a different algorithm) that was used in this case for the identification of roles.

Last, but not least, we noticed that although the size of the analyzed systems makes them appealing, – as it allowed us to inspect them also manually – the systems are too small and it is mandatory to conduct further case-studies.

6.2. Roles-Aware Detection of Design Flaws

In Section 4 we proposed an enhanced detection technique for two design flaws that deliver more accurate results on enterprise applications. Next, we discuss the results obtained by detecting the design flaws using first the initial detection rules found in [21] and then the enhanced version described in Section 4.

	KITTA	TRS	Payroll
Data Class	4	8	15
Data Transfer Object	0	2	11
Revised Data Class	4	6	4

Table 3. Detection of Data Class.

The first line of Table 3 presents the number of Data Classes from the applications obtained by considering each analyzed enterprise application as being a ”regular” object-oriented one. The dates from the second line of Table 3 are obtained by applying the *Data Transfer Object* identification presented in Section 4. By putting in correspondence the identified roles in each application with the first line of Table 3, the results does not surprise us because when we encounter a class whose role is a *Table Data Gateway* we are aware that is possible to have *Data Classes* with a special role. Using the enhanced detection technique for identifying Data Classes, *Data Transfer Object* classes will not be detected as being affected by the design flaw, as we can see in the last row of Table 3.

	KITTA	TRS	Payroll
Feature Envy	1	7	48
Revised Feature Envy	1	2	9

Table 4. Detection of Feature Envy.

In Table 4 we present the number of methods from each analyzed system that are affected by the *Feature Envy* design flaw. Similar to the previous table, the first line contains the results obtained by disregarding the fact that the system is an enterprise one. It was no surprise that Payroll was the system where most *Feature Envy* methods were detected, when the ”classical” detection technique applied, due to the fact that it has 11 *Data Transfer Objects* (which is also a consequence of the large number of *Table Data Gateway* classes identified before).

When we applied the revised *Feature Envy* detection technique (see Section 4.2) the accesses to the attributes of *Data Transfer Objects* classes were not anymore counted. Consequently, the large majority of the methods that were apparently affected by *Feature Envy* disappeared from the reported suspects because they were either used for storing/retrieving data in/from the database (methods from the data source layer) or they were manipulating the data from

IEEE COMPUTER SOCIETY

the database (methods from the domain layer).

Conclusive Remarks. The performed case studies reveal that roles have a big impact on eliminating "noise" (*i.e.*, false positives) from "classical" detection rules.

In order to obtain accurate results when we apply "regular" object-oriented detection techniques on enterprise applications we need to analyze the impact of roles upon the detection techniques for other design flaws.

The results encourage us to extend the number of identifiable roles, as this will be definitely necessary for improving the accuracy of the detection for further design flaws.

7. Related Work

We dedicate this section to a briefing of several representative solutions that fall in (or are closely related with) the assessment for enterprise applications.

Embedding within the source code of enterprise applications SQL statements as strings that are sent in order to be executed upon a SQL relational databases to well-known methods as *executeQuery(String sql)* from Java makes impossible their correctness verification from the syntactical point of view at compile time. As a result, runtime errors might appear. In [13] some methods that address the problem of syntactical verification of SQL incorporated statements are presented. In [24] is proposed an automatic generator for classes to be used for the relational database manipulation. The same issue is addressed in [4]. In [7] is proposed a tool set for testing relational database applications.

All the aforementioned techniques help us in order to ensure the *correctness* of the communications performed within the entities from the data source layers. Unlike the previous approaches the aim of our approach, by identifications of roles, is to increase the level of understanding of the enterprise applications'design and the accuracy of object-oriented problem detection techniques, when applied to enterprise software systems.

Increasing the level of understanding in software systems by identifying features (roles) in the source code is not a new technique. For example, in [28] is introduced a concern graph for finding and describing concerns (subsets of a program source code activated when exercising a functionality) using structural program dependencies in object-oriented applications. Our approach, by extracting from the source code the dependencies between object-oriented design entities (e.g. classes, methods) and relational entities (e.g., tables, columns, primary and foreign keys), makes possible the identification of a complementary set of features specific for enterprise applications.

When we extracted the design information specific to the relational part of the application we presumed that the information regarding semantics of attributes, primary keys and foreign keys in the database tables is complete [6] and can be derived directly from the database tables. Recently, Yeh and Li states in [29] that sometimes this may not be the case and they propose an approach where various procedures, like field comparison, data analysis, code analysis are applied in order to determine the semantics of attributes, followed by the identifications of primary keys, foreign keys and cardinality constraints. A tool specifically design for database reengineering in also proposed in [5].

In enterprise applications we can find the persistency level implemented using the EJB framework. In [16] were introduced several quality attributes design primitive associated to a JavaBean distributed enterprise system. They are architectural building blocks that target the achievement of quality attribute requirements like performance, modifiability, reliability and usability. Having performance and scalability upon an EJB application is also tackled in [2]. None of the previous approaches *modify* quality assurance techniques specific to "regular" object-oriented systems in order to make them suitable for enterprise applications.

8. Conclusions. Future Work

This paper is a step forward in design understanding and quality assessment of enterprise applications by taking into account their particularities that distinguish them from "regular" object-oriented systems. The main features of the introduced approach are:

- It allows us to identify automatically roles which design entities (classes and methods) might have within an enterprise application. The identification helps understanding the specific purpose of each entity and, in the same time, helps localizing within the system all design fragments that have the same specific role.
- It increases the accuracy of the detection of two well-known design flaws (Data Class and Feature Envy) by making them take into account the identified design roles that entities might have. This way, design related analyses for "regular" object-oriented became suitable when applied to enterprise applications.

The aforementioned features required the extension of the meta-model used for storing information about a "regular" object-oriented application in order to make it support additional design information related to enterprise applications.

We conducted an experiment in which we identified roles in enterprise applications by using the extraction algorithm presented in Section 4. The performed experiment also emphasizes the obtained enhancement upon the detection of design flaws by comparing the obtained results by considering the applications as being "regular" object-oriented, respectively enterprise .

We will focus our future work on the next fronts:

- We intend to define and automatize a comprehensive suite of specific quality assessment analyses for enterprise applications (*e.g.*, analyses related to the separation between the main layers).
- We intend to continue the evaluation of the introduced approach against other enterprise applications and experimenting the extraction of Row Data Gateway and Active Record roles using different thresholds regarding the isomorphic schema between the class and the accessed table within the class.
- Due to the fact that roles proved to have a big impact on eliminating false positives from ”classical” detection rules, we are going to extend the number of identifiable roles in order to analyze their impact on other detections of design flaws (*e.g.*, Shotgun Surgery [10]).

Acknowledgments This work is supported by the Romanian Ministry of Education and Research under Project CEEX No. 3147/11.10.2005. I would like very much to thank Tudor Gîrba, Radu Marinescu and Petru Mihancea for all their encouragement and fruitful discussions. I would also like to thank Diana Flacăr for helping me with a lot of the implementation effort related to my research. Last but not least I would like to thank the LOOSE Research Group (LRG) for being such a great team.

References

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practicies and Design Strategies.* Prentice Hall, 2001.

[2] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *Proc. OOPSLA*, 2002.

[3] O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. In *Technology of Object-Oriented Languages and Systems*, 1999.

[4] W. Cook and S. Rai. Safe query objects: Statically typed objects as remotely executable queries. In *Proc. International Conference on Software Engineering*, 2005.

[5] I. de Guzman, M. Polo, and M. Piattini. An integrated environment for reengineering. In *Proc. IEEE International Conference on Software Maintenance*, 2005.

[6] M.L. Pedro de Jesus and P.M.A. Sousa. Selection of reverse engineering methods for relational databases. In *Proc. European Conference on Software Maintenance and Reengineering*, 1999.

[7] Y. Deng and D. Chays. Testing database transactions with agenda. In *Proc. International Conference on Software Engineering*, 2005.

[8] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. IEEE International Conference on Software Maintenance*, 1999.

[9] K. Brown et al. *Enterprise Java Programming with IBM Websphere.* Addison-Wesley, 2001.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[11] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[13] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. International Conference on Software Engineering*, 2004.

[14] W. Keller. Object/relational access layers: a roadmap, missing links and more patterns. In *Proc. European Conference on Pattern Languages of Programming and Computing*, 1998.

[15] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *OOPSLA 2001 proceedings*, 2001.

[16] A. Liu, L. Bass, and M. Klein. *Analyzing Enterprise JavaBeans Systems Using Quality Attribute Design Primitives.* Technical Note CMU/SEI-2001-TN-025., 2001.

[17] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Raţiu, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *Proc. IEEE International Conference on Software Maintenance(Industrial and Tool Volume)*, 2005.

[18] F. Marinescu. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms.* Wiley Computer Publishing, 2002.

[19] R. Marinescu. Using Object-Oriented Metrics for Automatic Design Flaws in Large Scale Systems. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*. Springer-Verlag, 1998.

[20] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, ”Politehnica” University of Timişoara, 2002.

[21] R. Marinescu. Detection strategies:metrics-based rules for detecting design flaws. In *Proc. IEEE International Conference on Software Maintenance*, 2004.

[22] R. Martin, D. Riehle, and F. Buschmann. *Pattern Languages of Program Design 3.* Addison-Wesley, 1998.

[23] R.C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[24] R. McClure and I. Kruger. Sql dom: Compile time checking of dynamic sql statements. In *Proc. International Conference on Software Engineering*, 2005.

[25] C. Nock. *Data Access Patterns: Database Interactions in Object-Oriented Applications.* Addison-Wesley, 2003.

[26] D. Ratiu. *Memoria: A Unified Meta-Model for Java and C++.* Master Thesis, Politehnica University of Timişoara, 2004.

[27] A.J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[28] M.P. Robillard and G.C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. International Conference on Software Engineering*, 2002.

[29] D. Yeh and Y. Li. Extracting entity relationship diagram from a table-based legacy database. In *Proc. European Conference on Software Maintenance and Reengineering*, 2005.

IEEE COMPUTER SOCIETY