Exploring Dynamic Compilation Facility in Java

Dingwei He and Kasi Periyasamy
Computer Science Department
University of Wisconsin-La Crosse
La Crosse, WI 54601
kasi@cs.uwlax.edu

Abstract

Traditional programming practice requires re-compilation of the entire code whenever it is modified. Such re-compilation may take more time if the size of the resulting code after modification is too big. This will be problematic for applications that are frequently changed. Component Based Software Engineering (CBSE) provides one option to overcome this problem by implementing several components separately and then coherently composing them together to form an application. In this case if properly designed, modification to one component may not significantly affect other components in the system but re-compilation of the modified component is still necessary. The latest Java technology provides another option in which a new piece of code can be compiled and then dynamically loaded, along with the existing code, without re-compiling the latter.

This paper describes the dynamic compilation facility (also referred to as "on-the-fly" compilation in this paper) in Java and demonstrates its application using a simple case study on a banking environment. The case study illustrates how it is possible to add a new service to the banking environment by adding the code for the new service, compiling and loading it along with existing services provided by the bank. For example, in addition to the normal transactions such as 'deposit', 'withdraw' and 'check balance', a manager will be able to add a new service called 'special withdraw' that will be used by privileged customers. A critical comparison of this approach to CBSE with regard to the case study is also addressed in the paper.

The major advantage of this approach is not only that it eliminates the need for re-compilation of the whole application but it greatly supports incremental development of large applications. The authors are using this technique in developing a GUI-based front-end for an object-oriented database. In this case, a new object can be created using the GUI and the code for this object is generated, compiled and added to the application. The paper briefly discusses the comparison of "on-the-fly" compilation with other dynamic compilation techniques. This comparison distinguishes the use of dynamic compilation facility at the application level and continuing work in this direction.

1 Introduction

One of the major tasks of software maintenance is *enhancement* by which new features are added to existing ones. Traditionally, when a new feature is added, the entire code must be recompiled and the application is deployed again. Typically such enhancements are done in newer releases of the software and are infrequent. However, there are some types of applications where the frequency of enhancements or modifications is so high that re-compilation and re-deployment consume so much time and significant affect the usage of the product. Examples of such software systems include bank transactions where policies are changed quite frequently and/or often new policies are also introduced. Another example is the set of applications loaded on a mobile phone. With increasing demand for mobile applications and the fact that changes occur so frequently, re-compilation and re-deployment are real bottlenecks.
Component Based Software Engineering (CBSE) [3, 5] provides one possible solution for re-compilation problem. In this case, the software system includes several components that are somewhat self-contained packages. These components are cohesively composed together to form an application. When a new feature is expected to be added to an existing system, it is designed as another component and integrated with the set of existing features. When using this approach, the designer has an obligation to carefully identify the components and to ensure that the dependencies among the components are less. In addition, the approach still requires re-compilation of the new component and the set of existing components with which the new component interacts, if not re-compilation of the entire code.
In this paper, the authors look at another possible approach called "on-the-fly" compilation provided by the current Java technology. This approach supports dynamic compilation and dynamic loading of new code without the need for re-compilation of existing code. By dynamic, it means that a programmer can write a piece of new code which can be read, compiled and loaded by the same application while the application itself is running. Java supports this facility in version 1.6 onwards. The programmer can write the code in such a way that an application accepts a Java program through an external file, invoke a Java compiler and loader, and add the byte code that is compiled will be added to the same application. Java's previous work in this direction resulted in Just-In-Time (JIT) compilation facility [1]. However, JIT technique was mostly used by compiler writers and was not visible for application programmers. Unnikrishnan and others [6] discussed more on reducing compile time overhead of Java programs but mostly focused on system programming. Masuhara and others [2] have explored the dynamic compilation facility much similar to the "on-the-fly"' compilation. They focus on developing and using a reflective language. The major problem in their approach is that there is no architecture to realize or to implement the approach [2]. A follow-up work by Radhakrishnan and others [4] describes how Java run-time charactersitics can be exploited in terms of dynamic loading of byte cods much similar to the JIT technique. The authors in this paper claim that the "on-the-fly"' compilation technique is different from JIT mainly because the former is application-oriented whereas JIT is system-oriented. Consequently, JIT technique is more useful for Java programmers who wish to focus on run-time performance issues.
The "on-the-fly"' compilation approach is illustrated with a case study on a bank transac-

tions system in which a dynamic service can be added to the set of existing services provided by the bank. The major advantage of this approach is that it not only eliminates the need for re-compilation but it also supports incremental software development approach. For example, one can use the CBSE design technique to identify the components at design time but add the code for each component one at a time.The authors are using this technique in developing a GUI-based front-end for an object-oriented database. In this case, a new object can be created using the GUI and the code for this object is generated, compiled and added to the application.

2 Methodology

Consider a data-oriented application such as a banking system that provides a set of transaction services to its customers. If a new transaction is required to be added, first additional code must be written for this transaction and is saved in a separate file. This file is then compiled by the banking system and loaded with the banking system itself. In order to do this, the programming language should support dynamic compilation of this new code and running the code. Java version 1.6.0 provides all these services. Following is a simple Java code by which one can call the system Java compiler from within a Java program.

```
public void invokeCompiler (String filename) {
JavaCompiler compiler =
   ToolProvider.getSystemJavaCompiler();
if (compiler != null) {
       int result =
          compiler.run (null, null, null, filename);
  if (result == 0)
      System.out.println (" Compilation successful");
      else
          System.out.println (" Compilation failed");
}
else System.out.println
     (" Unable to get system java compiler");
```

The "filename" in the above example refers to the name of the file containing the newly written source code. The above code uses the 'SystemJavaCompiler' which is usually available somewhere in the Java directory. To make this program work, one needs to include the path of the Java Compiler (namely 'javac') in the CLASSPATH environment variable so that the run-time system, when executing this code, will be able to find the compiler. Similar to dynamic compilation, a pre-compiled Java program can also be run dynamically from within another Java program. This is done by invoking the Java run-time system, very similar to invoking the Java compiler in the previous example. The following piece of code shows how a compiled program can be run by loading the program first and then using it.

```
    Class SpecialTransactionClass = null;
    try {
        URLClassLoader cl =
           URLClassLoader.newInstance
               (new URL[]{new URL("file:///c:/dir")});
        SpecialTransactionClass  =
           cl.loadClass("SpecialTransaction ");
        }
    catch (Exception ex)
        { System.out.println(ex);}
```

In the above code, “dir” refers to the folder in which the user puts the dynamically compiled classes, and “SpecialTransaction.class” is assumed to be the newly compiled code which is put in “c:/dir/transactions/”. This new class can be used in two ways: (1) using the reflection API in Java, and (2) using an organized class path. The following code shows the usage in both ways. Assume that the “SpecialTransaction” class has only one method namely,“public void execute(Object arg)”.

Using the reflection API

```
    try {
       Method  executeMethod =
           SpecialTransactionClass.getMethod("execute", Object.class);
       executeMethod.invoke
           (SpecialTransactionClass.newInstance(), "StringParameter");
        }
    catch (Exception ex)
       { System.out.println(ex);}
```

The above code indicates that the method “execute” is selected using the reflection API call “getMethod” and then is invoked using the parameters supplied in “StringParameter”.

Using organized class path
Assume that we have a program called “DynamicATM” and all extended functionalities are expected to be stored in a directory called “d:/DynamicATMExtensions”. Assuming that the file “SpecialTransaction.class” is placed in the directory “DynamicATMExtensions/transactions/”, the following command (on Windows Operating System) is issued to start the application:

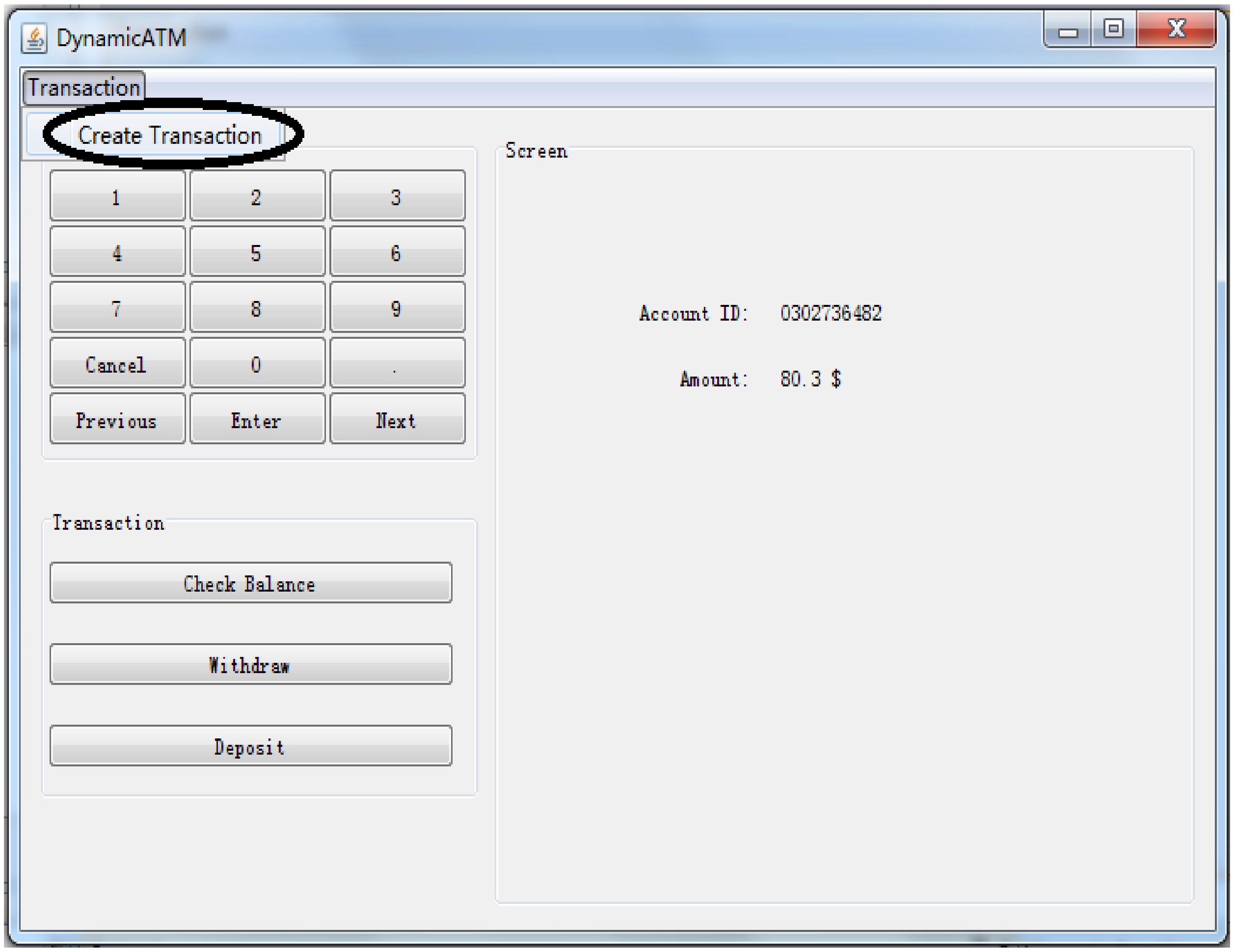
```
java -classpath %CLASSPATH%;d:\DynamicATMExtendsions -jar DynamicATM.jar
```

Now, the dynamically created class can be used as shown below:

```
transactions.SpecialTransaction st =
     new transactions.SpecialTransaction();
st.execute("StringParameter");
```

3 Case Study

The methodology described in the previous section is illustrated through a bank transaction system. The bank normally offers three types of transactions for a customer - deposit, withdraw and check balance. Assume that a customer cannot withdraw more than the available balance in an account. Figure1 shows the initial screen shot of the transaction system. Suppose that the bank manager would like to add a new service called *Special transaction* which lets a customer withdraw more than the available balance in an account thus leading to a negative balance in the account. To do this, the manager first clicks the menu item ‘Transaction’ on the screen (shown in Figure 1). This will open up the transactions menu which currently has only one item named ‘Create Transaction’. When the user selects this item, it leads to a different screen (shown in Figure 2). The manager

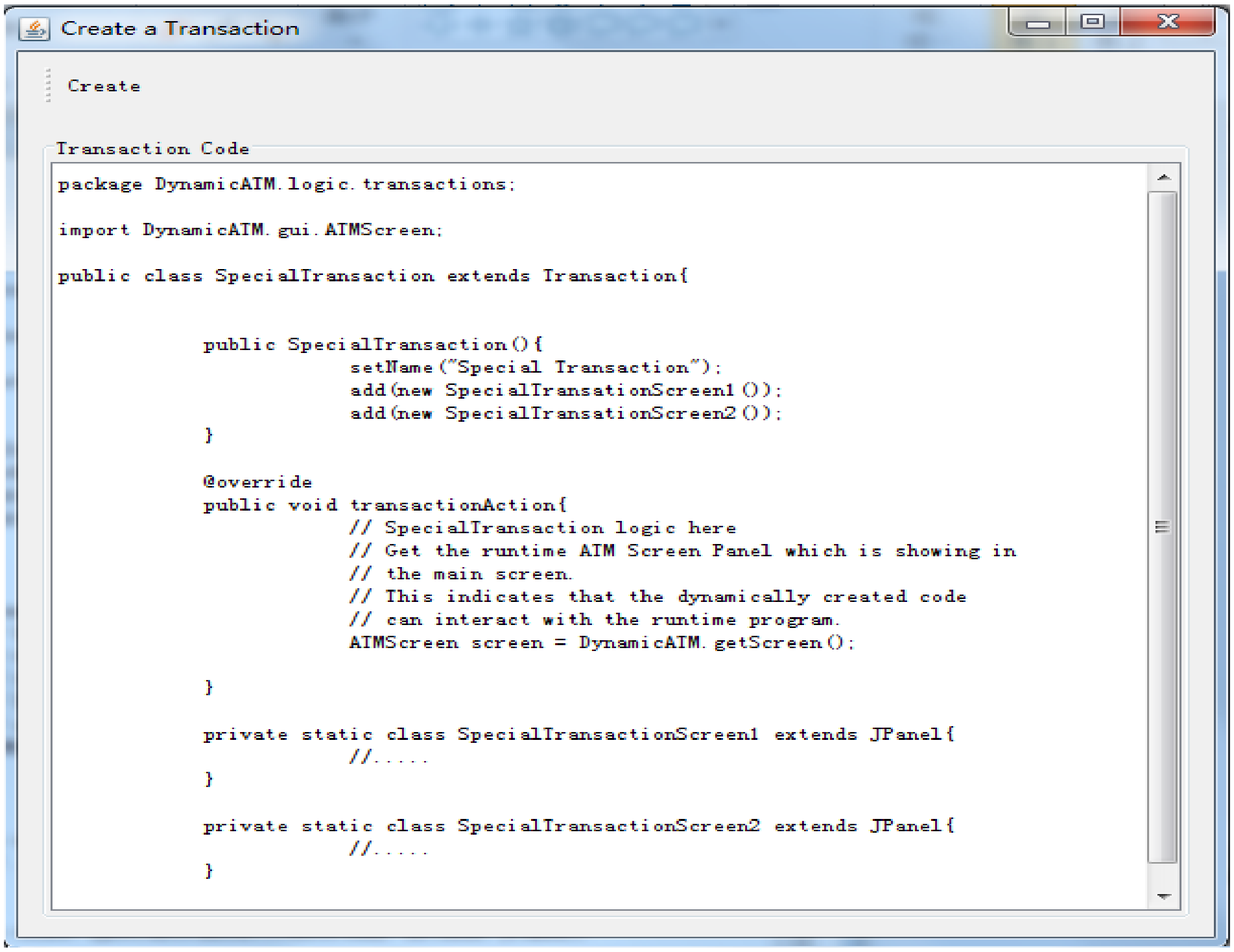


Account ID: 0302736482

Amount: 80.3 $

Figure 1: Main Screen of the Bank Transactions System

then adds the code for the new transaction as shown in Figure 2 and clicks the ‘create’ menu item appearing on the top left corner. The tool internally compiles and loads the new code. The new transaction is added to the existing three transactions as shown in Figure 3. It can be used as any other transaction thus enabling dynamic addition of services.

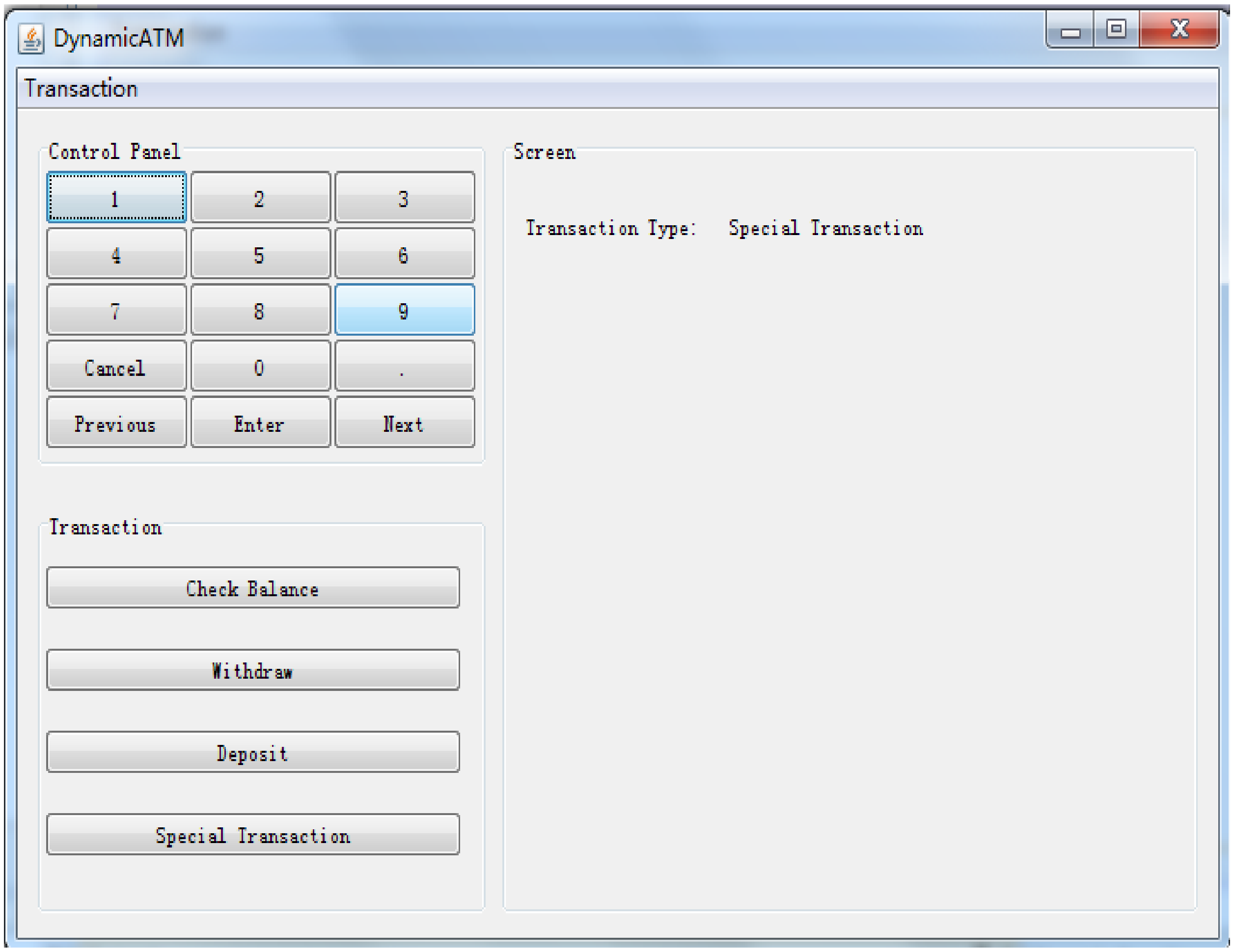


```
package DynamicATM.logic.transactions;

import DynamicATM.gui.ATMScreen;

public class SpecialTransaction extends Transaction{


            public SpecialTransaction(){
                        setName("Special Transaction");
                        add(new SpecialTransationScreen1());
                        add(new SpecialTransationScreen2());
            }

            @override
            public void transactionAction{
                        // SpecialTransaction logic here
                        // Get the runtime ATM Screen Panel which is showing in
                        // the main screen.
                        // This indicates that the dynamically created code
                        // can interact with the runtime program.
                        ATMScreen screen = DynamicATM.getScreen();

            }

            private static class SpecialTransactionScreen1 extends JPanel{
                        //.....
            }

            private static class SpecialTransactionScreen2 extends JPanel{
                        //.....
            }
```

Figure 2: Code for creating a new transaction

4 The Front-end of a Database

The authors are developing an object-oriented database which can be used by a Java application. Users of this database can store and retrieve Java objects into/from this database. The major advantage of this database is to hide the tabular implementation of a relational database and provide a seamless integration of the object-oriented technology from the graphical user interface to the database through the application layer. Like most of the commercial databases, this object-oriented database also comes with a GUI to manage the database directly without accessing it through a program. While most end users using commercial, relational databased such as Access and SQL server know how to create and maintain tables, it is not possible to expect end users to know objects (as used in OO approach). The GUI design for this OO database thus posed a challenge for the authors. It is in this context, the authors decided to provide a simple GUI for the end users to input the structure of the object which they want to store. The back-end code will then generate the class and its associated methods for this object, dynamically compile and load it with the application so that both end users as well as any application program that uses the database can access this newly created object. Figure 4 shows the *type editor* for this database. Using



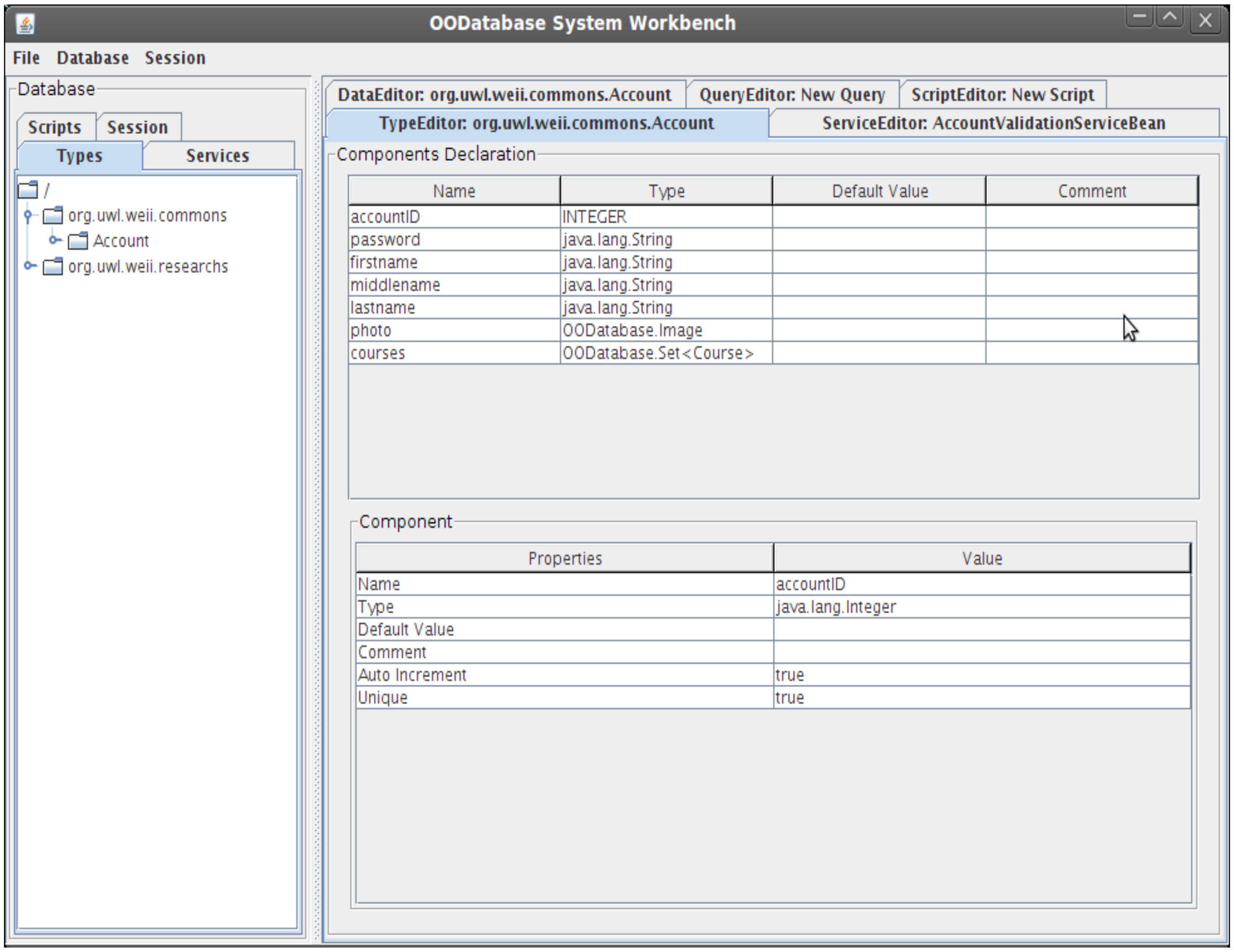
Transaction Type: Special Transaction

Figure 3: Updated Screen for the Bank Transaction System

the *type editor*, a user of the database can first create an object. Notice that the user does not need to know anything about OO programming. Instead, the user is expected to define the composite structure of an entity. Upon completion of this structure, the OO database will create a class for this structure with *get* and *set* methods. This class will be compiled and loaded with the application and at the same time an instance of this class will be stored in the database. For OO programmers who wish to access this newly created object, the GUI provides another editor called *service editor* which can be used to add a new piece of code on the fly. Figure 5 shows the *service editor* in which a Java programmer adds new code to access the newly created object. This way, the OO database can be used by both programmers as well as by end-users, the latter including non-programmers.

5 Limitations

As seen from the current implementation of the case study, the end user must know Java programming in order to add new code to the system. This does not seem to be a major problem because new code is necessary to enhance the existing features, no matter whether



Name	Type	Default Value	Comment
accountID	INTEGER		
password	java.lang.String		
firstname	java.lang.String		
middlename	java.lang.String		
lastname	java.lang.String		
photo	OODatabase.Image		
courses	OODatabase.Set<Course>		

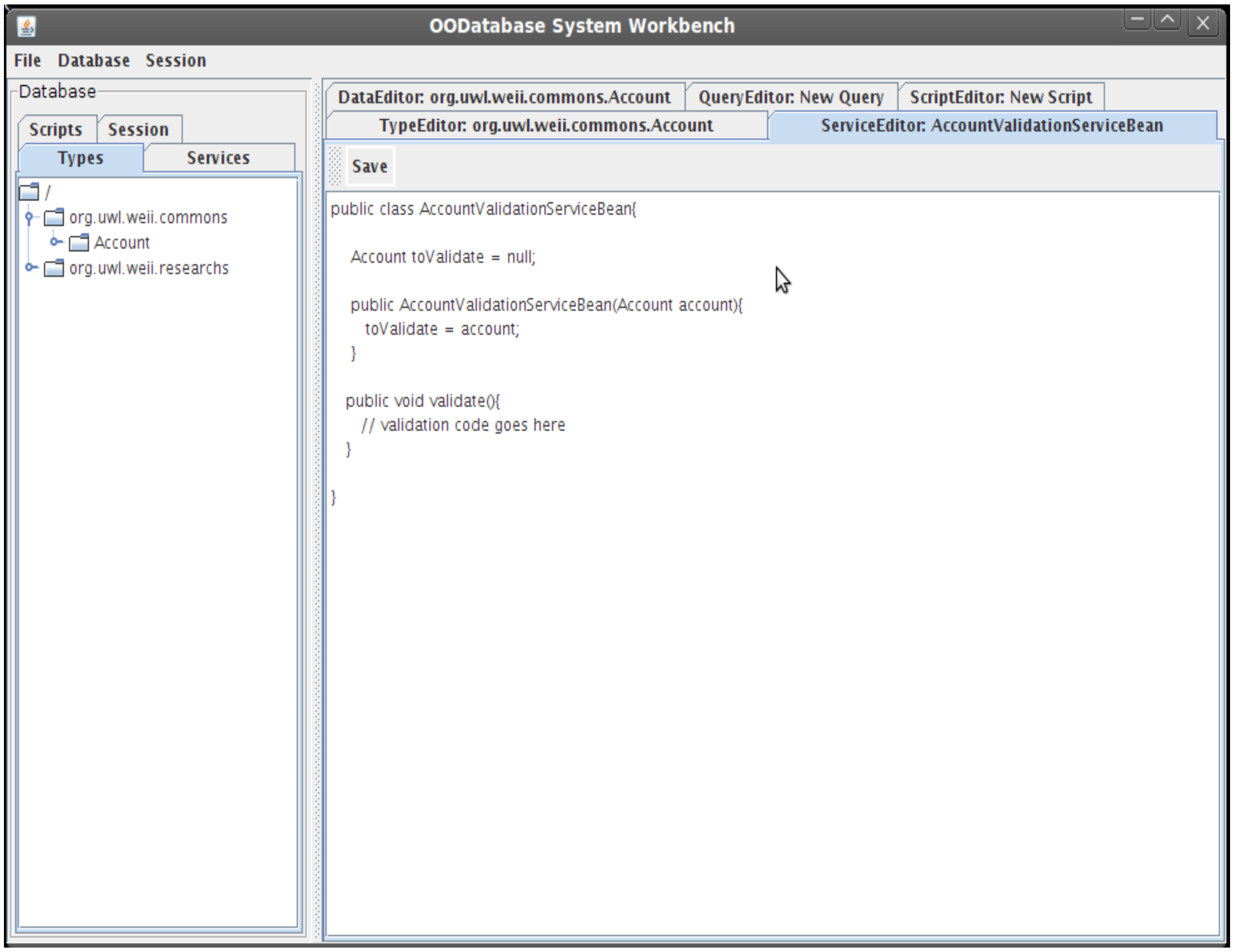
Properties	Value
Name	accountID
Type	java.lang.Integer
Default Value	
Comment	
Auto Increment	true
Unique	true

Figure 4: Type Editor for the OO Database

it is written by an end user or by the developer. However, when an end user has the flexibility of adding a new code by himself/herself, security becomes a major concern. In case of the banking example, strict access policies must be implemented in order to protect the code.

Second, the new code can only be used independently on its own; no existing service (such as deposit or check balance) can use this service because existing code does not have any knowledge of the new code. This problem is analogous to the relationship between a super class and a subclass in an inheritance hierarchy. The super class does not need to know its subclasses whereas the subclass inherits and uses the features of the super class.

Third, the impact of adding new service is not fully explored. For example, traditional software engineering process conducts regression testing whenever new code is added or existing code is modified. However, there is no such testing involved in this process. Therefore, the impact of the new code needs to be further explored.



```
public class AccountValidationServiceBean{

   Account toValidate = null;

   public AccountValidationServiceBean(Account account){
     toValidate = account;
   }

  public void validate(){
    // validation code goes here
  }

}
```

Figure 5: Service Editor for the OO Database

6 Conclusion

This paper describes how the dynamic compilation facility in Java can be utilized to add code to an existing application without re-compilation. This technique, referred to as "on-the-fly" compilation in this paper, is useful for applications that encounter changes in implementation. The paper illustrates the technique with a case study on a simple banking system and also explains how the authors have used this approach in the implementation of an OO database. The impact of this new technique on testing has not yet been identified and the authors continue to work in this direction.

References

[1] Aycock, J., "A Brief History of Just-In-Time", *ACM Computing Surveys*, 35:2, June 2003, pp. 97-113.

[2] Masuhara, H. *et al.*, "Dynamic Compilation of a Reflective Language Using Run-Time Specialization", *International Conference on Principles of Software Evolution*, Nov 2000, pp. 128-137.

[3] Heineman, G.T. and Council, W.T., *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley Professional, Reading 2001.

[4] Radhakrishnan, R. *et al.*, "Java Run-Time Systems: Characterization and Architectural Implications", *IEEE Transactions on Computers*, 50:2, Feb 2001, pp. 131-146.

[5] Szyperski, C., *Component Sofwtare: Beyond Object-Oriented Programming*, Second Edition, Addison-Wesley Professional, Boston 2002.

[6] Unnikrishnan, P. *et al.*, "Reducing Dynamic Compilation Overhead by Overlapping Compilation and Execution", *Asia and South Pacific Conference on Design Automation*, Jan 2006, pp. 929-934.