Identifying Data Transfer Objects in EJB Applications

Alexandar Pantaleev
Ohio State University
pantalee@cse.ohio-state.edu

Atanas Rountev
Ohio State University
rountev@cse.ohio-state.edu

Abstract

Data Transfer Object (DTO) is a design pattern that is commonly used in Enterprise Java applications. Identification of DTOs has a range of uses for program comprehension, optimization, and evolution. We propose a dynamic analysis for identifying DTOs in Enterprise Java applications. The analysis tracks the reads and writes of object fields, and maintains information about the application tier that initiates the field access. The lifecycle of a DTO is represented by a finite state automaton that captures the relevant run-time events and the location of the code that triggers these events. We implemented the proposed approach using the JVMTI infrastructure in Java 6, and performed a study on a real-world Enterprise Java application which is deployed for commercial use. Our results indicate that the dynamic analysis achieves high precision and has acceptable overhead.

1 Introduction

Data Transfer Object (DTO) [6, 1, 25] is a design pattern that is commonly used in distributed systems in general and Enterprise Java applications in particular. Every method call made to a business object in an enterprise system is potentially remote. In older Enterprise JavaBeans (EJB) applications such remote invocations use the network layer regardless of the proximity of the client to the server, creating network overhead. Such method calls may permeate the network layers of the system even if the client and the enterprise application layer are both running in the same Java Virtual Machine (JVM). When multiple attribute values need to be obtained, using multiple calls to `getX` methods (one per attribute) is highly inefficient.

A DTO, also called a transfer object or a value object,[1] encapsulates a set of values, allowing remote clients to request and receive the entire value set with a single remote call. In EJB applications, in most cases a DTO is an exact replica of the state of an *entity bean*; such beans serve as the liaisons between the application layer and the database.

The goal of our work is to define a dynamic program analysis that identifies classes which implement the DTO pattern. The identification of DTOs is useful in several contexts. First, it can assist *program comprehension* by pointing out instances of this Enterprise Java pattern; this can also be used to create additional documentation (e.g., JavaDoc comments). Second, DTOs may involve serialization, which can create performance bottlenecks [20, 17]. As a *performance optimization*, identified DTOs can be subjected to customized serialization mechanisms (i.e., type-aware serialization and acyclic-graph serialization [27, 22]). Finally, identifying DTO instances is an important step towards *software evolution* for migrating Java Enterprise applications based on the older EJB 2 specifications to the new EJB 3 model. In EJB 3, entity beans can be detached from the persistence context related to a database, modified elsewhere in the application, and merged back to the respective persistence context. As DTOs in older EJB applications usually mirror the state of entity beans, it is highly desirable to simplify the design by using the same entity bean object to represent state throughout the application layer stack.

Contributions. Our work has two specific contributions:

- *Dynamic analysis.* We propose a dynamic analysis for identifying DTOs in Enterprise Java applications. The analysis tracks the reads and writes of object fields, and maintains information about the application tier that initiates the field access. The lifecycle of a DTO is represented by a finite state automaton (FSA) that captures the relevant run-time events and the location of the code that triggers these events.
- *Experimental study.* We implemented the proposed approach using the JVMTI infrastructure in Java 6, and performed a study on a real-world Enterprise Java application — the J2EE Certificate Authority, which is deployed for commercial use on a number of web sites. Our results indicate that the dynamic analysis achieves high precision and has acceptable overhead.

[1]Note that a DTO is different from the GoF Value Object pattern [11].

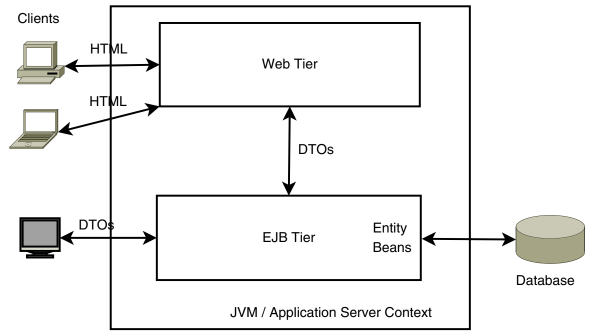

Figure 1. Structure of a JEE application.

2 Background and Problem Statement

A Java Enterprise Edition (JEE) application consists of layers, or tiers. The *data tier* manages the organized information that the application needs. The usual participants in this tier are database management systems and LDAP repositories. The next layer is the *EJB tier*, which contains the entire business logic of the application — it manipulates data obtained from the data tier and passes along the results. The *web tier*, which is not always present, obtains the processed data from the EJB tier and presents it to the client's web browser. This structure is depicted in Figure 1.

The EJB tier usually provides various means for clients to access it. There are interfaces to both clients within the same JVM (i.e., the web tier) or remote clients. Thus it is possible for a user to access the functionality of a well-designed JEE application by going to the web site of the application and browsing its web tier, or by running a desktop application designed specifically for that purpose, which connects directly to the EJB tier through Java RMI or a similar remoting mechanism.

2.1 Uses of Data Transfer Objects

The information that a client (be it the Web tier or a remote application) receives from the EJB tier could be a primitive value, a Java Collection object, or a DTO that is specific to the particular enterprise application. The client may also create DTOs and pass them to the EJB tier in order to decrease the network overhead. A DTO may even be created in one tier (EJB, web, or remote client), carry its state to another tier, and then the same object may be used to carry new state back to the tier where it was created. This more complex pattern is referred to as *Updatable DTO*.

The single most important part of the definition of a DTO is that it must be capable of being passed over the wire. Indeed, there are multiple explanations and/or definitions of the DTO design pattern [6, 1, 25], but the only property they all strictly require of a DTO class is that it must implement either interface *java.io.Serializable* or (in extremely rare cases) *java.io.Externalizable*. Figure 2 shows a part of a DTO class from the EJB Certificate Authority (EJBCA [9]) application used in our experimental study.

```
public class UserDataVO implements Serializable {
  private String username;
  private String subjectDN;
  private int caid;
  private String subjectAltName;
  private String subjectEmail;
  private String password;
  private int status;
  private int type;
  private int endentityprofileid;
  private int certificateprofileid;
  private Date timecreated;
  private Date timemodified;
  private int tokentype;
  private int hardtokenissuerid;
  private ExtendedInformation extendedinformation;
  public void setUsername(String user) { ... }
  public String getUsername() { ... }
  public void setDN(String dn) { ... }
  public String getDN() { ... }
  public int getCAId() { ... }
  public void setCAId(int caid) { ... }
  ...
}
```

Figure 2. A DTO example from EJBCA.

2.2 DTO Lifecycle

A DTO passes through several states during its lifecycle. In State 1, the object has just been created, and it does not yet carry any application-specific state, meaning that its fields have not been initialized. Populating the just-created DTO with application state by initializing its fields leads to State 2. When in State 1 or State 2, the DTO exists within the same JEE tier (we will refer to it as Tier 1); this is usually the EJB tier. The fields of the DTO may be accessed and/or modified in Tier 1 after initialization for various reasons, for example reading a field indicating the application-specific status of the DTO.

The DTO enters State 3 when it is passed to an object belonging to a different tier (Tier 2). The application state carried by the DTO is read and/or written in Tier 2. The DTO then might be passed back to State 4 in Tier 1, where its data is read and potentially modified again. In fact, the object can "oscillate" between State 3 and State 4. Finally, when the use-case ends, the DTO is prepared for garbage collection (in either Tier 1 or Tier 2) and enters State 5. The FSA for this lifecycle is shown in Figure 3.

2.3 Problem Definition

The goal of our work is to perform dynamic analysis of the execution of a JEE application in order to identify

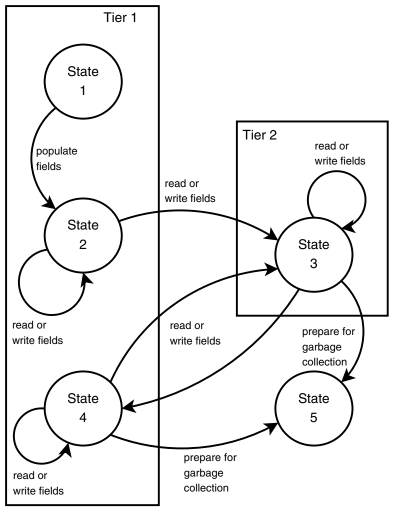

Figure 3. Lifecycle of a DTO.

classes whose instances implement the DTO pattern. To achieve this goal, the analysis tracks the lifecycle events of potential DTO objects, and matches the observed sequence of events (on per-object basis) with the DTO FSA. The key research questions that need to be answered are as follows:

- What specific kinds of run-time information should be collected during the dynamic analysis, and how should this information be used to identify DTO objects?
- What is the false positive rate? That is, how often does the analysis observe objects whose behavior matches the DTO state transition diagram, even though these objects do not actually implement the DTO pattern?
- What is the false negative rate? In other words, how many DTO objects violate the pattern described above, and therefore are not reported by the analysis?
- What is the run-time overhead of the analysis?

3 Dynamic Analysis for DTO Identification

DTOs always carry state, and sometimes also implement application logic. Our approach is focused on the state, which makes it unnecessary to track method call/return or entry/exit events. To identify DTOs, the analysis tracks field accesses (reads and writes) for objects of potential DTO classes, together with the location of the code that triggers the events. The relevant run-time events are observed with the help of the Java Tool Interface (JVMTI), which provides a portable and standardized infrastructure for implementing our dynamic analysis.

Processing at class loading. Classes that are DTO candidates are all Java classes that (1) belong to the enterprise application, and (2) implement interface *java.io.Serializable*.

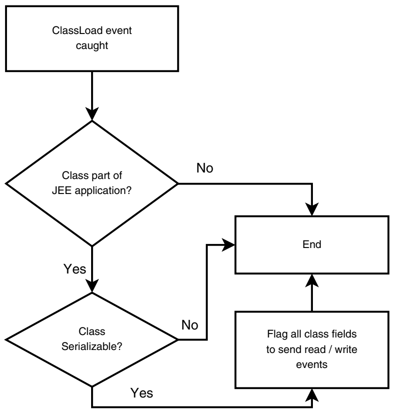
ClassLoad event caught
Class part of JEE application?
No
End
Yes
Class Serializable?
No
Flag all class fields to send read / write events
Yes

Figure 4. Processing at class loading.

The analysis intercepts all *class load* events within the application server's JVM, considers whether the loaded class is a DTO candidate, and if so, tags all its fields. Tagging enables run-time events related to tagged fields: whenever a field in such a class is accessed (read or written) anywhere within the same JVM, a JVMTI event is generated and processed by our analysis. Events are generated for a tagged field regardless of whether the field is static (i.e. the field directly pertains to a DTO candidate class) or instance (i.e. the field belongs to an object that is an instance of a DTO candidate class). Figure 4 illustrates this component of the analysis.

Processing of field reads and writes. When a *field read* or a *field write* event is observed, the analysis identifies the object that caused that event and the location (i.e., application tier) of that object. Figure 5 summarizes this processing. To find the object that caused the event, the analysis obtains a handle to the call stack of the current thread. A traversal of the stack frames, starting from the most recent one, identifies the first method that belongs to an application class different from the class that the field belongs to. The receiver object of that method is the one accessing the candidate DTO.

The analysis has to consider the state of the run-time call stack because it is very common for DTOs to implement getter and setter methods. If a field is read through a getter, or written through a setter, the top call stack frame would be within the context of a method belonging to the same class as the field. However, we are interested in the object that caused the field read or write to happen in the first place.

Note that the traversal of the call stack must identify a method that belongs to an application class. This is necessary due to the frequent use of reflection in JEE applications and application servers. We match the current frame method's class against the packages of the JEE application to ensure that we do not identify the cause

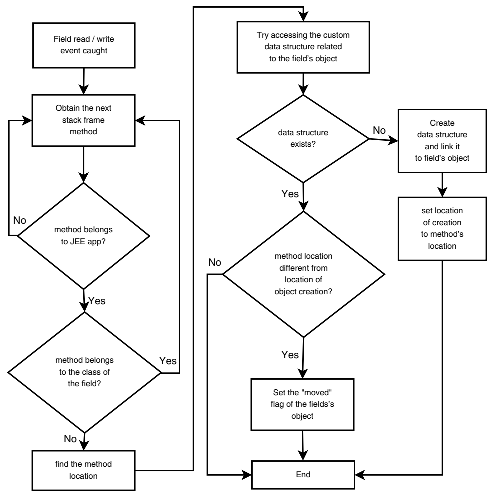
Field read / write event caught
Obtain the next stack frame method
method belongs to JEE app?
No
Yes
method belongs to the class of the field?
Yes
No
find the method location
Try accessing the custom data structure related to the field's object
data structure exists?
No
Create data structure and link it to field's object
set location of creation to method's location
Yes
method location different from location of object creation?
No
Yes
Set the "moved" flag of the fields's object
End

Figure 5. Processing of field reads/writes.

of the field read/write event as a *java.lang.Class* object, *java.lang.reflect.Method* object, or some other irrelevant object which is only used as part of a mechanism internal to the application server.

Once the analysis pinpoints the method causing the field read/write event, it determines the tier that contains the method's declaring class, and thus the tier that accesses the DTO candidate. This determination is done using precomputed lists of class names for each application tier. A simple static analysis of the package hierarchy of the JEE application can be used to create such lists. JEE applications usually follow a strict package hierarchy, with web tier classes consolidated into a package (or a package hierarchy), EJB tier classes consolidated in another package hierarchy, etc. The application is even deployed as a combination of different modules — for example, a web module is deployed as a .war archive, and is strictly separated from other modules. We use that information to find the location of the object (or class, if the method is static) that caused the field read/write event. That location (i.e., application tier) is also the current location of the DTO candidate.

The analysis maintains information about a DTO candidate object, including the fully-qualified class name, the location of its creation, as well as a flag indicating whether the object has moved to a different location during its lifetime. When a field read/write event is observed and the current location of the DTO candidate is identified, the analysis checks whether this is the first such event ever to happen for that object. If this is the case, the object has just been created, and is entering State 2 of its lifecycle. The location (i.e., tier) of its creation is recorded as the object's current location.

If this is not the first such event, the object has already been accessed. If this is not a DTO, its current location should always be the same as the location of its creation. If the object a DTO, however, it may change its location, which means it might be entering State 3 of the DTO lifecycle. If the location of the DTO candidate is different from the location of its creation, the "moved" flag is set to true, which essentially means that the object is marked as a DTO.

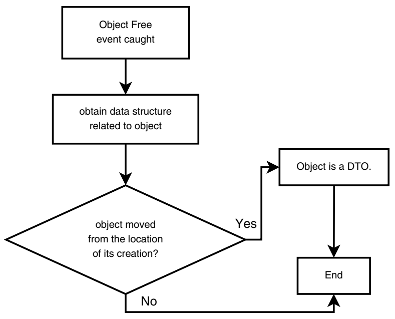
Object Free event caught
obtain data structure related to object
object moved from the location of its creation?
Yes
Object is a DTO.
End
No

Figure 6. Processing at garbage collection.

Processing at garbage collection. When a DTO candidate is being garbage collected, the analysis observes the corresponding event and processes it as outlined in Figure 6. At this time no information about the object is accessible — including its class name — so we have to rely on our custom data structure for all information related to that object. At this stage the analysis simply checks whether the object that is being garbage collected has moved from the location of its creation, and if so, the object's class is reported as a DTO class. Our implementation ensures that we observe a garbage collection event for every DTO candidate by aggressively forcing garbage collection through JVM's internal mechanisms.

4 Implementation Details

We used the Java 6 version of the JVM Tool Interface (JVMTI) to implement the dynamic analysis outlined above. JVMTI provides the various capabilities we need, as well as the required event hooks to the internal workings of a Java 6 JVM. Our code is written completely in C, and interfaces with the JVM through the Java Native Interface mechanism, of which JVMTI is an extension. The JVMTI capabilities that our tool requires (and enables) are the ability of the JVM to generate field read events, field write events, object free events, and the capability for our agent to set and get object tags.

JVMTI provides the useful capability for agents to set and get object tags, which we use extensively. Unfortunately, there is a constraint on the use of JVMTI object tags — an object tag is a single integer, which is supposed to be used by agents as an internally-generated (possibly auto-incrementing) key. A single integer of 32 bits cannot fit all the information tracked about an object. Since our agent is written in C, we simply use for the object tag (of type integer) the memory address of an instance of our custom data structure. The analysis allocates memory for an instance of its custom data structure the first time an object is used, and populates the object's tag with the memory address of that instance. During the lifetime of the object, the agent casts the tag from an integer to pointer and vice versa as needed. Finally, when the object is being garbage collected, the analysis also deallocates the memory used by the associated data structure.

We did consider tracking method entry and exit events instead of field read and write events. Since our goal is DTO detection, which can be achieved through tracking state changes and their location, both sets of events would be able to produce results suitable for interpretation. However, there is an important difference in the implementation of these event hooks in JVMTI. Once the capability of sending method entry/exit events is enabled, every single such event (i.e. every call stack push or pop) is reported, which introduces very significant overhead. On the other hand, it is possible to flag only certain fields, and manipulating only those fields would trigger field read/write events. To evaluate run-time overhead, we performed a simple test with only method entry/exit events enabled. The EJB application server took over two hours to initialize and start, with no enterprise applications running or even deployed. As a result of this test, we decided to track state changes through the much more lightweight field read/write events.

A useful optimization that JVMTI provides automatically is its sending garbage collection events only for objects which have had their tag set. This functionality ensures that events are sent when objects tracked by our analysis are garbage collected, but no overhead is incurred for any other objects. We have not made any other efforts to optimize the implementation of the algorithm or the data structures. Clearly, there is ample room for improvement in terms of optimizations, which will be pursued in our future work.

5 Experimental Study

The dynamic analysis was evaluated on EJBCA [9] — the J2EE Certificate Authority version 3.4.1, running on the JBOSS Application Server version 4.0.5 [13]. EJBCA is a J2EE 1.3 (EJB 2.0) application consisting of approximately seven hundred Java classes. It was chosen for our experiments because of its size, robustness, source code availability under the Lesser General Public License (LGPL), and the fact that it is an industrial-strength enterprise application that has been successfully deployed on a number of web sites. JBOSS currently is the most widely used Java Enterprise application server, and its source code is available under the LGPL. We used HSQLDB — the Java-based database engine that is bundled with JBOSS — for the database layer. The machine used for the experiments has an AMD Athlon XP 2200+ CPU running at 1798.751 MHz with 1 GB RAM, under Ubuntu 6.06 LTS GNU/Linux. JBOSS, together with EJBCA and HSQLDB, was run through Sun's Java 6 JVM.

Analysis precision. We tested various uses of EJBCA by accessing the web tier from a web browser. The results can be summarized as follows. There were a total of 132 classes that implement *java.io.Serializable* and are part of EJBCA, and whose bytecode was loaded from disk by the JVM for potential execution. Since we had no other way of detecting the actual EJBCA DTOs, and because documentation is almost nonexistent, we had to manually inspect the source code of every potential DTO class (i.e., each of these 132 classes) and decide whether that class was a DTO or not based on our experience with JEE applications. After this manual analysis, 13 out of the 132 classes were deemed to be DTO classes. Of those 13, 11 were actually used by EJBCA when the test cases were executed. (The other two remained loaded and prepared, but unused — no objects were instantiated from them at any point.) Ideally, the dynamic analysis would report these 11 classes.

The dynamic analysis reported 11 classes, of which 10 were DTO classes as determined by he manual examination of the code. These 10 classes demonstrated various levels of complexity of behavior. There were classes that were true DTOs and nothing more — they only had fields, constructors, getters, and setters. There were classes that implemented *java.lang.Comparable* in addition to being DTOs, which means they had additional logic for comparison. There were classes that had many fields, with the corresponding getters and setters, but also had methods capable of returning additional information based on the values of the fields and complex hashing algorithms. Finally, there was a class *org.ejbca.util.Query* that had four fields, two of which were Vector references, but it never returned them directly. The fields of that class were only used to answer certain boolean queries through corresponding methods.

The single false positive that our analysis reported was *org.ejbca.util.Query*. This class may actually be considered a DTO. It was reported by the analysis because its instances move between the JEE tiers. The problem is that these *Query* instances never allow other objects to have direct access to their fields through setter and getter methods, and as such the class does not fit the "standard" definition of a DTO. Thus, in the manual examination of the code it was classified as a non-DTO class. However, the purpose of this class is still to carry data between the tiers, and it allows some access (albeit very circumspect) to this data. Arguably, this class implements the DTO pattern "in spirit".

Of the 11 classes that were manually determined to be DTOs and were also used during the execution of the test cases, 1 class was not reported by the analysis — that is, this was a false negative. The reason the class was not reported is because it was wrapped by another DTO class. The agent detected the wrapper DTO, not the inner one, because the cause for all reads/writes for the inner DTO is actually the wrapper DTO, according to the analysis algorithm (the wrapper is different from the inner DTO, but it still is a part of EJBCA, and belongs to the same tier). Consequently, according to the dynamic analysis, the inner DTO has never moved because all locations that cause field reads and writes (i.e., methods of the wrapper DTO) are part of the same tier as the inner DTO.

In summary, the analysis correctly identified 10 DTOs, and had one false positive and one false negative. These promising results indicate that DTO identification can be performed with high precision using the proposed run-time analysis techniques.

Analysis cost. We measured the start-up time of JBOSS with EJBCA deployed with and without our agent running in the background. The purpose of this test is to estimate the startup overhead due to our analysis. The application server started completely in 1 minute and 32 second without our agent, and in 2 minutes and 56 seconds with it. These results correspond to run-time overhead of about 91%.

We also ran a batch of test use-cases against the RMI interface of EJBCA to track the execution time with and without our agent. We used this method to estimate the overhead, as opposed to tracking the times for the web use-cases, because of the event-driven structure of our analysis. When testing web use-cases, both the web tier and the EJB tier are located within the same JVM. Due to the HTTP session objects the web tier keeps for individual users, it would be nearly impossible to draw hard lines between the separate test use-cases in terms of memory and processing time, unless we tracked the time it took every single method to execute and return. As discussed earlier, tracking method entry/exit events incurs impractical overhead. We chose instead to test the RMI interface of the EJB tier only, and to track the execution time on the client side. Because of the remote client, in this experiment our analysis did not report any DTOs — from its perspective, no objects were ever moved to a client tier. However, the analysis still responded to all relevant events and performed all stages of the algorithm, resulting in a meaningful estimate of overhead.

The running time of the batch of remote tests without our agent was 4 minutes and 53 seconds. The corresponding time with our agent turned on was 17 minutes and 44 seconds. These results correspond to run-time overhead of approximately 263%. While significant, this overhead is not prohibitive and the approach remains suitable for practical use. As mentioned earlier, no attempts were made to optimize the performance of the analysis implementation; future work may be able to improve this performance and to reduce significantly the run-time overhead.

6 Related Work

Researchers have proposed a number of techniques based on static analysis to recover design patterns from existing programs (e.g., [2, 15, 3, 21, 18, 19]). There is also a body of work related to formalizing design patterns (e.g., [23, 26, 4]). Such formalizations can later be used to match structural patterns in the source code of a program to the structure of a design pattern. However, many design patterns in general, and the DTO pattern in particular, have significant behavioral aspects that static analysis cannot capture precisely. For example, in order to model precisely the temporal sequence of events that constitutes a pattern instance, a static analysis may have to employ expensive algorithms with flow/context/path sensitivity, which creates significant scalability challenges for real-world Enterprise Java applications. As another example, dynamic features such as reflection and dynamic class loading (commonly used in enterprise Java applications) present a serious challenge for static analysis. Even though some existing work has addressed scalability problems related to analysis of FSA-based properties for large programs (e.g., [10, 8]) as well as handling of dynamic features (e.g., [24, 16]), the current state of the art does not provide enough evidence that static analysis of patterns in EJB application can achieve correctness and precision at a practical cost. Thus, we believe that for such applications the use of dynamic analysis is a more natural choice, at least until more advances are made in static analysis research.

Wendehals and Orso [28] propose an approach that combines static and dynamic analysis. A static analysis examines structural properties in order to identify pattern-instance candidates. A dynamic analysis considers a set of such candidates and checks whether the run-time interactions match the behavioral properties of the pattern. A FSA is used to match the observed method calls to the expected behavior. This approach is similar to our work in that it uses a predefined FSA-based abstract pattern specification to capture relevant program behavior and match it to design patterns. More generally, there is a body of work on performing dynamic analyses based on properties expressed by finite state automata (e.g., [5, 7, 12, 14]). Our work uses a similar technique, with the focus being on (1) reads and writes of fields, and (2) the application tier that initiates the read or write, as determined by examining the run-time call stack. Most existing approaches target properties related to method entry/exit events, with or without information about the identity of the receiver object. However, our experience with JVMTI indicates that such an approach may be impractical for JEE applications due to the complexity of the underlying middleware (i.e., a JBOSS application server), which leads to substantial run-time overhead.

7 Conclusions and Future Work

JEE applications present various challenges for software understanding, testing, debugging, verification, and optimization. The identification of DTOs is one of numerous software engineering problems in this domain. We propose a dynamic analysis which tracks FSA properties related to the access patterns for object state and the movement of objects between application tiers. Our preliminary results indicate high analysis precision, achievable at practical cost.

Future work includes various optimizations of the implementation, as well as improving the completeness and precision of the algorithm. An example optimization is compressing most of the information used to track an object into a single byte. Another example is avoiding false negatives of the type described previously (a DTO wrapping another DTO) by tracking the classes that cause a field read/write event, and matching them against the list of already found DTOs. More generally, in future work we plan to consider dynamic analyses for JEE applications for various FSA-based properties.

References

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns, Second Ed.* Prentice Hall PTR, 2003.

[2] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *International Workshop on Program Comprehension*, pages 153–160, 1998.

[3] E. L. A. Baniassad, G. C. Murphy, and C. Schwanninger. Design pattern rationale graphs: Linking design to source. In *International Conference on Software Engineering*, pages 352–362, 2003.

[4] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of design patterns in Java. In *International Conference on Automated Software Engineering*, pages 224–232, 2005.

[5] E. Bodden. J-LO: A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, Nov. 2005.

[6] W. Crawford and J. Kaplan. *J2EE Design Patterns*. O'Reilly and Associates, 2003.

[7] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *International Workshop on Dynamic Analysis*, 2005.

[8] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *International Symposium on Software Testing and Analysis*, pages 12–22, 2004.

[9] *EJB Certificate Authority*. `www.ejbca.org`.

[10] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis*, pages 133–144, 2006.

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

[12] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[13] *JBoss Application Server*. `jboss.org`.

[14] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. V. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[15] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering*, pages 208–215, 1996.

[16] B. Livshits, J. Whaley, and M. Lam. Reflection analysis for Java. In *Asian Symposium on Programming Languages and Systems*, LNCS 3780, pages 139–160, 2005.

[17] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, Nov. 2001.

[18] J. Niere, W. Schaefer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *International Conference on Software Engineering*, pages 338–348, 2002.

[19] G. Pappalardo and E. Tramontana. Automatically discovering design patterns and assessing concern separations for applications. In *ACM Symposium on Applied Computing*, pages 1591–1596, 2006.

[20] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[21] J. Seemann and J. W. von Gudenberg. Pattern-based design recovery of Java software. In *Symposium on the Foundations of Software Engineering*, pages 10–16, 1998.

[22] M. Sharp and A. Rountev. Static analysis of object references in RMI-based Java software. *IEEE Transactions on Software Engineering*, 32(9):664–681, Sept. 2006.

[23] N. Soundarajan and J. O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *International Conference on Software Engineering*, pages 666–675, 2004.

[24] V. Sreedhar, M. Burke, and J. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Conference on Programming Language Design and Implementation*, pages 196–207, 2000.

[25] *Transfer Object Pattern*. `java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html`.

[26] B. Tyler, J. O. Hallstrom, and N. Soundarajan. Automated generation of monitors for pattern contracts. In *ACM Symposium on Applied Computing*, pages 1779–1784, 2006.

[27] R. Veldema and M. Philippsen. Compiler optimized remote method invocation. In *IEEE International Conference on Cluster Computing*, pages 127–137, 2003.

[28] L. Wendehals and A. Orso. Recongizing behavioral patterns at runtime using finite automata. In *International Workshop on Dynamic Analysis*, pages 33–40, 2006.