A Middleware-based Approach to Model Refactoring at Runtime

Ling Lan, Gang Huang, Weihu Wang, Hong Mei
Key Laboratory of High Confidence Software Technologies, Ministry of Education
School of Electronics Engineering and Computer Science, Peking University
Beijing, 100871, China
Corresponding to: huanggang@sei.pku.edu.cn, meih@pku.edu.cn

Abstract

Model refactoring is emerging as a desirable means to improve design model by restructuring it while preserving the behavior properties. It applies the concept of refactoring to a higher level of abstraction and makes refactoring more convenient and effective. Model refactoring always arises at design phase, but unfortunately, 7(days) x 24(hours) high availability requires that refactoring takes effect at runtime without stopping the running systems. In this paper, we present a middleware-based approach to applying model refactoring for component based applications at runtime. First of all, ill-structures in an application are abstracted as bad patterns, each of which has at least one good pattern abstracting the refactored part in the application without the ill-structure. People can define the bad/good patterns using a MOF-based metamodel. After that, with the help of middleware, the ill-structures will be automatically detected and removed by refactoring the running application under the guide of predefined patterns.

1. Introduction

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure. It is used to improve the quality of the software, such as extensibility, reusability, efficiency [18]. A recent trend is to apply the concept to a higher level of abstraction. As a result, model refactoring is emerging as a desirable means to improve design model by restructuring it while preserving the behavior properties [8]. Applying refactoring at design phase can make refactoring more convenient and effective. With the help of the information of design, the errors, which are made at design process, can be discovered more easily and fixed with lower cost.

However, considering the large numbers of existing software systems, to apply model refactoring at design phase solely is not enough. First of all, 7 (days) x 24 (hours) high availability becomes a critical, even obligatory, property for large-scale systems. It means that they could be only refactored without stopping the whole systems. On the other hand, even if the system could be stopped, it is very expensive to redevelop, redeploy and restart the system for improving the design models. Though model driven development methods can reduce the cost of redevelopment by automatically regenerating codes from refactored design models, the total cost is still high. More importantly, if the refactoring is ineffective or invalid, roll-back is needed; that is to say, the running system has to be stopped, redeveloped, redeployed and restarted again. In a word, model refactoring should be able to take effect at runtime and be more responsive.

In this paper, we present a middleware-based approach to applying model refactoring for component based applications at runtime. In this approach, ill-structures in an application are abstracted as bad patterns, each of which has at least one good pattern abstracting the refactored part in the application without the ill-structure. A MOF-based metamodel, which is perceptible and understandable for middleware, is provided to define the patterns. As long as patterns defined by users, the ill-structures will be automatically detected and removed by refactoring the running systems with the help of middleware.

The rest of this paper is organized as follows: Section 2 presents an overview of our approach. Section 3 gives an example application and demonstrates how to apply model refactoring at runtime. Section 4 puts forward some discussion about this approach. Related work is discussed in Section 5 and we conclude this paper in Section 6.

1530-1362/07 $25.00 © 2007 IEEE
DOI 10.1109/ASPEC.2007.45

2. Approach Overview

For current component based applications, middleware not only provide a runtime environment to support operation of applications, but also provide the functions to monitor and change the running applications. In our approach, model refactoring is applied at runtime based on the abilities provide by the middleware.

2.1. The metamodel architecture

In order to perform refactoring with the guide of patterns, we need to specify them first. There are three requirements for pattern models in our approach: First, it is obvious that they should be precise; second, considering that patterns are the abstraction of parts within applications, pattern models are supposed to be easily related with application models, which are specified by some popular modeling language such as UML; last but not least, the models need to be middleware-aware so that the pattern detection and pattern-based refactoring can be performed by the middleware automatically.

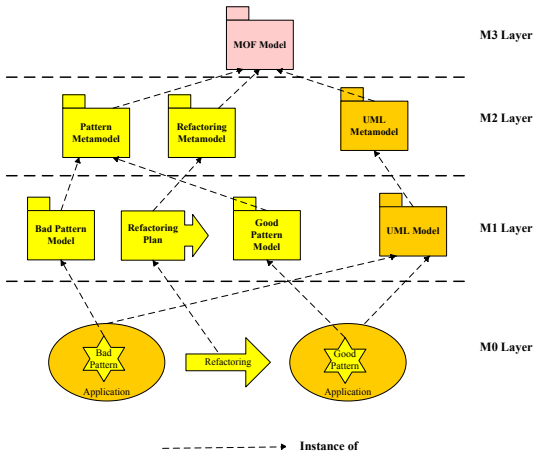

Figure 1. Metamodel architecture of the approach

A metamodel architecture is introduced in our approach to meet the above requirements (Figure 1). It is compatible with the four-layer MOF (Meta Object Facility) metamodel architecture defined by OMG [12]. In the MOF metamodel architecture, MOF (M3) is a meta-metamodel which is used to define metamodels (M2), such as UML. Then models (M1) can be created and the information (M0) which we want to describe, such as running application, is the instance of model. Pattern specification for refactoring at runtime in this architecture is described as the following:

- **M2 layer:** A pattern metamodel and a refactoring metamodel are added in this layer. The two metamodels can be used respectively to define the bad/good pattern models and the refactoring plan between them. The MOF-based metamodels can be used to create precise models of patterns and refactorings. On the other hand, they can be related with other MOF-based metamodels such as UML metamodel, which is used widely to create application models, and then the mapping between patterns and applications can be set up easily. Furthermore, the metamodels are middleware-aware. It means that every element defined in pattern metamodel must be perceivable by the middleware and any refactoring defined by the metamodel must be mapped to the relevant operations supported by the middleware. Details of the metamodels will be introduced in the following part.
- **M1 layer:** The bad/good pattern is abstracted from the bad/good structure of applications. Refactoring plan describes how to transform a bad pattern to the relevant good pattern and it is a sequence of some atomic refactoring operations, which are defined by the refactoring metamodel.
- **M0 layer:** The bad pattern instance denotes the ill-structure in the running application, while the good pattern instance denotes the refactored part in the application without the ill-structure. Refactoring is the instance of refactoring plan and it is the factual operation to change the running application.

2.2. Pattern metamodels

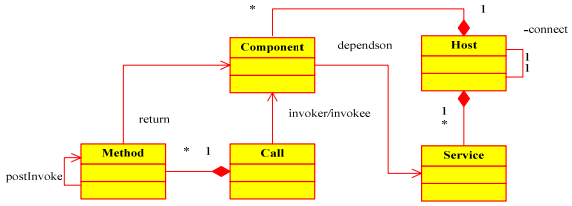

Figure 2. Pattern metamodel

Figure 2 presents an overview one of the pattern metamodel in our approach. The following is a brief description of the entities:

- *Host* defines a host in distributed environment. *connect* presents that there is a network path between two host instances.
- *Service* defines a service provided by middleware such as transaction and security. If a *Host* owns a *Service*, it means that a service instance is deployed on a host instance.
- *Component* defines an application component. If a *Host* owns a *Component*, it means that a component instance is deployed on a host instance. *dependson* presents that a component instance will use a service instance.

- *Call* defines a method call sequence between two components. *invoker/invokee* presents that a component instance is the invoker/invokee of a method call sequence instance.
- *Method* defines a method call between two components. If a *Call* owns a *Method*, it means that a method call instance belongs to a method call sequence instance. *postInvoke* presents that a method call instance occurs after another method call instance. *return* means that a method call instance will return a component instance.

There are also some other elements in the pattern metamodel. For every entity introduced above, it owns several attributes which denote the attributes belonging to the corresponding entity instance. In order to describe refactoring operations supported by the middleware, some middleware-specific entities and relations are imported to the metamodel. Considering the limitation of the paper length, these elements will not be introduced in detail here and will be illustrated in Section 3.

For ill-structures in a running application defined by the pattern model, we can use *Host* and *Service* to describe the running environment of the application, use *Component* to denote the application components' attributes and configurations, and use *Call* and *Method* to present interactions between components. Whatever, all of the information described by the metamodel can be collected by the middleware.

2.3. Refactoring metamodels

In order to transmit a bad pattern to its relevant good pattern, the refactoring plan is needed. Refactoring plan is a sequence of some atomic refactoring operations. Based on the pattern metamodel, a refactoring metamodel is defined. The following three kinds of atomic refactoring operations are provided in this metamodel: refactoring of entity, refactoring of entity's attribute, refactoring of relation between entities.

Definition 2.3.1 A refactoring of entity is a 3-tuple R_E = (E, ID, OP), where E is the type of refactoring entity, ID is the exclusive label of refactoring entity and OP denotes the operation, which is performed on the entity, including "Add" and "Delete".

Definition 2.3.2 A refactoring of attribute is a 4-tuple R_A = (E, ID, A, V), where E and ID respectively denote the entity's type and label that refactoring attribute belongs to, A is the name of refactoring attribute and V is the attribute's value after refactoring.

Definition 2.3.3 A refactoring of relation is a 6-tuple R_R = (E_1, ID_1, E_2, ID_2, R, OP), where E_1, ID_1, E_2 and ID_2 respectively denote type and label of the entities the relationship is associated, R is the name of refactoring relation and OP is the operation, "Add" or "Delete".

The refactoring metamodel is middleware-aware, that is, every atomic refactoring operation defined by the metamodel can be mapped to the actual operations supported by the middleware. Refactorings on *Host* or *Service* and the correlative relations are mapped to the middleware operations that change the running environment of application; refactorings on *Component* and the relations are mapped to lifecycle operations of components (such as deployment, undeployment, generation of a new component, etc) or reconfigurations of components; refactorings on *Call* or *Method* and the relations are mapped to modifications on the interaction between components, including to change the call sequence, to preprocess the parameters/return-values in method calls, etc. Because different middleware may have different abilities to change the running applications and the different implementation, the factual mappings are middleware-specific.

2.4. Process of the approach

The process of our approach can be divided into the following four phases:

1. **Define the patterns:** A bad pattern describes the ill-structure in an application and a good pattern describes the relevant part in the application after refactoring. Users can define the patterns using the pattern metamodel and the following phases can be completed by the middleware automatically.
2. **Detect the bad pattern:** At first, the runtime information of the application is obtained by the middleware. Then the information is analyzed to match the bad patterns defined by users. If a bad pattern is matched, its instance is discovered.
3. **Get the refactoring plan:** The bad pattern model and good pattern model only describe the source and target conditions of an application. Besides these user-defined patterns, a refactoring plan is needed to guide how to transform a bad pattern to the relevant good pattern.
4. **Perform refactoring:** The refactoring plan is instaniated as the factual operations supported by the middleware, which will change the running application without stopping it.

3. Illustrative Example

We have implemented a prototype of the approach based on a J2EE application server, PKUAS [6] [10]. In this section we will demonstrate how to perform

model refactoring at runtime in our approach based on an illustrative application.

3.1. The application for demo

In order to give a more vivid demonstration on how to detect the bad pattern and perform refactoring to remove it in our approach, we implement a sample application with some ill-structures. This application is a part of an e-bookstore and can be used to obtain the information of books and display it on web pages. It includes an entity bean, named BookEJB and some web-tier components, both of which are deployed to different machines.

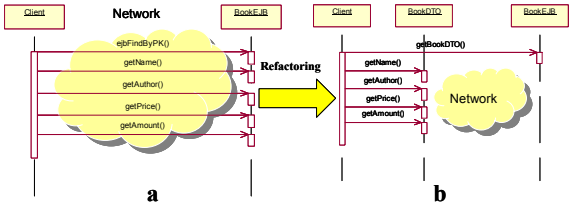

Figure 3. An ill-structure and its refactoring

Imagine the scenario when a client needs to get a set of attributes of a book, such as name, author, price, etc. As shown in Figure 3-a, the client gets the data it needs by executing multiple fine-grained calls to the server in the application. The problem with this approach is that each call to the server is a network call, requiring the serialization and deserialization of return values, consuming much time to transmit data through network. Executing multiple network calls in this fashion will contribute to significant degradation in performance and obviously this is an ill-structure in our sample application. This ill-structure is named Fine-grained Remote Calls [2].

As mentioned before, this application needs to be refactored and the classical EJB design pattern DTO (Data Transfer Object) [5] can be applied to this situation. A DTO is a plain serializable Java class that represents a snapshot of some server-side data, used to transport any kind of data between the tiers in a J2EE system. Using DTO in our sample application (as shown in Figure 3-b), the BookEJB would create a DTO, namely BookDTO, and endow it with the attributes that the client requires. This data would then be returned to the client in one bulk return value.

3.2. Define the pattern models

First of all, the models of bad pattern and its relevant good pattern(s) should be defined based on the pattern metamodel. Then, these models can serve as guidance for the following steps during refactoring. Figure 4 shows the bad pattern model in our case. This model shows a bad pattern like this: For a component whose type is Entity Bean, if it receives a call sequence that includes an ejbFindBy*() method invoked at first and followed by more than one get*() method, which all come from the same remote client, then this is a Fine-Grained Remote Calls bad pattern instance in the application.

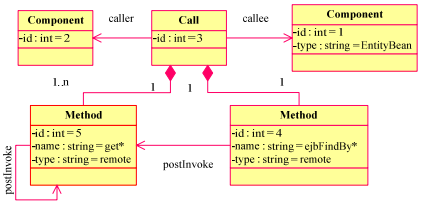

Figure 4. The bad pattern model

Bad pattern model can guide the process to detect the ill-structure in application and one or more relevant good pattern models should be defined for refactoring. Good pattern is the abstraction of the refactored part in application which has removed the ill-structure. The said bad pattern can be removed after DTO pattern is applied in the application by restructuring the design model and generating the code automatically or manually like the traditional model refactoring. Unfortunately, the code of application has been modified and the running system should always be "stopped-updated-restarted" to complete model refactoring in this way.

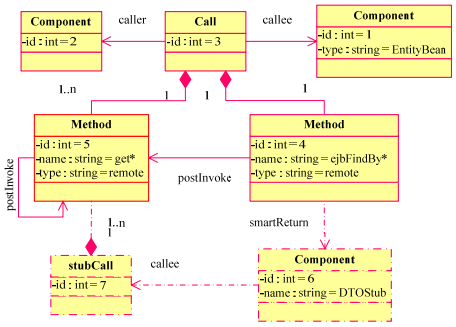

Figure 5. The good pattern model

In our approach, model refactoring can be performed without stopping the running system with the help of middleware. For the J2EE applications that keep operating in the middleware, the middleware can change the behavior of the applications in a certain extent by changing the runtime environment but not the applications. For example, when calling a method of an EJB, the client never calls the method directly because the client and the EJB are running in different JVM in distributed environment. In fact the client will call the method in the EJB's interface. The implementation of the interface on the client side is

called a stub. When receiving a request, the stub will delegate the request to the EJB in the remote server. The stub is in charge of communication through the network, lies in middleware layer and keeps invisible to the application. If we modify the stub, we can change the operation of the applications without modifications on them. To sum up, the refactorings can be completed by middleware.

Figure 5 displays the good pattern model in our case. There are two views in the model: the part with real lines is the application view, which describes the application only; the one with dashed lines is the middleware view, which displays the runtime environment, i.e. actual state of the applications in the middleware. There are two middleware-specific elements in this model. The new entity *stubCall* denotes which component the method calls will be delegated to by the stub. By default the requests from client will be redirected to the remote EJB. The association *smartReturn* means that the method ejbFindBy*() returns a stub that has been modified for some purposes. The method call with name get*() is delegated to the component DTOStub, which is returned by ejbFindBy*(), not to the remote EJB. Notice that from the application view, the model in Figure 5 has gone through no more changes and there are changes only in the middleware view. It means that the refactoring can be achieved without modifying the application and all the changes in the middleware remain transparent to the application.

3.3. Detect the bad pattern

Because the pattern model in our approach is perceptible for middleware, which means any element in the model, including the entity, attribute of entity, and relation between entities, can be monitored by middleware, the bad pattern instance can be discovered by middleware automatically as long as the bad pattern model has been defined.

In the example, detection of the bad pattern needs the following kinds of information: attribute of entity *Component*, including type of the component; attributes of entity *Method*, including name of the method, type of the method call (remote or local); relation between *Method* entities, including sequence of the method calls. The information can be achieved by middleware easily. Considering our implementation, PKUAS can get the information automatically at runtime without any modification on the application.

Once information has been collected by middleware, the following step is trying to match the collected information with the bad pattern model. To match the static information, such as the type of component, is effortless. We can loop through all the components in the application to find the ones that match the bad pattern model. But to match the runtime information, such as the sequence of method calls, is difficult, because the amount of this kind of information is huge. Fortunately, some data mining approaches, like [17], can help to solve this problem. For example, in our case we can get the result that for the BookEJB, 80% of the call sequences are "ejbFindByPrimaryKey(), getName(), getPrice()" from the runtime information, then we can detect this as a bad pattern instance and try to do some refactorings. The information is also useful during refactoring because it can be decided that the DTOStub should include the attributes "name" and "price" of BookEJB.

3.4. Get the refactoring plan

Through comparing the bad pattern model and good pattern model, the atomic refactoring operations in refactoring plan can be found easily. In this case, based on the patterns presented in Figure 4 and Figure 5, the set of atomic refactoring operations are discovered as follows:

1) R_E(Component, 6, ADD);
2) R_A(Component, 6, name, DTOStub);
3) R_R(Component, 6, StubCall, 7, callee, Add);
4) R_R(Method, 4, Component, 6, smartReturn, Add);
5) R_R(Method, 5, StubCall, 7, composite, Add);
6) R_E(StubCall, 7, ADD).

Considering the potential dependency between atomic refactoring operations, it is necessary to order these operations because to perform refactoring operations in an incorrect order may result in the failure of refactoring, even the crash of running system. A set of rules are presented in our approach to judge the dependency between any two atomic refactoring operations and based on these rules these operations can be arranged to a correct order. In this case, we can get the following orders based on some rules:

- **Rule:** To add an entity should be preferential than to modify its attribute.
  Result: 1=>2;
- **Rule:** To add an entity should take priority over to add the relation associated with this entity.
  Result: 1=>4, 6=>5, 1=>3, 6=>3.

Then the correct sequence of atomic refactoring operations can be achieved based on above-mentioned orders. Obviously the correct sequence is possibly not exclusive because the operations which are independent can be performed in any order. Finally, the refactoring plan of this example is: 1=>2=>6=>5=>4=>3.

3.5. Perform refactoring at runtime

The last phase of our approach is to perform factual refactoring on running application with the help of middleware at runtime.

Refactoring plan describes the refactoring of pattern models and it cannot be fulfilled directly on running application. At the detection phase, the bad pattern model instance has been discovered and the association between the pattern model and the running application has been set up. As the result, the refactoring plan can be instaniated to associate with the instances in running application. As for our case, the refactoring is performed on the BookEJB and its call sequence "ejbFindByPrimaryKey(), getName(), getPrice()".

Refactoring is completed by middleware in actuality at last, so the instaniated atomic refactoring operation should be mapped to the operations supported by middleware. This mapping of the two kinds of operation is not one-to-one and several atomic refactoring operations may be mapped to only one middleware's operation. The refactoring operations are mapped to two practical operations of middleware in our sample:

1. **Generate the DTOStub:** Refactoring operation 1 and 2 are mapped to this one. Compared with the normal stub class, DTOStub adds a few attributes in the class and modifies the mapping methods named get*(), which is responsible for getting the attributes. For the normal stub, the method get*() will redirect the request to the remote EJB. These methods in DTOStub are modified to directly return the attribute values in the DTOStub instance. Which attributes are added and which methods are modified in the DTOStub are decided by which attributes are got by the client at runtime. The information can be induced from what's collected at runtime and then the DTOStub will preload only the information that the client needs. This time the attributes name and price are added. After being generated, the DTOStub should be compiled and loaded by the classloader of BookEJB, which contains all the classes the EJB needs, and will be distributed to the client side through the standard mechanism of passing a remote object reference from the server to the client.
2. **Add the DTO-Pattern-Controller:** This operation is mapped from 3, 4, 5 and 6 of the refactoring plan. In the EJB container of PKUAS, we can add and remove pattern controllers dynamically. The pattern controllers are implemented to add additional functions or change the behavior of the container. DTO-Pattern-Controller is in charge of instantiating a DTOStub to replace the old stub. This controller is expected to add the DTO pattern to the applications transparently. The controller can also be easily implemented as application independent and then could be reused in other applications suffering the same ill-structure.

After these two operations are executed, refactoring is completed and the ill-structure "Fine-grained Remote Calls" in the sample application is removed. Moreover, if the refactoring is invalid or incorrect, it can be easily rolled back automatically, that is, performing the previous changes in reverse. In the sample, just to disable the DTO-Pattern-Controller is enough. Obviously, the rolling back of refactoring is also transparent to the application and doesn't interrupt the operations of the application.

3.6. Evaluate the result of refactoring

In our approach, middleware-supporting refactoring is processed to remove the ill-structures in application at runtime. This manner is different from the traditional one that performs refactoring on the source code of the application, but will in fact lead to similar results. In this section we will evaluate the runtime refactoring based on the sample application. The details of runtime environment are described as follows: the BookEJB is allocated on a node with Pentium 4 CPU 2.8G and 512M memory; the web-tier components are allocated on a node with Pentium 4 CPU 2.4G and 256M memory; the bandwidth between the two nodes is 100M.

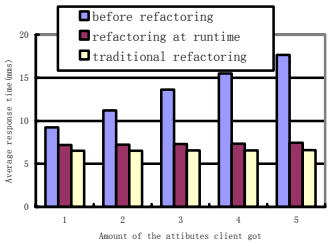

Figure 6. Average response time before/after refactoring

Figure 6 gives the average response time of the scenarios that the client needs to get a set of attributes of a BookEJB in three states of the sample application: with the ill-structure, refactoring at runtime and with traditional refactoring. The quantity of the attributes the client gets, namely the quantity of the get*() method calls, is an important factor that effects the response time and we have got the average response time for different quantities, from 1 to 5.

From the result shown in Figure 6, we can see that runtime refactoring can lead to less response time than

no refactoring. In the state of non-refactoring, response time will have an increase of 2 ms if the client tries to get one more attribute from the server side. The reason is that one remote call will occur when getting one attribute. Considering the state of runtime refactoring, we can discover that the average response time almost keeps unchanged when getting more attributes and it is because that only one remote call occurs when the client calls the method ejbFindBy*() in spite of the number of attribute values retrieved.

With an eye to the average response time in the state with traditional refactoring, it is less than the one with runtime refactoring. It is because that the operations of middleware in runtime refactoring are not application specific, while traditional refactoring will modify the code of application, so it can bring more improvement. However the gap is virtually very narrow. Compared with the response time with traditional refactoring, the call sequence will consume 0.7 more ms (about 9%) with runtime refactoring. More importantly, the advantage of runtime refactoring is that it can process during the runtime without interrupting the operation of applications.

4. Discussion

Some essentials should be put forward. Although the pattern models need to be defined by users manually, it is not difficult to complete this step. There are many researches to introduce the potential ill-structures and the corresponding solutions, such as antipattern [20] and design pattern [4]. They are much helpful for us to define bad/good pattern models in our approach. On the other hand, patterns are usually common to many applications and then predefined bad/good patterns are reusable. As a result, users only need to select the appropriate patterns when refactoring.

Refactoring is used to improve the quality of software, including extensibility, reusability, complexity, performance, reliability, etc. For our approach, because it focuses on the runtime and doesn't modify the application code, it is effective for the quality concerned at runtime, such as performance and reliability.

In order to validate the applicability of our approach, we have tried to define 38 pairs of bad pattern and good pattern which are derived from classical ill-structures and relevant solutions, which have impacts on the performance and are introduced in [1] [2] [4] [5] [18] [20].

Figure 7 shows the result of the applicability of the approach. 35 bad patterns (92%) are defined by the pattern metamodel successfully and can be detected in our approach. The rest 3 ill-structures (8%) result from the business logic within the component and they can't be discovered without the code in fact. Considering refactoring, 23 good patterns (60%) are defined successfully and can be applied to application at runtime. Within them 9 (24%) can be applied automatically. 14 (36%) will need a little human intervention to generate some assistant classes, such as the DTOStub in the case this paper introduces. Fortunately, some assistant classes can be generated by middleware, such as the DTOStub, and these good patterns can be applied automatically in fact. To apply the good patterns needs to modify the attributes of component or the interaction between components. The other 15 ones (40%) need modification on the parts within the component, and can only be completed after rewriting the program.

To draw a conclusion, our approach is practicable for the ill-structures which do not result from the business logic and for the solutions which need no modifications inside the component.

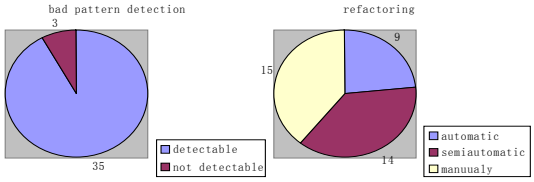

Figure 7. Applicability of the approach

There are some open issues in our approach. Now, the effectiveness of refactoring is guaranteed by users who define the patterns. In fact, it can be verified or evaluated before refactoring. There are some researches to solve this problem [3] [14]. We can make use of them to improve our approach in the future.

Adaptability of middleware is very important to the final effectiveness of model refactoring. For example, when replacing a component, the state may or may not be taken into account, either way of which may lead to different impacts on the running application. More details about middleware should be taken into account during refactoring. Some constraints should be added to the pattern and refactoring metamodels to describe the adaptability of middleware.

5. Related Work

There are many researches on refactoring in past years and most of them focus on the code level, in which the code is altered directly [15] [19]. Similar to our approach, design pattern can be applied to the legacy code in the work introduced in [11]. Furthermore, J. Jahnke proposes an approach to detect poor design pattern in OO program and rewrite it by good design pattern [9]. Different from them, our

approach is focus on a higher level: refactoring of model. Although the code will be altered indirectly finally in our approach, users only need to provide the good/bad pattern model and the following work is completed by middleware automatically.

Model refactoring is a recent trend of refactoring and existing researches always focus on the problems of how to transmitting the design models correctly and expediently [7] [8]. There are some researches similar to ours. R. France et al. present an approach to evolving the design models with the guide of pattern model defined by relevant specification, which is an extension of UML [13]. To apply these works to existing applications, there will be some limitations. For example, application must stop and restart to validate the refactoring on model. Compared with this work, the distinct feature of our approach is that model refactoring is applied at runtime. In our approach, refactoring on model is instaniated to the operations supported by middleware, which can modify the context of application. As the result, the running application need not stop while refactoring.

Some researches are closer to our approach and are applied to detect ill-structures in running application with the help of middleware. T. Parsons et al. proposes a framework for detecting performance antipatterns [16]. The approach makes use of statistical analysis and techniques from the field of data mining to summarize the collected performance data. Performance antipatterns are detected from the summarized data using a rule-engine approach. Our work is similar to this framework, but goes further. In our approach, developers can get more assistance when processing refactoring by the middleware.

6. Conclusion

In this paper a middleware-based approach is proposed to applying model refactoring at runtime. In this approach the ill-structure in running application and the relevant refactored application part are abstracted respectively to bad pattern and good pattern. A metamodel architecture is introduced into this approach and users can define the good/bad pattern model using the pattern metamodel. The metamodel is perceptible for middleware and then middleware can discover bad pattern instance and perform refactoring automatically. Finally, refactoring is completed at runtime by the middleware without any modification on the application. Therefore, the runtime refactoring will not stop the running application.

Acknowledgements

This work has been supported by the National Grand Fundamental Research 973 Program of China under Grant No.2005CB321805, the National Natural Science Foundation of China under Grant No. 90412011, 90612011 and 60403030.

References

[1] B. Dudney, S. Asbury, J. K. Krozak, K. Wittkopf, *J2EE Antipatterns,* Wiley Press, 2003.

[2] B. Tate, M. Clark, B. Lee, P. Linskey, *Bitter EJB,* Manning Publications, 2003.

[3] D. Sands, *Total Correctness by Local Improvement in the Transformation of Functional Programs*, Trans. Programming Languages and Systems, 1996, vol.18, no.2, pp. 175-234.

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

[5] Floyd Marinescu, *EJB Design Patterns: Advanced Patterns, Processes, and Idioms,* John Wiley and Sons, Inc., 2002.

[6] G. Huang, H. Mei, F. Q. Yang, *Runtime Recovery and Manipulation of Software Architecture of Component-based Systems,* International Journal of Automated Software Engineering, Springer, 2006, Vol. 13, No. 2, 251-278.

[7] J. Xu, W. Yu, K. Rui, G. Butler, *Use Case Refactoring: A Tool and a Case Study*, APSEC 2002, pp. 484-491.

[8] J. Zhang, Y. Lin, J. Gray, *Generic and Domain-Specific Model Refactoring using a Model Transformation Engine*, Model-Driven Software Development, Research and Practice in Software Engineering, 2004.

[9] J.H. Jahnke, A. Zündorf, *Rewriting Poor Design Patterns by Good Design Patterns*, Proc. ESEC/FSE '97 Workshop Object-Oriented Reengineering, 1997.

[10] Mei, H. and G. Huang, *PKUAS: An Architecture-based Reflective Component Operating Platform*, invited paper, 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, Suzhou, China, 2004, 26-28.

[11] Mel Ó Cinnéide, P. Nixon, *Automated Application of Design Patterns to Legacy Code*, Proceedings of the Workshop on Object-Oriented Technology, 1999, pp. 176-180.

[12] OMG, *Meta Object Facility (MOF) Specification Version 2.0,* 2003.

[13] R. France, S. Ghosh, E. Song, D. K. Kim, *A Metamodeling Approach to Pattern-based Model Refactoring*, IEEE Software, 2003, pp. 52-58.

[14] S. Demeyer, *Maintainability versus Performance: What's the Effect of Introducing Polymorphism*, technical report, Lab. On Reeng., Universiteit Antwerpen, Belgium, 2002.

[15] S. U. Jeon, J. S. Lee, D. H. Bae, *An Automated Refactoring Approach to Design Pattern-Based Program Transformations in Java Programs*, APSEC 2002, pp. 337-345.

[16] T. Parsons, *A Framework for Detecting Performance Design and Deployment Antipatterns in Component Based Enterprise Systems,* 2nd International Middleware Doctoral Symposium, 2005.

[17] T. Parsons, J. Murphy, *Data Mining for Performance Antipatterns in Component Based System Using Run-Time and Static Analysis*, Transaction on Automatic Control and Computer Science, Vol. 49(63), 2004.

[18] Tom Mens, *A Survey of Software Refactoring,* IEEE Transactions on Software Engineering, Vol. 30, No. 2, Feb. 2004, pp. 126-139.

[19] V. Jamwal, S. Iyer: *Automated Refactoring of Objects for Application Partitioning*, APSEC 2005, pp. 671-678.

[20] W. J. Brown, R. C. Malveau, H. W. McCormicm, T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis,* John Wiley and Sons, Inc., New York, 1998.