Automated Detection of Likely Design Flaws in Layered Architectures

Aditya Budi, Lucia, David Lo, Lingxiao Jiang, and Shaowei Wang
School of Information Systems, Singapore Management University
{adityabudi,lucia.2009,davidlo,lxjiang,shaoweiwang.2010}@smu.edu.sg

Abstract

Layered architecture prescribes a good principle for separating concerns to make systems more maintainable. One example of such layered architectures is the separation of classes into three groups: Boundary, Control, and Entity, which are referred to as the three analysis class stereotypes in UML. Classes of different stereotypes are interacting with one another, when properly designed, the overall interaction would be maintainable, flexible, and robust. On the other hand, poor design would result in less maintainable system that is prone to errors. In many software projects, the stereotypes of classes are often missing, thus detection of design flaws becomes non-trivial. In this paper, we provide a framework that automatically labels classes as Boundary, Control, or Entity, and detects design flaws of the rules associated with each stereotype. Our evaluation with programs developed by both novice and expert developers show that our technique is able to detect many design flaws accurately.

1 Introduction

Layered architecture is a recommended industry practice as it promotes separation of various concerns into layers [17]. By using this architecture, when requirements change, most of the changes could be localized to a limited number of classes in a particular layer. Thus, no changes would be needed for classes in unrelated layers as long as the interfaces between the layers remain the same. As software evolves over time, layered architectures are more likely to have better reusability, improve comprehension and traceability, and ease maintenance and evolution tasks than single-tier architectures.

One commonly used layered architecture is the separation of classes into three stereotypes, namely: Boundary, Control, and Entity, following the Unified Modeling Language (UML) and its suggested objectory process [15, 14]. *Boundary* classes are responsible to interface with external systems or users. *Control* classes are responsible to realize particular functionalities or use cases by coordinating the activities of various other classes. *Entity* classes are responsible to model various domain concepts and store and manage system data.

Class stereotypes are not just symbols; they come with design rules governing their behaviors and responsibilities. If the nature of a class is not apparent to developers or its behaviors do not match its stereotype label, developers are prone to make mistakes, violating the rules, especially as the code evolves over time. There are two common rule variants: *robustness rules* [15], and *well-formedness rules* [14].

Unfortunately, many software projects, during development or maintenance, have little documentation. Many design documents, including those specifying stereotype labels of the classes in a program, are often missing. Such information is often not obvious in the source code either due to poor variable and class names, code changes, etc. Also, keeping design documents and stereotypes up-to-date manually could be time-consuming and error-prone.

To address the above issues, we propose a framework that can automatically reverse engineer class stereotypes and detect violations of design rules associated with them.

We empirically evaluated our proposed system on a number of student projects and a real software system. Our preliminary experiments are promising. Compared with manually stereotyped labels, our approach achieves on average 77% of accuracy. Design defects resulted from violations of robustness and well-formedness rules could be detected with up to 75% precision and 79% recall.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 presents the concept of class stereotypes and their associated design rules. Section 4 describes our design flaw detection framework. Section 5 presents evaluation results. We discuss limitations and applicability in Section 6 and conclude in Section 7.

2 Related Work

There are a number of studies on the characteristics of class stereotypes [2, 8, 1, 12]. Andriyevska et al. study the effect of stereotypes on program comprehension [1]. Kuz-

niarz et al. also study the effects of stereotypes on program comprehension but focus on user-defined stereotypes rather than the standard three (i.e., Boundary Control, and Entity) [12]. Atkinson et al. propose different de facto ways in which stereotypes are used [2]. Gogolla and Henderson-Sellers analyze the part of the UML metamodel that deals with stereotypes and provide recommendations for improving the definitions and uses of stereotypes [8]. Dragan et al. investigated an automated way to infer class and method stereotypes [6, 7]. To the best of our knowledge, we are the first to propose an automated way to automatically detect likely violations of design rules governing stereotypes based on automatically identified stereotype labels.

Various studies also address the problem of detecting and correcting design flaws and code smells [9, 11, 19, 13, 18]. Guéhéneuc et al. [9] find code segments that do not conform to a particular design pattern and transform them accordingly. Khomh et al. use Bayesian Belief Networks to detect code and design smell [11]. Vaucher et al. study *god* classes and propose an approach to distinguish good *god* classes from bad ones [19]. Moha et al. extract concepts from text descriptions and establish formal specifications of code smells so that they can detect code smells automatically [13]. Trifu and Reupke also detect structural flaws in object oriented programs and use optimization-based techniques to automatically restructure programs [18]. Our work in this paper focuses on detecting class stereotypes and checking violations of design rules involving stereotypes, which enriches the type of design information and defects detected by past studies and helps users to reverse engineer their designs.

3 Design Rules of Class Stereotypes

This paper provides a mechanism to identify class stereotypes automatically and detect design flaws in programs. The class stereotypes and design rules associated with the stereotypes are described in the following subsections.

3.1 Class Stereotypes

Identifying class stereotypesis an important step for designing, analyzing, understanding, and maintaining a software system. In particular, this paper focuses on automated identification of three class stereotypes, i.e., *Entity*, *Control*, and *Boundary*, which was introduced as an extension to the standard UML [14, 3]. The UML extension describes the responsibilities of classes belonging to each stereotype. It promotes separation of different concerns into different class stereotypes, and thus software changes related to one concern would only affect one particular stereotype involving a limited number of classes [17].

Classes with the *Entity* stereotype store and manage information in a system. In this paper, we further distinguish *Regular Entity* from a special kind of entity called *–Data Manager*, which is used to persist to storage systems (e.g., databases, file systems, etc). As an example, *Course* is a possible entity class in a University Management System (UMS) and *CourseStore* is a possible data manager class that stores and retrieves course data from databases.

Classes with the *Boundary* stereotype serve as an interface between a system and external systems interacting with it. External systems, represented as *Actors*, could be other computing systems or the users of the system. These interface classes would be the ones affected if the behavior of external systems change. In a typical UMS, *CourseManagementUI* is a possible boundary class.

Classes with the *Control* stereotype act as a glue among entity and boundary classes, and control the activities of other classes for particular tasks. For example, *CourseRegistration* class is a possible control class that interacts with a user interface class and related entity classes, e.g. *Course*.

3.2 Design Rules

Class stereotypes, based on their supposed responsibilities and the principle of separation of concern, should follow certain design rules, such as, an *Entity* class cannot call a *Boundary* class directly. Regulating the interactions among different classes of various stereotypes can help to ensure the understandability and maintainability of a software system.

Our work provides an automated mechanism for checking compliance of design rules governing the interactions among class stereotypes. In particular, we instantiate the checking against two sets of rules which reflect various architectural styles, namely *robustness rules* and *well-formedness rules*. Our checking mechanism is designed to be flexible enough to take various rules for checking.

3.2.1 Robustness Rules

Robustness analysis, as described in Rosenberg and Scott's UML book [15], provides a set of rules that indicate all valid and invalid interactions among different class stereotypes. The rules are paraphrased as follows, where *Actors* represent users of a system which could be humans or classes/objects outside the system under analysis

- R1 Actors can only call boundary objects.
- R2 Boundary objects can only call controllers or actors.
- R3 Entity objects can only call controllers.
- R4 Controllers can call entities, other controllers, and boundaries, but not actors.

3.2.2 Well-Formedness Rules

The *well-formedness rules* are defined in the UML extension [14], and rephrased as follows: [1]

[1] The complete set of rules in the UML extension also allows the subscriber-publisher style of interaction. This paper considers only interactions via direct calls and thus omits a part of the rules governing subscribe-publish interactions.

Caller	Callee							
	Actor		Entity		Control		Boundary	
Actor							R1	W1
Entity				W3	R3			
Control			R4	W4	R4	W4	R4	W4
Boundary	R2	W2		W2	R2	W2		W2

Table 1. Robustness and Well-Formedness Rules

W1 Actors can only call boundary objects.
W2 Boundary objects can call entities, controllers, other boundaries, and actors.
W3 Entity objects can only call other entities.
W4 Controllers can call entities, other controllers, and boundaries, but not actors.

The two sets of rules can also be represented as a matrix shown in Table 1. Each entry indicates a rule that validates the corresponding caller-callee relation. Unmarked entries signify bad caller-callee relations that violate the rules.

4 Design Flaw Detection Framework

Our framework are shown as a flowchart in Figure 1. We use a classification framework that has two phases, namely training and violation detection. In the training phase, we build a statistical model using a machine learning technique that can discriminate the class stereotypes based on a set of classes with given stereotypes, i.e., training data labeled with Regular Entity, Data Manager, Control, and Boundary stereotypes. In the violation detection phase, given a class or a program containing more than one class with no stereotype labels (i.e., test data), we first predict the corresponding stereotype for each class based on the model. We then detect design defects by using the inferred stereotypes verified against either robustness or well-formedness rules.

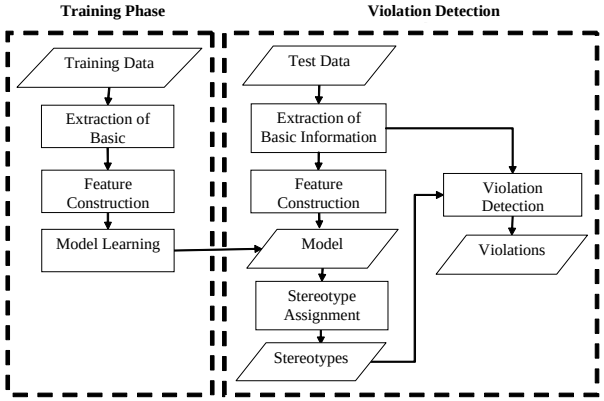

Figure 1. Design Flaw Detection Framework

There are five processes in the framework: extraction of basic information, feature construction, model learning, stereotype assignment, and violation detection. The following paragraphs describe these processes in more detail.

Extraction of Basic Information. In this process, we extract information about the classes in each Java program from its source code. The basic information we extract includes all the methods in each class, all the instructions contained in each method, the call-relations among classes (represented as call graphs), and the classes that contain operations related to I/O or database operations. We built our information extractor upon WALA [20].

Feature	Description
Size	Number of instructions that a class has
NOM	Number of methods that a class has
ASize	Average size of all the methods in a class
Fan-out	Number of other classes that a class calls
Fan-in	Number of other classes that call a class
GetCnt	Number of getters method in a class
SetCnt	Number of setters method in a class
CRUD	Number of methods performing create, read, update, or delete to data sources in a class

Table 2. Features Used in Our Statistical Model.

Feature Construction. Based on the basic information, we form features that could help in differentiating the training classes belonging to each of the four given stereotypes. In this work, we compute the set of features shown in Table 2 for each class. Instead of using absolute values for the features, we normalize their values to be in the $[0, 10]$ range.

Model Learning. In this process, we take the training data with its features and learn a model that could discriminate the four stereotype labels: Regular Entity, Data Manager, Boundary, and Controller. We use Support Vector Machine (SVM) [5] for this task since it is a well-known machine learning technique that has been shown to have good accuracies in many application domains. Regular SVMs learn models that only discriminate between two labels. We use an SVM extension handling multiple class labels [4]. Implementation-wise, we use the publicly available $SVM^{multiclass}$ [16].

Stereotype Assignment. We use the model learned in the training phase and the features extracted from the test data to assign stereotype labels to each class in the test data. We use the classification capability of $SVM^{multiclass}$.

Procedure Violation Checking
Inputs:
R : A set of rules (e.g., robustness, well-formedness)
C: A set of classes
L: The corresponding stereotypes for the classes
Output: V: A set of violations against R
Method:
1: Let V = {}
2: Let $RMap$ = Process R and represent it as a pair $\langle Caller, \{Callee\}\rangle$, where $\{Callee\}$ is the set of stereotypes that can be called by the stereotype $Caller$ as expressed in R.
3: **For each** class c in C
4: Let $Caller$ = c's stereotype
5: Let $\{Callee\}$ = $Caller$'s information in $RMap$
6: Let C' = All other classes that are called by c
7: **For each** class c' in C'
8: **If** c''s stereotype $\notin \{Callee\}$
9: $V \leftarrow V + \{c'\}$ // A violation is found
10: **OUTPUT** V

Figure 2. Violation Detection

Violation Detection. After the stereotypes are inferred, we can check for violations against a set of class design rules by

leveraging the caller-callee relations extracted from code, and the inferred stereotypes. In this paper, we consider *robustness rules* [15] and *well-formedness rules* [14]. For a set of rules, the automated rule checker performs the steps shown in Figure 2 to search for violations. At line 1, we initialize the output set. At line 2, we represent a set of rules as a set of pairs of valid interactions from a stereotype (i.e., classes with this stereotype) to other stereotypes. At line 3-10, we visit each class and for each, we extract its stereotype (line 4), other valid stereotypes that it could call (line 5), and the set of other classes called by it (line 6). At line 7-8, we check if any of the called classes has a stereotype that violate the rule (i.e., not in the set $\{Callee\}$). If this is the case, we record this violation at line 9. We finally report all violations found (line 10). For example, for the robustness rule R3, given a class with stereotype *Entity*, any call from the class to other Entity or Boundary classes will be reported as a violation. As another example, for the well-formedness rule W4, any call originated from a Control class is valid.

5 Empirical Evaluation

In this section, we describe our dataset and evaluate the accuracies of our approach in inferring stereotypes and detecting design flaws.

5.1 Dataset

We perform our evaluation on 15 Java projects developed by students of an object-oriented application development (OOAD) course. The projects are all about a single player hunting game. The number of Java classes per project ranges from 36 to 67 with an average of 45. Each project has 3431 to 9220 lines of code (including comments and blank lines), with an average of 5168. We also perform experiment using a real open source software namely OpenHospital, which is a hospital management system. The system consists of 233 classes, with 59,087 lines of code.

For each project, we manually labeled the classes with either *Boundary*, *Control*, *(Regular) Entity*, and *Data Manager*. The manual labels provide us valid classes stereotypes for the training phase and an oracle to measure the accuracy of our approach in the testing phase.

5.2 Accuracy of Stereotype Inference

We employ ten-fold cross validation to evaluate our approach. It divides all data points (i.e. classes in a project) into ten disjoint subsets of (approximately) equal size. To obtain a representative training data, the classes of the same stereotype are distributed over the subsets. Then, one subset is used as test data, while the others are used as training data. This process is repeated ten times (iterations); each iteration uses a different subset as test data.

We evaluate the accuracy of a trained model in inferring stereotypes as a ratio of number of correctly inferred class stereotypes with number of classes in test data. We compute the accuracy for each iteration for each project and average them as the accuracy of each project. The accuracy of our approach is then computed by taking the average of the accuracies of all projects, which is 77%.

Real Vs. Inferred Label	Number and Proportion of Predicted Classes			
	Boundary	Control	Entity	Data Man.
Boundary	81.35%	9.84%	1.04%	7.77%
Control	10.37%	58.54%	23.78%	7.32%
Entity	3.21%	2.88%	92.31%	1.60%
Data Manager	18.33%	6.11%	12.22%	63.30%

Table 3. Confusion Matrix of the Stereotypes Inferred

```
1  public class InventoryController{
2  private TrapDataManager trapDM;
3  private BaitDataManager baitDM;
4  private PlayerDataManager playerDM;
5  public InventoryController(){
6  {    trapDM = TrapDataManager.getInstance();
7       baitDM = BaitDataManager.getInstance();
8       playerDM = PlayerDataManager.getInstance(); }
9  public void setTrap(Player p, int trapID)
10 {   trapDM.setTrap(p, trapID); }
11 public ArrayList<Trap> retrieveAllTraps(String username)
12 {   return trapDM.retrieveAllTraps(username); }
13 public void setBait(Player p, int baitID)
14 {   baitDM.setBait(p, baitID); }
15 public ArrayList<Bait> retrieveAllBaits(String username)
16 {   return baitDM.retrieveAllBaits(username); }
17 public ArrayList<InventoryItem> retrieveAllInventory(String username)
18 {   return playerDM.retrieveAllPlayerInventory(username); }
19 public void readPlayerChoice(Player p, String choice)
20 {   ...
21      if (tOrBChoice == 'T')
22      {    InventoryUI inventoryUI = new InventoryUI(); ...}
23      else if (tOrBChoice == 'C'){
24      {          InventoryUI inventoryUI = new InventoryUI(); ...}
25      else{      InventoryUI inventoryUI = new InventoryUI();
26                 System.out.println("Please enter a VALID item ID > "); }}
27 public String getBaitInUse()
28 {   return baitDM.getBaitInUse(); } }
```

Figure 3. Example of a Wrongly Labeled Control Class

We draw a *confusion matrix* to evaluate the accuracy of each stereotype prediction produced by the trained model [10]. A confusion matrix is a table with rows corresponding to real labels and columns corresponding to inferred labels. A cell (X,Y) in the matrix corresponds to the number of test data points with real label X that are assigned label Y by a classifier/model. Table 3 shows the accuracy of the inferred stereotypes in percentages.

Considering the diagonal entries of the matrix, we notice that boundary and entity classes can be detected with very good accuracies of more than 80%. However, it is less accurate when assigning labels to control classes. Control classes are often confused with entity classes. Upon inspection, we find that many students implement their control classes poorly. For example, consider the control class named *InventoryController* in Figure 3. It is assigned an entity stereotype by our approach. This is the case, as all of

the methods in this class except *readPlayerChoice* method perform either get data operation (e.g., *retrieveAllTraps*) or set data operation (e.g., *setTrap*). The control class simply delegates the execution of these operations to the respective data manager classes.

5.3 Accuracy of Design Flaw Detection

After the labels are inferred, we can detect design flaws as violations of the robustness and well-formedness rules. In this subsection, we show sample detected violations and analyze the quality of our violation detection mechanism.

Sample Violations. Figure 4 shows an example where violations occur in a Boundary, a Control, and an Entity.

According to the robustness rule R2, a boundary can only call controllers or actors. We detected a violation of R2 in Code-1: the boundary class named `RegistrationPage` calls an entity class named `RegistrationManager` (lines 8 and 13). Note that this is not a violation when we check it against the well-formedness rule W2.

Both robustness and well-formedness rules allow a controller to interact with any class but not actors. In Code-2, we detect a violation of the rules: the controller class named `SendingController` calls `System.out.println` (lines 6, 8, and 12) to display a message directly to a user (i.e., an actor) and uses `Scanner(System.in)` (line 16) to elicit inputs directly from the user.[2]

Robustness and well-formedness rules deal differently with entity classes. The robustness rule R3 allows an entity to call only controllers, while the well-formedness rule W3 allows an entity to call only entities. Code-3 of Figure 4 shows that an entity class named `Player` violates both of the rules: `Player` uses another entity `Inventory` (line 4) and thus violates R3; It also uses a controller `StarbugsController` (line 8) and thus violates W3. In addition, this class interacts with an actor directly via `System.out.println` (line 15), violating R3 and W3.

Quality of Detected Violations. With correct stereotype labels, our checking mechanism will detect all violations perfectly (i.e., no false positives or negatives) as both robustness and well-formedness rules are well specified. However, since stereotypes inferred by our approach could be wrong, we may detect wrong violations (false positives) or miss some violations (false negatives).

To measure false positives and negatives, we can simply compare the violations detected with inferred labels against those detected with correct labels: The size of the intersection of the two sets relative to the sizes of the two sets are indicative of false positive rates and negative rates, which can be measured by the notion of precision and recall. Precision is the ratio of inferred violations that are true and recall is the ratio of true violations that are inferred.

[2] We have (manually) predefined a list of classes and functions that send or receive messages to users which are treated as actors'.

	Class & Type		Class Only	
Evaluation	**Rob.**	**Well.**	**Rob.**	**Well.**
Precision	61.2%	74.6%	68.6%	69.9%
Recall	68.7%	61.8%	78.8%	59.8%

Table 4. Precision and Recall of Detected Anomalies for Robustness Analysis (Rob.) and Well-Formedness Analysis (Well.)

$$Precision = \frac{\|\{Inferred\,Violations\} \cap \{True\,Violations\}\|}{\|\{Inferred\,Violations\}\|}.$$

$$Recall = \frac{\|\{Inferred\,Violations\} \cap \{True\,Violations\}\|}{\|\{True\,Violations\}\|}.$$

When comparing violations during the intersection operation, we consider two equivalence criteria. One criterion considers two violations are matched only if both the violating class (i.e., the class where the violation occurs) and the type of the violation are matched (Class & Type). The other considers only the violating class (Class Only).

The total numbers of true violations of robustness and well-formedness rules are 244 and 138 respectively. The total numbers of inferred robustness and well-formedness violations are 255 and 112 respectively. The overall precision and recall of our approach is shown in Table 4. The precision and recall values are aggregated averages across many model building and testing iterations. We calculate them using the two equivalence criteria.

Table 4 shows that violating classes (Class Only) can be detected with precision and recall of 68.7% and 78.8% (robustness), and 69.9% and 59.8% (well-formedness). When considering both violating classes and violation types (Class & Type), the precision and recall are reduced by 7.5% and 10.1% (robustness), and increased by 4.7% and 2% (well-formedness), due to some violations are detected with correct violating classes but wrong violation types.

6 Discussion

Effects of Features Used. We used eight code features in our experiments. It would be interesting to consider other features, such as code complexity metrics, which might help to improve the accuracy of the stereotype inference further. The confusion matrix shown in Section 5.2 particularly suggests more features related to controllers should be used to reduce the number of confusion occurrences.

Effects of Dataset Investigated. We perform experiments on software systems written by novice programmers and one real medium-sized software system. These systems are chosen based on the availability of the class labels. None of the processes in our framework is expensive: basic information extraction, feature construction, model learning, and stereotype assignment make use of an inexpensive static analysis technique and a scalable classification engine, i.e., SVM. We believe the framework is able to process larger programs. We plan to analyze larger systems in the future.

Effects of Design Rules Checked. We detected violations against only robustness and well-formedness rules. There

Code - 1

```
1  public class RegistrationPage{
2   ...
3   private RegistrationController RC=new
                                RegistrationController();
4   private RegistrationManager RM=RC.getRM();

5   public RegistrationPage()
6   {   Scanner sc= new Scanner(System.in); ....
7       username= sc.next();
8       if(RM.usernameIsAvailable(username)) {
9         ...
10        String password =
11        String confirmation
12        if(confirmation.equals(password)) { ....
13           RM.processRegistration(username,password);
14         }else { ... }
15       }else{
16       System.out.println(username+" is already in use");
17  }}}
```

Code - 2

```
1  public class SendingController
2  {
3    public void displayPlayer()
4    {   ArrayList<Player> playerList =
                PlayerDataManager.retrievePlayers(); ...
5      if(playerList.size() < 2)  {
6        System.out.println("There is no other
7       } else {
8        System.out.println("Which user ...");
9        for (int i = 0; i < playerList.size(); i++)
10       {  Player aPlayer = playerList.get(i);
11         if(
12         { System.out.println(... aPlayer.getName());
13           playerID++;  } }}}
14   public void displayGiftOption(Player recipient)
15   {  ...
16     Scanner sc = new Scanner(System.in); ...
17   }
```

Code - 3

```
1  public class Player {
2  ...
3  public Player(String username, String password, String rank, String
     region, int experience, int gold)
4  {  ...  inventory = new Inventory(); }

5  public int getGold()
6  {  try
7    {  Player p = this;
8     StarbugsController starbugsController = new
9     ArrayList<BaitLineItem> alBli = new ArrayList<BaitLineItem>();
10    alBli = p.getInventory().getBaits();
11    BaitLineItem currentBait = null;
12    currentBait = p.getInventory().getBait();
13    if (alBli.isEmpty() && !(currentBait != null)) {
14       if (this.gold < 50) {
15          ... System.out.println("Your gold ...");
16       } } catch (Exception e) {}
17    return this.gold; }
```

Figure 4. Violations in Boundary, Controller, and Entity Stereotypes

are other design rules, for example, some design rules relax the robustness rules by allowing direct interactions between Entity objects. Our approach can be easily extended to handle such variants. However, if the design rules involve constraints such as conditional call-relations, our violation checking mechanism would then need further improvements, such as, taking control-flow conditions embedded in inter-procedural call graphs into consideration.

Threats to Validity. To reduce the threats to construct validity, we used standard evaluation metrics, namely accuracy, precision, and recall, which are commonly used in data mining and information retrieval tasks. However, it remains a question what is the effect of inaccuracies on program comprehension. To answer this question, a user study would be needed and is left as future work. To reduce the threats to internal validity and selection bias, the 15 projects used in the experiment are chosen randomly from a pool of 93 student projects. However, as aforementioned, there are still threats to external validity on the generalizability of our results. We plan to evaluate our framework on various types of software systems of various sizes to alleviate the threats.

7 Conclusion & Future Work

In this paper, we present a framework that detects likely design flaws in layered object-oriented architecture with classes belonging to various stereotypes, which should follow certain design rules. Also, to accommodate to systems without sufficient stereotype annotations, our framework learns a statistical model to distinguish various class stereotypes available from a training set. This model in turn is used to give labels to unannotated classes. Likely design flaws are later detected by finding violations of well-known design rules. We have evaluated our approach on Java projects developed by novice and expert developers. The results show that our approach can identify class stereotypes with 77% accuracy on average and can detect violations of the design rules associated with each stereotype with up to 75% accuracy and up to 79% recall.

In the future, we plan to further investigate more useful features for the inference of class stereotypes and more software systems. We believe our framework is general and can be adapted for reverse engineering other kinds of domain-specific stereotypes.

Acknowledgement. We would like to thank Yeow-Leong Lee for providing some stereotype labels.

References

[1] O. Andriyevska, N. Dragan, B. Simoes, and J. Maletic. Evaluating uml class diagram layout based on architectural importance. In *VISSOFT*, 2005.

[2] C. Atkinson, T. Kuhne, and B. Henderson-Sellers. Systematic stereotype usage. *Software and Systems Modelling*, 2:153–163, 2003.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[4] K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *JMLR*, 2002.

[5] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-Based Learning Methods*. Cambridge, 2000.

[6] N. Dragan, M. Collard, and J. Maletic. Reverse engineering method stereotypes. In *ICSM*, 2006.

[7] N. Dragan, M. Collard, and J. Maletic. Automatic identification of class stereotypes. In *ICSM*, 2010.

[8] M. Gogolla and B. Henderson-Sellers. Analysis of uml stereotypes in the uml metamodel. In *UML*, 2002.

[9] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *TOOLS USA*, 2001.

[10] J. Han and K. Micheline. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2006.

[11] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *QSIC*, 2009.

[12] L. Kuzniarz, M. Staron, and C. Wohlin. An empirical study on using stereotypes to improve understanding of UML models. In *IWPC*, 2004.

[13] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE TSE*, 36:20–36, 2010.

[14] Rational Software et al. *UML Extension for Objectory Process for Software Engineering ver. 1.1*, 1997.

[15] D. Rosenberg and K. Scott. *Use case driven object modeling with UML: a practical approach*. Addison-Wesley, 1999.

[16] `http://svmlight.joachims.org/svm_multiclass.html`.

[17] P. Tarr, H. Ossher, W. Harrison, and S. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.

[18] A. Trifu and U. Reupke. Towards automated restructuring of object oriented systems. In *CSMR*, 2007.

[19] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In *WCRE*, 2009.

[20] `http://wala.sourceforge.net`.