# DOSSO – AUTOMATIC DETECTOR OF SHARED OBJECTS IN MULTITHREADED JAVA PROGRAMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUNARA TOLUBAEVA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

MARCH 2009

Approval of the Graduate School of Informatics

____________________

Prof. Dr. Nazife Baykal
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

____________________

Prof. Dr. Yasemin Yardımcı
Head of Department

This is to certify that we have read this and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

____________________

Assist. Prof. Dr. Aysu Betin Can
Supervisor

Examining Committee Members

| | | |
|---|---|---|
| Prof. Dr. Semih Bilgen | (METU, EEE) | __________________ |
| Assist. Prof. Dr. Aysu Betin Can | (METU, II) | __________________ |
| Assist. Prof. Dr. Erhan Eren | (METU, II) | __________________ |
| Dr. Sevgi Özkan | (METU, II) | __________________ |
| Assist. Prof. Dr. Tuğba T. Temizel | (METU, II) | __________________ |

**I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and reference all material and results that are not original to this work.**

| | |
|---|---|
| **Name, Surname:** | **Munara Tolubaeva** |
| **Signature:** | ________________ |

# ABSTRACT

DOSSO – AUTOMATIC DETECTOR OF SHARED OBJECTS IN MULTITHREADED JAVA PROGRAMS

Tolubaeva, Munara
M.S., Department of Information Systems
Supervisor: Assist. Prof. Dr. Aysu Betin Can

March 2009, 68 pages

In this thesis, we present a simple and efficient automated analysis tool called DoSSO that detects shared objects in multithreaded Java programs. DoSSO reports only the shared objects that are modified by at least one thread. Based on this tool, we propose a new approach in developing concurrent software where programmers implement the system without considering synchronization issues first and then use appropriate locking mechanism only for the objects reported by DoSSO.

To evaluate the applicability of DoSSO, we have conducted a case study on a distributed and concurrent system with graphical user interfaces. Case study results showed that DoSSO is able to identify objects that become shared among explicitly defined threads and event threads, and objects that become shared through RMI.

Keywords: Concurrent programming, Static analysis, Automation, Java

# ÖZ

DOSSO – KOŞUTZAMANLI JAVA PROGRAMLARDAKİ ORTAK KULLANIM NESNELERİNİN OTOMATİK TESPİT EDİLMESİ

Tolubaeva, Munara
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Yrd. Doç. Dr. Aysu Betin Can

Mart 2009, 68 sayfa

Bu tezde koşutzamanlı Java programları içindeki ortak kullanım nesnelerini otomatik olarak bulmak için bir analiz yöntemi geliştirilmiştir. Bu analiz yöntemi geliştirdiğimiz DoSSO adındaki araç ile hayata geçirilmiştir. DoSSO analiz sonucunda ortak kullanımda olan ve en az bir iş parçacığı tarafından yazılan nesneler raporlanmaktadır. Aracımız sayesinde programcı senkronizasyonu düşünmeden yazılımını yapabilir ve daha sonra DoSSO sonuçlarına gore sadece gerekli nesneleri uygun bir senkronize mekanizması kullanabilir. Bu şekilde zor olan senkronizasyon gerçekleştirimi en sona bırakılır ve iş mantığı üzerine yoğunlaşılır. DoSSO'nun etkinliğini değerlendirmek amacıyla, kullanıcı arayüzüne sahip dağıtımlı ve koşutzamanlı bir sistem üzerinde durum çalışması gerçekleştirilmiştir. Durum çalışmasında DoSSO'nun belirtilmiş iş parçacıkları aralarında ortak kullandıkları nesneleri; uzaktan erişim nesneleri sebebi ile ortak kullanılan nesneleri; ve kullanıcı olaylarını yakalayan iş parçacığı ile belirtilmiş iş parçacıkları arasında ortak kullanılan nesneleri etkin bir şekilde yakaladığı izlenmiştir.

Anahtar Kelimeler: Koşutzamanlı Proğramlama, Statik Analiz, Otomasyon, Java

**To my dearest father and mother**

## ACKNOWLEDGEMENTS

First and foremost, I express my deepest gratitude to my advisor Assist. Prof. Dr. Aysu Betin Can for introducing me to the fields of program analysis, software verification and software engineering in general. Moreover, I thank my advisor for her support, motivation, patience and understanding during two years of collaborative work in the project.

I would like to thank my friends Erdal Kuzey, Refika Köseler, Gülgün Afacan and Erkan Er for their endless help during my studies, and for supporting me every time when I felt myself hopeless and unenthusiastic.

Lastly, I thank my family for their motivation and faith in me.

# TABLE OF CONTENTS

## LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

## 1.1 Introduction

Developing concurrent programs is quite a challenging work, because it is hard to manually spot and protect all objects accessed by different threads. If not well designed, shared objects in a program may create some conflicts which may directly or indirectly affect the flow of a program. Thus, paying attention to shared objects is the most important part of concurrent programming.

Unfortunately, history clearly showed us what can be the costs of developing concurrent systems without properly protecting shared objects. A widely known disaster caused two cancer patients to get fatal radiation overdoses in 1986 in a computer controlled radiation therapy machine called THERAC – 25 [1, 2]. The reason of disaster was mishandling synchronous accesses to a shared object in THERAC – 25. Another disaster which caused the breakdown of Energy Management System and was a reason for North America blackout in 2003 also happened because of unprotected shared variables in the system. The blackout disaster caused financial losses of about 6 billion USD [3, 4].

Although, the operating systems such as Symbian and Linux developed in last decade improved ways of handling concurrency issues successfully at operating system level [24, 25], concurrent programming still remains as a challenging concern for application developers.

## 1.2 Our Approach

In current multithreaded programming methodologies, programmers manually identify objects that are accessed by multiple threads and implement a synchronization mechanism to protect them. This process, however, is not always conducted flawlessly. It is possible to overlook some of the shared objects or to assume that the object will never be shared at all. In such situations, programmers miss some objects and do not protect them using any synchronization methods, which cause conflicts in the program execution later on.

To avoid this problem, we suggest a different approach in developing multithreaded Java programs. We suggest the developers to implement their system without considering any synchronization issues first. Then, our automated tool called DoSSO informs developers about the shared objects that need to be synchronized. Finally, developers protect these shared objects using an appropriate locking mechanism with a verification support, such as the concurrency controller pattern [8]. The verification support in [8] can also be used with Java concurrency utilities [9, 10] to detect faults in lock ordering and unprotected access to the given shared variables. In this way, our tool supports the design for verification paradigm [11] and also helps developers to realize more reliable multithreaded programs.

## 1.3 DoSSO

We have developed a tool which performs a static analysis to detect shared objects in a multithreaded Java program. DoSSO outputs only risky shared objects. Risky shared objects are the objects that are accessed by more than one thread, and modified by at least one thread.

DoSSO also determines potentially shared objects of programs that use graphical user interfaces (GUI) and remote method invocation (RMI). Programs involving GUI are considered as multithreaded programs because of an implicitly created thread by the runtime environment called the event dispatch thread that is created due to the GUI framework. Event dispatch thread is responsible for capturing user events and invoking the corresponding event handling methods, meaning that objects accessed or modified inside event handling methods become automatically shared. Event handling methods are the methods that are called when user performs some kind of actions at the UI level of a program, i.e. clicks a button, presses a key etc. So, our tool handles programs with GUI framework and is able to detect objects that became shared with the event dispatch thread.

As stated above, we also handle programs containing RMI. We consider there is a thread responsible for executing the remote calls and this thread's methods are the remote methods, i.e. the RMI methods. Consequently, objects and fields accessed or modified inside RMI methods are seen as being accessed in different thread scopes. That is why we deal with finding objects that become shared because of RMI methods.

## 1.4 Summary of Contributions

The contributions of this work are as follows:

- Instead of dealing with issues such as race condition after the program is developed, we suggest the prevention of such faults by reporting the risky objects and save the effort of developer in considering concurrency issues.

- Besides handling simple assignment and method call expressions in an input program, we also detect shared objects in distributed programs with remote method invocations (RMI)

- We also handle event handling invocations, specifically the events in programs with graphical user interfaces (GUI). In Java, the user interactions are represented as events. These events are handled by an implicit thread that invokes designed event handling methods. This means objects that are accessed by event handling invocations become shared.

## 1.5 Organization

The thesis is organized as follows:

**Chapter 2** briefly gives an information about concurrent programming, critical section problem and explains mechanisms how to avoid data synchronization problem. Moreover, this chapter describes the concepts of race condition and escape analysis which are closely related to our research.
**Chapter 3** describes the analysis technique we have developed to detect shared objects in a program.
**Chapter 4** introduces our tool "DoSSO" and describes its components and algorithms used in the tool in detail.
**Chapter 5** contains information about our small experiments, the case study and the experiment results of case study with "DoSSO".
**Chapter 6** describes advantages and limitations of our tool and gives concluding remarks on this thesis.

# CHAPTER 2

# BACKGROUND AND RELATED WORK

## 2.1 Concurrent Programming

When looking into the operating systems' evolution history, we see that scientists and engineers have always worked on to improve computers' performance quality. They tried to develop new operating systems (OS) which overcome previous versions in terms of response and process time, efficiently use of system resources, multitasking features etc. To improve OS performance issues, scientist developed multitasking OSs like UNIX, Windows NT, where system resources are strictly controlled by OS, multiple tasks or processes are able to run simultaneously and for most of multitasking OSs multiple user operation mode is provided. Processes could be executed independently, as well as communicate with each other if needed [5]. Still, there was a performance drawback such that processes were single threaded, that is a process could do only one thing at a time. The concept of multithreaded programming evolved at this point. "Thread is a different stream of control that can execute its instructions independently, allowing a multithreaded process to perform numerous tasks concurrently" [7]. With the use of multithreaded programming features, programs can run GUI operations in one thread, wait for I/O instructions in another thread, and process the information in a totally different thread.

Concurrent and multithreaded programming attracts the scientists mainly because it enables programmers to write efficient programs where the idle time wasted by CPU is kept at minimum. In a single threaded program, some tasks like user input, database and networking transactions are processed much slower than the computer processes information obtained from those tasks, meaning that CPU has to wait for these tasks to finish before processing obtained information. However, in multithreaded program, CPU can process these tasks simultaneously, and obtain results from tasks much faster than that of the execution of single threaded program [5]. Moreover, another reason to use multithreaded programming is that threads simplify and divide the overloaded single threaded programs into small clear separate tasks, where each task executes only its own operations [7].

When developing concurrent applications, programmer deals with controlling multiple threads and a shared memory. In this thesis we call the variables reside in this shared memory as shared variables. The essential part of concurrent programming is to correctly handle the accesses to these shared variables. If shared variables are not protected from being concurrently used by multiple threads, they can cause some problems during the execution of a program.

In sections 2.2 through 2.5 we explain these problems in more detail, and present some mechanisms in the literature that are developed to prevent them.

## 2.2 Critical Section Problem

Critical Section is a section of code where threads access and modify the values of shared variables. This section of code must be performed atomically, that is nothing interrupts the thread to execute all statements in critical section [7, 12]. This atomicity is necessary to avoid inconsistencies. Critical sections are the most important parts in a program code, because they contain shared variables which can be accessed and modified by multiple threads concurrently, which then may cause some conflicts during or at the end of execution. Because of this reason, careful attention should be paid while writing critical sections of code.

## 2.3 Data Synchronization Mechanisms

In order to write reliable, bug – free concurrent programs, different types of synchronization mechanisms and algorithms have been developed. These mechanisms provide the atomicity required for the critical sections of a program.

### 2.3.1 Semaphores

A binary semaphore S is a non negative integer, and has got 2 operations which are Wait(S) and Signal(S). It can take values only 0 or 1 which control the access to critical section.

Java did not support explicit semaphore primitives before J2SE 5.0 was developed. In J2SE 5.0, the Semaphore interface was added to the core Java libraries [5]. However before J2SE 5.0, Java has supported lock mechanism which is similar to semaphore logic. Every Java object has an implicit lock, and this lock can be implemented as an implicit binary semaphore inside a block with “synchronized” keyword.

Locks are used with 2 operations lock and unlock which correspond to Wait(S) and Signal(S), respectively. In fact, a Java lock is the same as a binary semaphore with an initial value 1 except two differences. The first distinction between binary semaphores and locks is that in locking mechanism locks are released only by the holders of the lock, whereas binary semaphores do not have such a restriction and can be incremented by any arbitrary process. The second distinction between them is that unlocking an unlocked lock has no effect in Java, whereas in binary semaphores

Signal(S) still is stored in the value of S though there is no any process waiting for the semaphore.

### 2.3.2 Monitors

Java provides monitors as built in synchronization primitives. Each monitor object has an implicit lock. This implicit lock guarantees mutual exclusion during the monitor implementation i.e. it does not allow multiple threads to access monitors procedures at the same time. Monitor object in Java is an object which uses the "synchronized" keyword in all its methods' definitions. Operations on a condition variable are as follows in Java:

wait() – which is the same as wait(C)
notify() – equivalent to signal(C)
notifyAll() – equivalent to signal(C), but wakes up all waiting threads.

Java monitor has got only one condition variable, and above operations refer to it automatically. However, explicit objects can be used as condition variables in Java monitor. Java monitors use *signal* and *continue* signaling discipline.

### 2.3.3. Explicit Locking

When one wants to provide mutual exclusive access to shared data the "synchronized" keyword is easy to implement for small programs. However, as the scale and concurrent executing components increase in the program, the complexity of writing this program using "synchronized" keyword also increases. In fact, it is really hard to develop large scaled reliable concurrent programs using only this mechanism. The main reason lies in the fact that synchronized mechanism restricts programmer to implement the critical section code inside the method block at maximum. When the program exits from the synchronized method the lock automatically is released. Using synchronized mechanism one is not able to control the flow of when to grab and when to release the lock. Because of this reason, explicit locking idea was introduced with J2SE 5.0. The Lock Interface was designed to provide the freedom of lock usage and with more functionalities than that of synchronized mechanism.

## 2.4 Race Condition

Our work is closely related to the detection of race condition concept. A race condition occurs in a multithreaded program where pair of threads access shared variables without any order, and at least one of accesses is a write operation. It is a flaw in the system where the output or the result of the execution is unexpectedly and critically depends on the sequence of thread interleavings.

A simple example explaining race condition would be as follows. There are two threads `T1 and T2` reading from the same variable `sh`. Consider this execution: `T1` reads the value of `sh` and then `T2` reads the same value from `sh`. Then both threads do some computation on the value and then want to update the variable `sh`. Here these threads race to see which one can write its value to `sh`. The thread that writes to `sh` first loses its information since it is overridden by the other thread. Note that here the sequence of which thread takes control effects the result of the program, i.e. the final value of the variable `sh`.

There has been a massive amount of work in building both dynamic and static analysis tools for detection of data races [4, 13-16]. As opposed to detecting the race condition after the software is fully developed, we aid the programmer by pointing out the objects that have to be protected before realizing the locking mechanism.

Eraser [14] is the best known dynamic data race detection tool for lock-based multithreaded programs. Eraser checks that all shared-memory accesses follow a consistent locking discipline using the lockset algorithm. DoSSO employs static time analysis as opposed to Eraser and it does not deal with lock ordering or errors in implementing locking mechanism. These errors can be prevented and detected by a design for verification approach which is complemented with our tool.

RacerX [13] is a static analysis tool for detecting race conditions and deadlocks in Java programs. Similar to Eraser, this tool also uses lockset algorithm. We defer the errors in locking mechanism to the error prevention technique such as [8]. On the other hand, our tool can be extended with lockset algorithm to improve the precision of the results of the tool tailored for the programs that already contain the synchronization mechanism.

## 2.5 Escape Analysis

Our work is strongly related with escape analysis, which has been used to analyze the scope of objects especially in the last decade by many research projects. Escape analysis is used to determine the objects that are

1. allocatable on the stack and
2. accessed only by a single thread

The goal of escape analysis is to determine objects that are local to a thread or method scope. Consequently, researches involving escape analysis are all related with stack allocation, synchronization elimination methods and optimization of code.

In their escape analysis, Choi et al. [17] store information about how heap allocated objects and their references are connected in a connection graph to detect objects that are local to a method or local to a thread.

Whaley et al. [18] have used points – to escape graph in their analysis. Their graphs represent objects, references between objects and information about which objects escape to other methods or threads. Other than the difference in goal, their combination of analysis results of each method, i.e. the interprocedural analysis is different than ours. They divide the program into analyzed and unanalyzed sections. The system may or may not check unanalyzed sections for escaped objects. When system does not check unanalyzed sections by skipping invoked method calls in a method, it looks only into analyzed sections of the program and determines stack allocatable objects. Then, objects passed as parameters to unanalyzed parts of a program are considered as method – escape objects. When system checks unanalyzed sections for escape objects, it then combines objects detected in unanalyzed section with objects in analyzed sections, which gives more precise results than checking only analyzed sections. On the other hand, our algorithm does not skip the analysis of invoked method. Instead it looks for the summary information of the invoked method whether it is completely analyzed or not. If it is complete, our algorithm updates current method information with the invoked method's summary information. If it is not complete, current method waits for the invoked method to be completed, and only then it updates its information.

The Marmot system developed by Fitzgerald et al. [19] performs synchronization elimination as one of its optimization purposes. It optimizes single threaded programs using Instantiation and Invocation Analysis (IIA). The system checks whether thread objects are started. If not, the system removes all synchronization operations from the program. However, this logic is insufficient in GUI-based programs, because even though GUI-based programs do not create any explicit thread objects, GUI methods are invoked by a separate framework which can be thought as different thread.

# CHAPTER 3

# THE METHODOLOGY

We have developed an analysis technique, realized as the tool DoSSO, to detect potentially shared objects that have read – write relationships between different threads. We also detect objects used by RMI objects and by event dispatching threads as in GUI – Swing framework. In our analysis, we thoroughly investigate these objects and operations performed on them in each method to obtain their scope and escape information. Escape information of an object is the information about whether the object escapes the scope of thread that created it, i.e. whether other threads may access the object or not. We store this information in a structure called Graph of Method (GoM) which is produced for each method definition and Object Table (OT) which stores the list of all objects declared in the input program. GoM and OT are comprehensively explained in section 3.2 and 3.3.

We base our analysis technique on the observation that objects explicitly become shared in three ways:

- Used with explicit Thread classes and Runnable interface
- Used with GUI framework
- Used with RMI Thread

Based on these three conditions, our tool classifies objects into three categories: 1) Local to thread, 2) Shared by threads, but not used (i.e., the threads just read these objects), 3) Shared and used by threads (i.e., visible to more than one thread and at least one of them writes to this object). We mark objects as GREEN, YELLOW and RED representing thread – local, read – read interaction between threads, and read – write interactions between threads, respectively. The level of marking increases from GREEN to RED.

Our technique reports only the objects that are marked as RED since these objects have a potential read – write conflict between concurrent threads. The technique does not report the shared objects that have read – read relationships in order not to overwhelm the users and because they are not critical for the program execution.

In the following sections, we explain different ways of how objects become shared. Then we continue with presenting the GoM structure which is used to store summary information of a method, the OT structure which is used to store object information and give details about implementation of our analysis technique.

## 3.1 Explicitly shared objects

In this section, we give detailed information about how objects become explicitly shared.

### 3.1.1 Explicit Thread classes and Runnable interface

When used in explicit thread classes and classes that implement Runnable interface, objects escape threads explicitly in three ways:

- At thread initiation when objects are passed as an argument to a constructor of a thread,

- At assignment operations, such as when an object is assigned to a shared object,
- At cases when objects are globally/statically declared, and multiple thread objects have a visibility to these objects.

Figure 1 displays a sample code where objects become shared according to observation 1 and 2 above. In line 6, the object `obj` is passed as an argument to the construction of the thread object `t`; hence `obj` becomes accessible both by the main and the thread `t`. Until line 10, `obj` is marked as YELLOW. In line 10, one of its fields is set to `anotherObject`. Before this line, `obj` was only reachable by two threads; however in this line the main thread modifies one of the fields of the object `obj`. Therefore, we say that this object is shared in a risky way and mark it as RED. Moreover, both object references namely `obj.field` and `anotherObject` now point to the same object.

```
...
5. AnyType obj = new AnyType();
6. Thread t = new Thread (obj);
...
10. obj.field = anotherObject;
...
```

**Figure 1 Explicitly Shared Objects Example**

## 3.1.2 GUI Framework – Event Thread

When there are GUI objects in Java program, the runtime environment instantiates a thread implicitly to capture the user interactions. This event thread is called `EventDispatchThread`. It captures the GUI events instantiated by user interactions and invokes the corresponding event listeners for event handling. This means that there is a potential for objects that are accessed by event listeners to be shared.

DoSSO collects all the event handling methods, which are the event listener methods, and treats them as the methods of the event thread. During the analysis, all the determined methods of the event thread are considered as thread entry points. Thread entry points can be understood more clearly with the example given in Figure 2:

```
public class X extends ActionListener{
     ...
     addActionListener(this);
     ...

    public void actionPerformed(ActionEvent e){
         ....
    }
}
```

**Figure 2 Event Handling Methods Example**

In the example in Figure 2, the `actionPerformed` method is actually called by event dispatching thread when the program is running. Because of this reason, we consider this method as a method of an explicit thread, and mark objects passed as an argument as escaped from the current thread scope.

Our analysis technique states that if objects are passed as arguments to the methods of the event dispatching thread, then they escape the current thread and become accessible by more than one thread. We consider the `addActionListener` method as one of event dispatch thread’s methods. Through this method, the event dispatch thread knows which object’s listener methods it will invoke when GUI events are captured. Therefore, we say that `this` object passes as an argument to the method of event dispatch thread, and thus becomes shared, though the `addActionListener` method call performed by the main thread. Moreover, the technique states that objects accessed or modified inside of Listener methods also become shared since the Listener methods are executed by the event thread. For instance, if we had objects used inside of `actionPerformed` method scope, then they would be shared.

### 3.1.3 RMI Thread

In RMI, each remote method can be invoked outside of the current application at any time. There is an implicit thread that serves these remote calls. Without losing generality, we assume that the underlying RMI framework creates one RMI thread for each remote method call. The methods of this RMI thread are the methods declared in the remote interface. As in the event thread case, all of the remote methods are considered as thread entry points.

Figure 3 [7] is a good example for explaining how RMI invocations are thought as thread entry points. When client calls remote object's method, a random thread becomes responsible for executing remote object's method. We assumed that for each remote method there is exactly one explicit thread responsible for executing it.

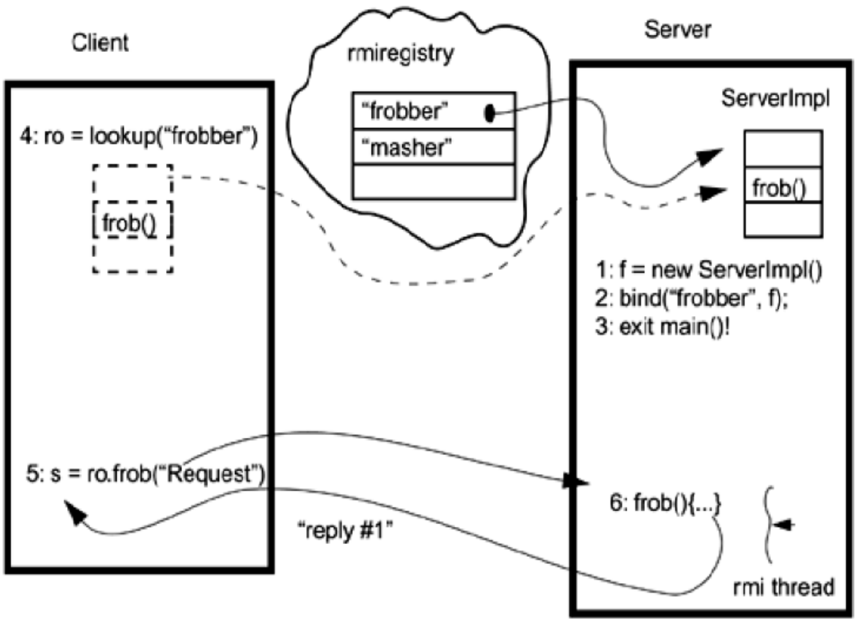


**Figure 3 Implicit RMI Threads [7]**

In Figure 3 [7], we see some parts of server and client implementation. In the server implementation, there is an object `f` which is bound to a `frobber` string. This object is the remote object. The 5th line of code in the client implementation executes `frob` method of the remote object `f` and this execution is carried by one explicit RMI thread which is instantiated at runtime. Since we have stated that all remote method invocations are considered as thread entry points, any remote object accessed or modified in remote methods is marked as shared. In Figure 3, the remote object `f` becomes shared if it accessed in the remote method `frob`.

## 3.2 Graph of Method (GoM)

GoM is developed for storing summary information obtained from the methods DoSSO has analyzed. The GoM structure includes detailed information about a method as well as detailed information about object references used in a method.

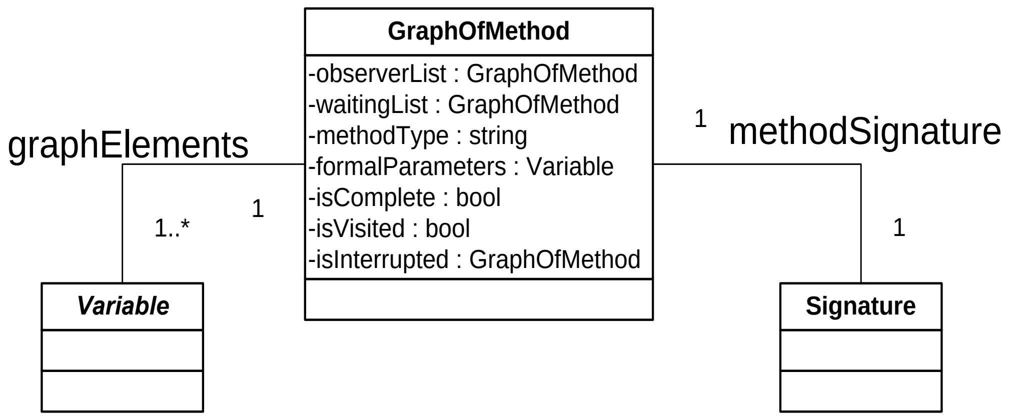


**Figure 4 Graph of Method Structure**

Figure 4 shows the structure of GoM. A GoM contains information about method which comprises:

- Method signature: The signature consists of the names of a method, class and package;

- Method type: This information indicates whether it is a setter or a getter method;

- Formal Parameters: This data stores information about formal parameters of a method.

- Method Completeness: This data indicates whether a method is visited and dependent on other method graphs. A GoM is dependent on other GoM when the other GoM is not complete.

- Point of Method interruption: This data shows whether a method is interrupted by another GoM. GoM A is said to be interrupted when there is a method call to another GoM B, and GoM B is incomplete. Then GoM A waits until GoM B becomes complete.

- Method waiting list: This structure stores the list of GoMs that the current GoM is dependent on and waiting for them to be completed. When there is a method call in a method, current GoM looks for the corresponding GoM that has the same signature as the method call. If obtained GoM is not complete, current GoM waits until it is complete by adding it to the waiting list.

- Method observing list: This is the list which shows GoMs already subscribed for the current GoM.

- Graph Elements: This part includes detailed information about objects created, accessed and used inside of method blocks. These objects are stored as elements of a graph. The Graph Elements are stored in Declared Variables type as shown in Figure 5. So, each element contains information about:

    - Object reference name
    - Object reference type
    - Object signature – gives concrete address of where the object is created including the package, class and method names

- Object Number – every created object has a unique object number which is used to distinguish between objects.
- Object escape type – tells whether the object is local to a method or not.
- Form – tells whether the variable is of composite type, i.e. array, vector, list etc.
- Access Type – keeps information about whether the access to the reference object is write or read.

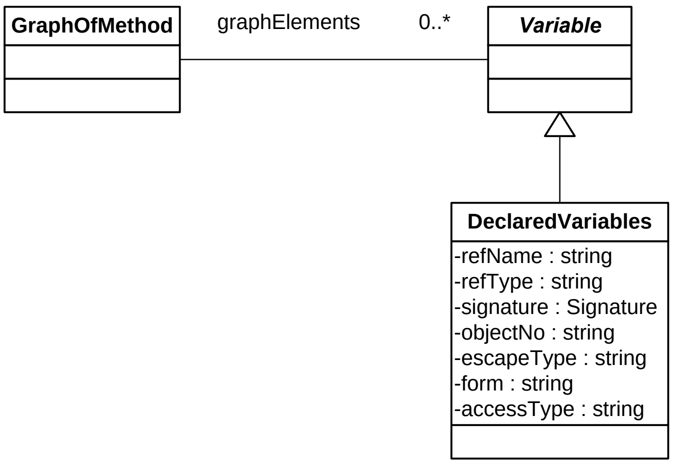


**Figure 5 Detailed View of Elements of Graph Structure**

Graph of Method structure uses Observer design pattern to inform other graphs about its completeness. For instance, when one GoM becomes complete, it notifies all other GoMs that are stored in its observing list about its completeness.

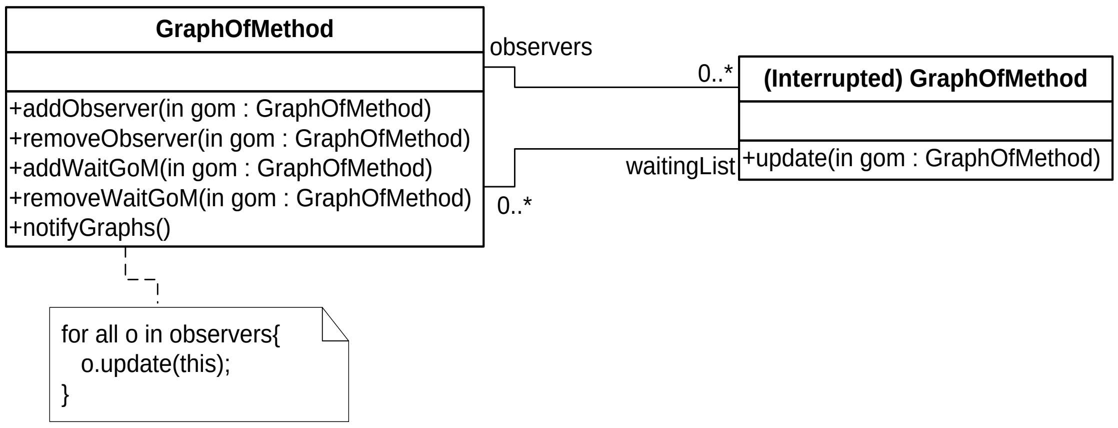

GraphOfMethod
+addObserver(in gom : GraphOfMethod)
+removeObserver(in gom : GraphOfMethod)
+addWaitGoM(in gom : GraphOfMethod)
+removeWaitGoM(in gom : GraphOfMethod)
+notifyGraphs()
observers
0..*
waitingList
0..*
(Interrupted) GraphOfMethod
+update(in gom : GraphOfMethod)
for all o in observers{
o.update(this);
}

**Figure 6 Implementation of Observer design pattern to a Graph of Method Structure**

In Figure 6, the implementation of Observer design pattern in Graph of Method structure is shown. In our design, both publisher and subscriber are of the same type – GoM. GoM keeps the list of observers, which holds the GoMs that are interrupted and dependent on the GoM. Also it keeps the list of GoMs whom it waits, i.e. the list of GoMs it is dependent on. A GoM includes operations like `addObserver` and `addWaitGoM` which add a GoM to the observers and `waitingList` lists respectively; `removeObserver` and `removeWaitList` methods which remove observer from the observers and `waitingList` lists respectively; `notifyGraphs` method wakes up all GoMs inside the observers list by calling their update method. For a GoM to be complete, the `waitingList` list should be empty, showing that it is not interrupted and not dependent on any other GoM.

## 3.3 Object Table (OT)

We choose to keep the track of escape information in terms of objects rather than object references in order to avoid aliasing problems and to be able to match formal and actual parameters. For this purpose, we make use of Object Table (OT) which stores object id, the marking level of the object and the signature of the method where this object becomes shared. When an object becomes shared and marked accordingly all its references which are stored in GoMs become shared automatically.

OT contains the list of entries of type `ObjectTableItem` structure which is shown in Figure 7.

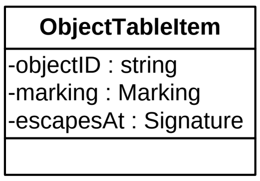


**Figure 7 Object Table Item Structure**

`ObjectID` field is the unique identifier of an object. Thus, OT stores the list of all objects declared in the program with unique `ObjectID`. On the other hand, Graph Elements in GoM store the list of object references, which refer to `ObjectID`s in OT. Multiple object references can refer to the same `ObjectTableItem` in OT.

If object reference becomes shared in the program, DoSSO takes the `objectNo` information of this object reference, and compares `objectNo` with `objectID` field of each entry in OT. When these two strings matches, DoSSO updates the `marking` field of the entry. In this way, we are able to keep markings of objects in consistent way.

## 3.4 The Analysis

To determine potentially shared objects our analysis performs five passes over the input program. In the first pass, we obtain information about Java source files given as an input. In the second pass, we collect classes that extend Thread class, or implement either Runnable or Remote interfaces. In the third pass, we collect names of methods that are called by event dispatching thread. In the fourth pass, we look for objects that become shared by RMI objects. In the last pass, according to conditions stated in section 3.1.1 and information obtained from the previous passes, we analyze objects' escape information.

We do our analysis by building GoMs starting from the thread entry points. Thread entry points are the main method of the input program, run method of each explicit thread, the methods of event thread, and the remote methods of RMI classes. During this traversal when a method invocation is encountered we look for the GoM of the callee method and propagate its information to the GoM of the caller method. While combining these two graphs we do the following:

1. We match the formal and actual parameters first. The sharing information of the formal parameters of the callee is indexed with phantom object identifiers. The parameters are matched in two ways based on the actual parameter's level of marking:

   **a. The actual parameter's level of marking is less than or equal to the formal parameter's level of marking**

   Phantom objects are matched to the actual objects of the call site. The call site object is updated with the information summary of the phantom object in the callee's GoM.

   **b. The actual parameter's level of marking is higher than the formal parameter's level of marking**

   Markings of formal parameters are set to be the markings of actual parameters. Callee method with new formal parameters is re-analyzed i.e. the calle method is revisited by DoSSO again and a new callee GoM is obtained. Afterwards, parameter matching is done according to part a.

2. Next, we add the callee's Graph Elements into the caller's Graph Elements list in the caller GoM.

   If the caller method belongs to a different thread than the callee, then according to the `accessType` information of each object reference, object references become shared. For instance, if the `accessType` of an object reference is read then the object is marked as YELLOW; if the `accessType` is write then the object is marked as RED in OT. We call a method belongs to a thread if it is called directly or indirectly inside of a thread scope.

3. Finally, we handle the escape information of returned object of the callee. If the caller method belongs to a different thread than the callee, then the returned object becomes thread non – local and is marked as YELLOW in OT.

The static analysis presented here is top down in the sense that it starts from the thread entry points which are the main method and the run methods (The entry points for RMI and event thread are handled differently as discussed in sections 3.1.2 and 3.1.3). In the analysis we traverse each method body encountered and produce summary information that is propagated to the method's all calling contexts to achieve context sensitivity.

The analysis results of DoSSO are an over approximation of the risky shared objects. Since DoSSO performs a static time analysis, determining the exact set of RED objects is impossible. Our analysis may produce a few false positives. In other words, DoSSO can report a nonshared object as shared.

The analysis requires that all GoMs to be complete in terms of dependencies to other graphs. In order for a graph to be complete, the graph should not be dependent to other graphs that have not been analyzed yet. While creating a graph for a method, if callee method is not analyzed, the current graph is marked as incomplete and current graph waits for the callee to be completed. When a graph becomes complete, all graphs that are waiting for it update themselves accordingly as explained above.

In order to provide complete information about GoMs used inside the input program, programmers have 3 alternatives:

1. Programmer can write his/her own stub classes. Stub classes are almost empty classes which contain only the method declarations necessary for the input program to compile and perhaps some fields which are used in application classes.

2. Programmer can provide the source code of the Java libraries that are referenced in the input program. In this case, DoSSO will analyze those class files as well as the input program. This way the results will be more accurate but the result will be crowded with the objects of the library classes.

3. Program may use our persistent storage. We provide a saving mechanism for storing already analyzed GoMs. Through this utility, external packages can be analyzed once and the analysis can reuse the GoMs for all the package methods that are stored in our persistent storage.  Currently the classes of the most of the classes of java.util package are in our persistent storage. This alternative will produce faster results than using the source code of the Java libraries. On the other hand, similar to alternative 2, the result will be more accurate than using stubs but crowded with library objects.

# CHAPTER 4

# “DOSSO” TOOL

As all other tools, DoSSO has some instructions that have to be followed so that programmer gets complete and correct results. We will refer these instructions as the requirements. These requirements need to be accomplished before running DoSSO, and they will be presented in the following section. Before stating these requirements, we first explain a framework that DoSSO heavily makes use of and how it is used.

The DoSSO tool takes Java source codes as an input. The tool uses the Java Tree Builder (JTB) framework [20] which performs parsing and visiting operations over the input program. JTB converts each source file into a syntax tree using the Java parser.  This tree representation includes all information from a source code. The conversion into syntax tree structure is done according to Java language semantics. For visiting operations, JTB framework includes several visitor classes which allow one to extend them into a variety of forms depending on one’s developing purpose. The DoSSO tool uses `DepthFirstVisitor` class. `DepthFirstVisitor` is developed using Visitor design pattern; and visits each node in a syntax tree in the depth first order. The purpose of Visitor design pattern is to define operations on elements of an object structure, without changing elements or structure itself [21].

## 4.1 DoSSO Requirements

There are 3 main requirements that have to be fulfilled before executing DoSSO:

1. Since we suggest developer to leave the synchronization issues to the last step of development, we assume that the input program is free of lock operations. Nevertheless, DoSSO is able to accept an input program where locking mechanism is implemented. However, it simply ignores locking operations while analyzing the program and does not store any information about them.

2. The code in an input program should be written in 2 level expressions. If not, it can be converted into 2 level expression using Soot [22], a Java optimization framework. For example,

```
a.b.c.foo();
```

**Figure 8 Multiple Level Reference Expression**

is converted to

```
T temp; T1 temp1; T2 temp2;
temp = a.b;
temp1 = temp.c;
temp2 = temp1.foo();
```

**Figure 9 Two Level Reference Expression**

3. Since DoSSO propagates the method information stored in its GoM to all calling contexts of that method, the input program should contain the source codes of all method and initiation expressions used inside it. When using frameworks, requiring all source code might be a problem. We provide a saving mechanism for analysis results for this purpose. If input program contains classes that have not been preprocessed before, programmer should make sure that she has given all source code. It is needed in order to obtain

complete information about methods, which is consequently needed to get complete results about objects that are escaped.

## 4.2 DoSSO Analysis

The DoSSO Analysis is performed using visitor objects which traverse abstract syntax trees and visit each node in the structure to collect, analyze and store the data required for GoM and to update object information in OT. To perform the analysis, the DoSSO tool uses five different visitors, all extended from `DepthFirstVisitor` class, for five different purposes. These visitors are executed independently from each other, however information obtained from each can be used by other visitors as well.

In Figure 10, the Data Flow Diagram (DFD) for Visitor classes used inside DoSSO is presented. The figure clearly shows how data obtained from one visitor is used by other visitors, and provides information about the order of visitor executions inside the DoSSO.

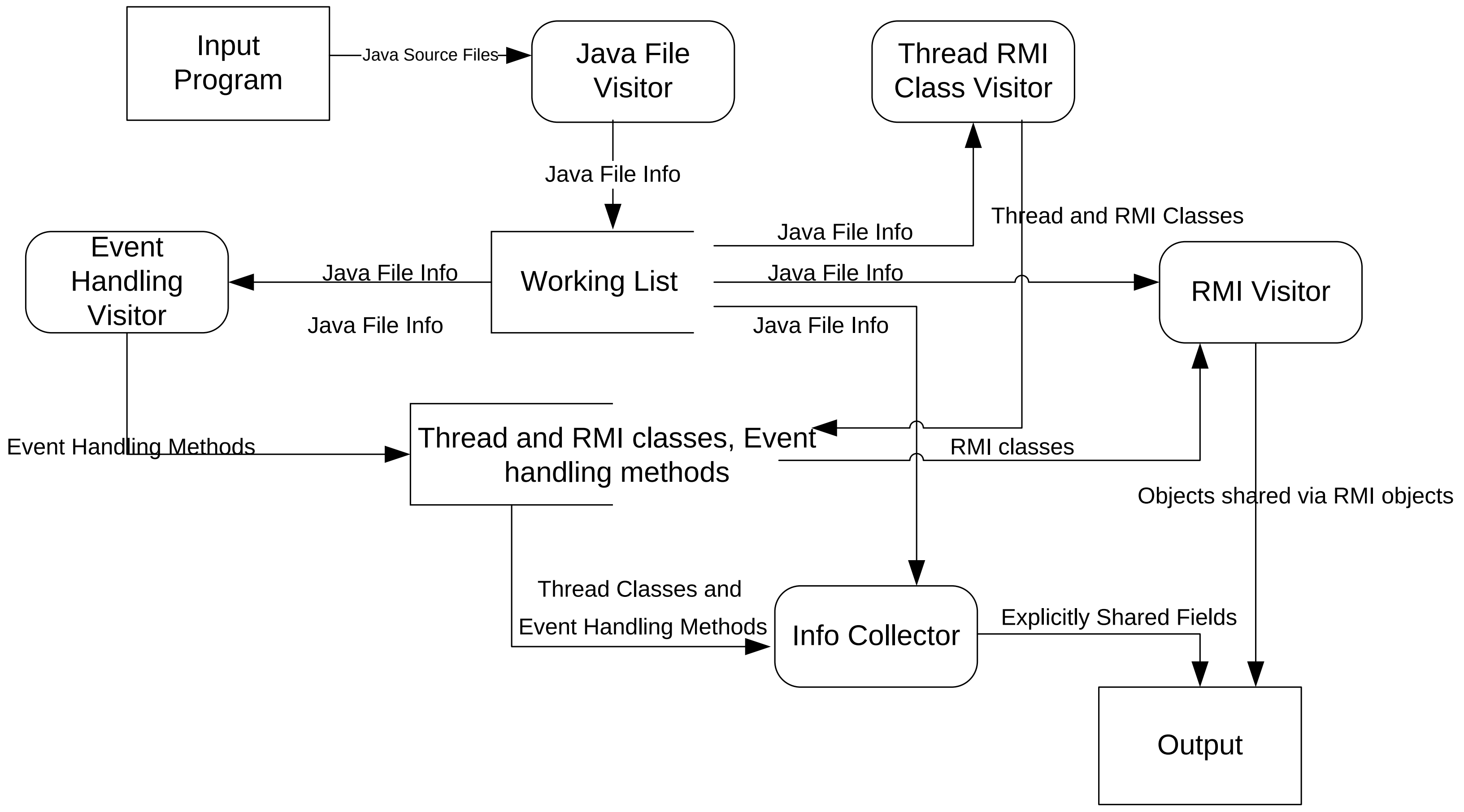


**Figure 10 Level – 0 DFD for Visitor Classes**

Below we give a description of each visitor where we state the purpose, explain how visitor collects information for his own purpose, and describe how this information is used in the analysis.

**Java File Visitor**

This visitor is used to obtain general information about the source files given as input to the tool. Gathered information includes file name, directory name, expected files and methods, file path and the rank of a file. The most important information it obtains from a file is its rank. The rank of a file is the number of different files it is dependent on. According to this rank, visitor orders syntax trees beginning from the least rank to the largest and stores it in the working list. But the ranks of classes that contain thread entry points are automatically set to zero in order to put them to the beginning of the working list and to analyze them first. DoSSO uses working list throughout all visitors. Information about expected methods is used for the purpose of correctly addressing the signature of a method, when the method call node is visited. Signature of a method is explained in section 3.2.

**Event Handling Visitor**

Event Handling Visitor is used to collect all method invocations performed by GUI framework. The results acquired from this visitor are the list of all method calls and method definitions that are executed by event dispatching thread. This visitor pays attention to method invocations and method definitions only. When it visits a node in the tree, it performs one of these two actions:

- If currently executed node is a method invocation node, then visitor looks if the method call string contains keyword `Listener`.
- If currently executed node is a method definition node, then it looks if the formal parameter type contains `Event` keyword.

If one of these checks match, then currently executed node information is added to the event method list. Info – Collector which is comprehensively explained in the next subsections uses this event method list to detect objects that are accessed or modified by event dispatching thread. Info – Collector visits each method call or method declaration node to check whether it is included in the event method list. If so, it marks the objects used inside of these nodes accordingly.

**Info – Collector**

The main work of DoSSO tool is performed by Info – Collector visitor. This visitor traverses each syntax tree in the working list, and obtains escape information about each object created and used in the code. Object escape information is obtained based on the rules stated in section 3.1.1 and is stored in OT**.** Also, this visitor stores method and object reference information in a GoM structure. DoSSO is able to use different GoMs together to obtain more specific information about objects and fields.

**RMI Visitor**

The purpose of RMI Visitor is to visit only RMI classes to find objects that are accessed and used by these classes. It traverses each element of working list, and checks whether it directly or indirectly extends the Remote interface. If the file is an implementation of a Remote interface, then RMI visitor checks whether there is a nonlocal object used inside a remote method. If so, the visitor updates its marking and object reference information accordingly. Nonlocal object is an object that is not created inside the method scope or its marking level is not GREEN. A nonlocal object can be a field, a static object, a formal parameter of a method or an object whose marking is not GREEN. This visitor reports the objects that became shared by RMI thread to the user directly.

**Thread – RMI Class Visitor**

This visitor is used to automatically find the classes that are directly or indirectly extended from the Thread class and the classes that implement the Remote or Runnable interfaces. Information obtained from this visitor is used in Info – Collector and RMI visitors.

### 4.2.1 The Analysis Flow

The DoSSO tool is scalable and is able to accept as an input a large program containing of about 200 Java files. It is fully automatic and does not need any user interference to detect RMI, explicit thread classes, and event handling methods. This section describes how DoSSO analyzes the input step by step:

1. When an input is given, firstly the tool gathers information about source codes using Java File Visitor by traversing through each syntax tree converted by the Java Parser. After each visit of a file, DoSSO puts it into a working list according to its rank. Based on rank of a file, DoSSO decides where the file should be inserted in the working list.

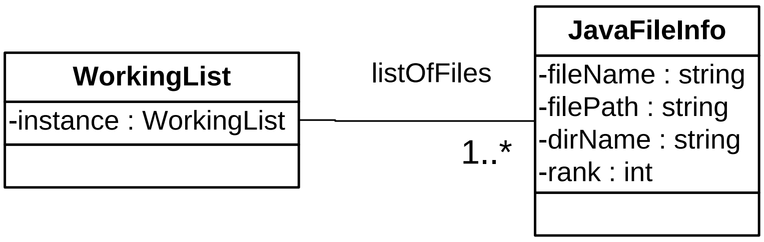


**Figure 11 Working List Structure**

2. Secondly, Thread – RMI Class Visitor operates to obtain a full list of classes which are either extended from Thread Class or implemented from Runnable or Remote interfaces. It is important to get the list of Thread classes and RMI classes beforehand, because it will be used in subsequent visitor operations.

3. Thirdly, the tool looks for method names that are called or owned by event dispatching thread. Event Handling Visitor collects the names of event handling invocations, and stores for Info – Collector to use.

4. Fourthly, the DoSSO tool searches for objects that are accessed remotely. Remote objects used in a program can be analyzed in two ways. From the view of a program which calls remote methods, objects are accessed locally. However, from the view of remote objects, objects used by RMI objects can

be accessed by more than one remote method, which means there may be a conflict in writing or reading the value of an object. From this view, it is obviously seen that each remote method acts like a separate thread, which can access the objects in a remote class any time. Thus, we say that objects used by RMI objects become shared, but inside of a program we deal with objects that are accessed by remote methods as local. Because of this, the RMI Visitor visits classes that implement Remote interface only, and collects information about objects accessed directly or indirectly in the RMI methods.

5. Lastly, the Info – Collector Visitor operates, which is the most important part in the process of finding shared objects. The purpose of Info – Collector Visitor is to find objects that have become explicitly shared between threads. The ways objects become shared are explained in section 3.1. To obtain correct results about thread escape objects, the Info – Collector Visitor makes use of information obtained from Event Handling and Thread – RMI Class Visitors. The Info – Collector Visitor visits the body and declaration of each method, and then within a Graph of Method structure it stores information about the method and detailed information about object references used in that method. Moreover it updates the `marking` and `escapesAt` information of objects in Object Table structure.

## 4.3 DoSSO Structure

### 4.3.1 DoSSO Package Diagram

DoSSO consists of 5 main packages which are UI, JTB Structure, JTB Visitors, Input Program and DoSSO Structure packages. The backbone packages that work behind the DoSSO are the JTB and DoSSO Structure packages. To execute DoSSO, programmer interacts with UI and Input Program packages only and does not touch backbone packages. Figure 12 shows how packages use each other.

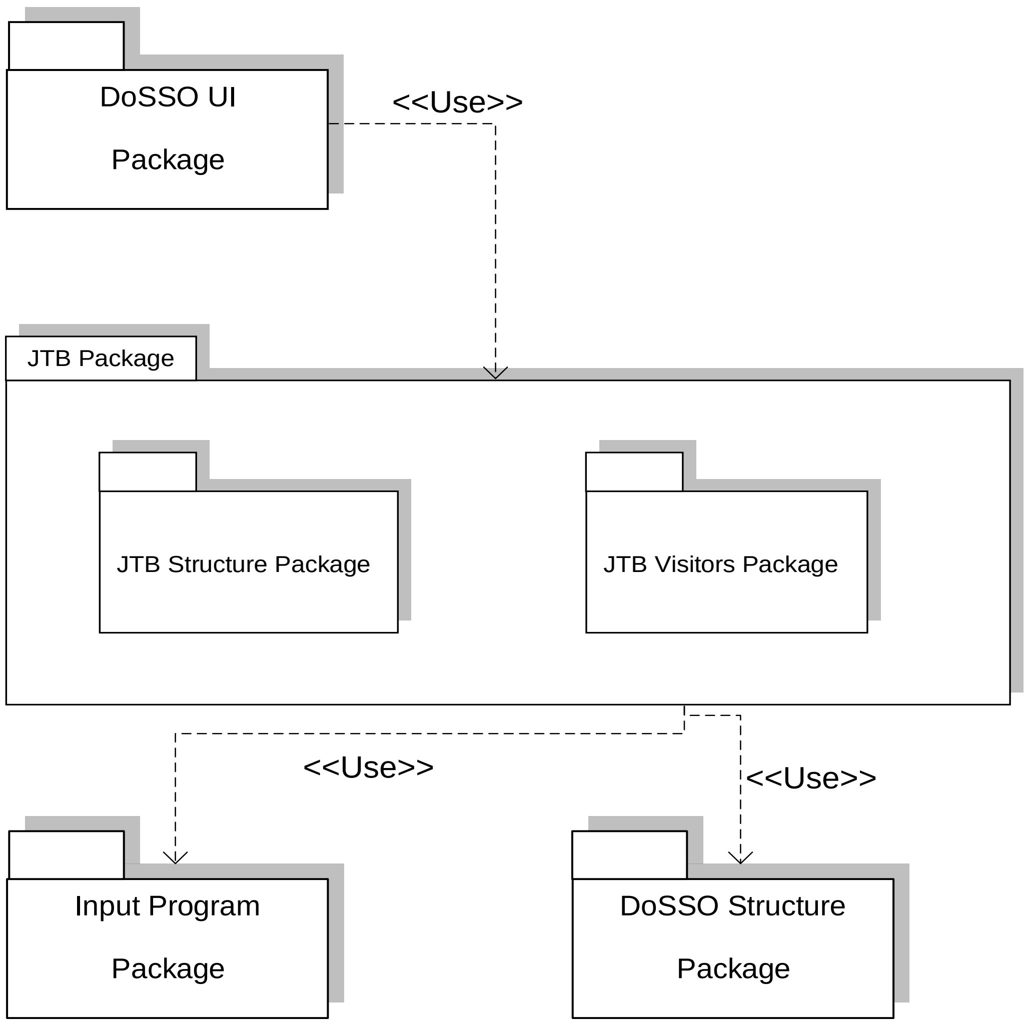


**Figure 12 DoSSO Package Diagram**

The DoSSO UI package is the package which interacts with the user to decide how and where results of execution are outputted. User is also responsible for locating the program in question into the Input Program package. UI package uses the JTB package to execute the DoSSO, whereas JTB uses both input program and DoSSO structure packages to run the DoSSO.

### 4.3.2 Data Flow Diagram for Visitors

The DFD in Figure 13 shows the flow of data obtained from each visitor, beginning from analyzing the input program until obtaining the output. Java File Visitor takes Java source files of the input program as an input data, outputs the Java file information for each source file which is then stored in the Working List structure. The data in the Working List structure is used as an input into Thread – RMI Class and Event Handling Visitors. Their outputs then are stored in “Thread and RMI classes, Event Handling methods” collection. The other visitors like Info – Collector and RMI visitors take Working List structure and “Thread and RMI classes, Event Handling methods” collection as an input. These visitors on the other hand give outputs directly to the programmer, and the outputs they produce are explicitly shared objects and objects shared via RMI objects, respectively.

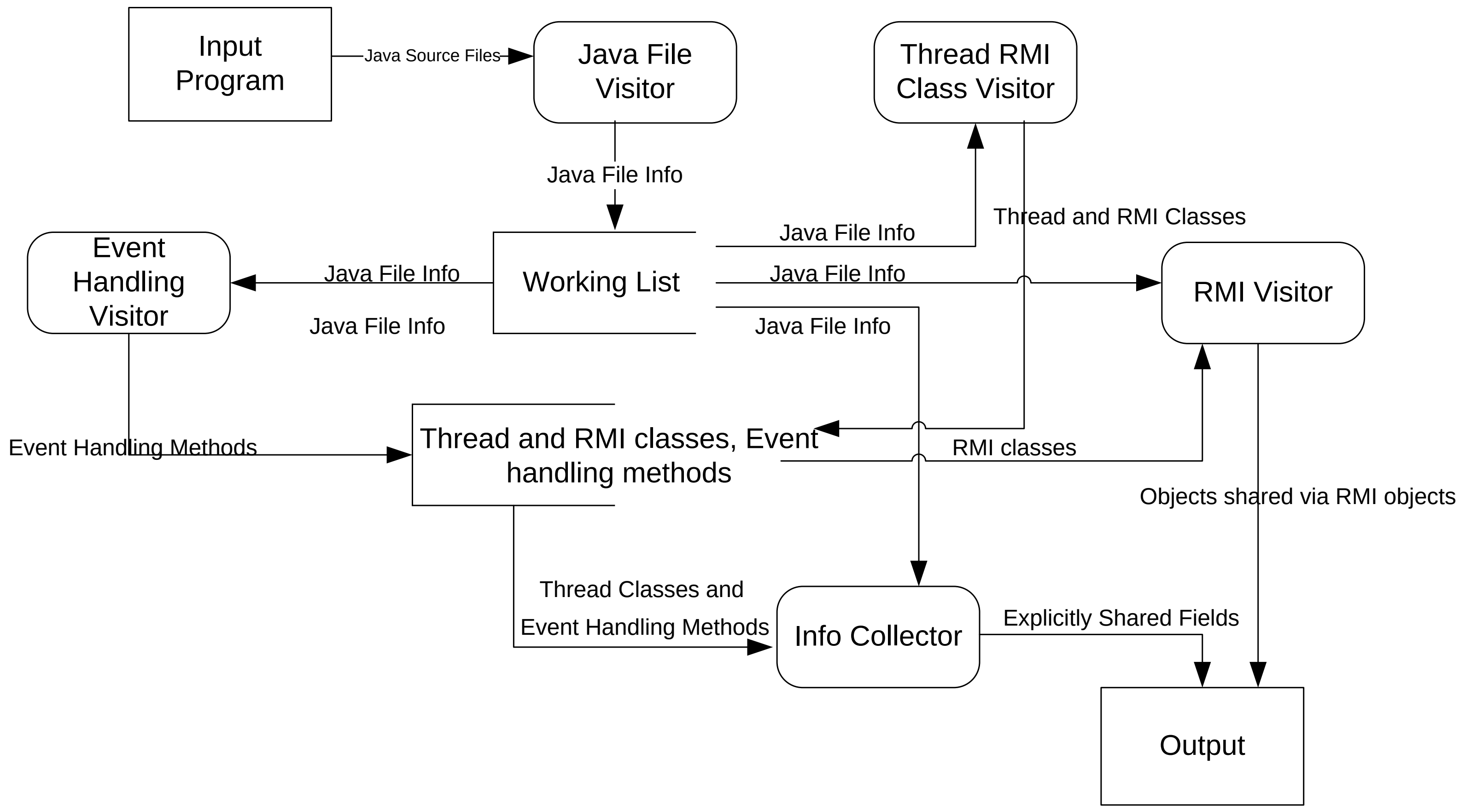


**Figure 13 Level – 0 DFD for Visitor Classes**

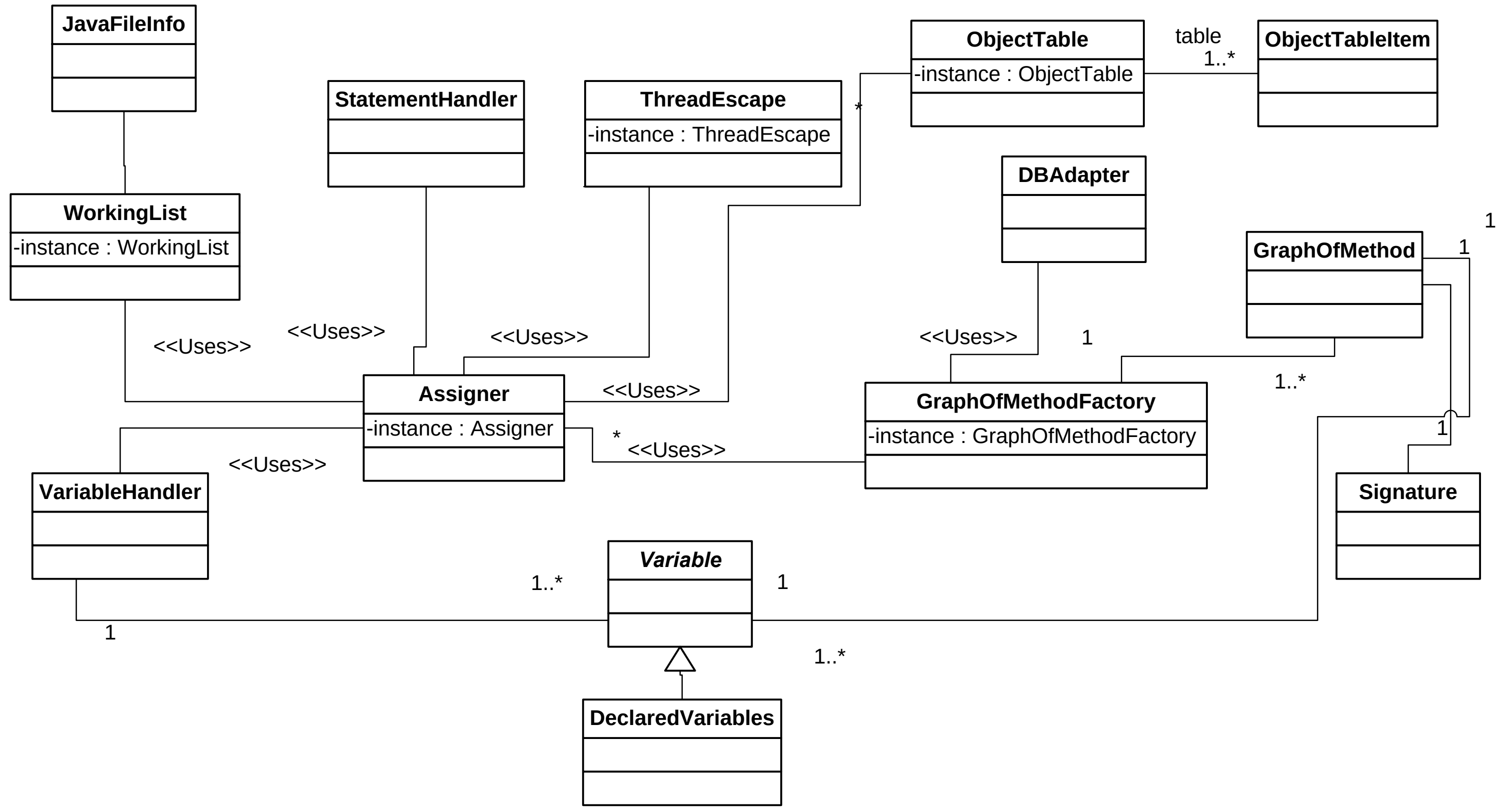


**Figure 14 Partial Class Diagram of Structure Package**

## 4.3.3 Class Diagram of Structure Package

In Figure 14, main classes of DoSSO Structure Package and their relationships are displayed. There are 5 Singleton classes in the package which are Working List, Thread Escape, Assigner, Object Table, and Graph of Method Factory. Below we will give brief description of each class shown in the diagram.

- Java File Info: When Java File Visitor visits each syntax tree given as an input, it creates an object of type Java File Info for each syntax tree and sets the fields of this object accordingly. Java File Info includes fields `fileName`, `filePath`, `dirName` and `rank` which give information about the name, the path, the directory name and the rank of a file, respectively.
- Working List: This structure contains the list of objects of type Java File Info, which is used by all visitors later on. This list shows the processing order of the source files. The files are ordered so that the file with minimum dependency to other source files is processed first. However, files containing thread entry points are processed before all other files without considering any dependency information.
- Assigner: This class is one of the major classes in the package. It is responsible for processing the data obtained from a visitor by assigning the job to either Variable Handler or Statement Handler objects. When a visitor finds useful information about objects such as assignment of two objects, method call and argument passing, it sends a collection of strings containing this useful information to the Assigner object to decide which handler should process the information. Moreover, Assigner object converts the obtained strings into objects and keeps them for later use.
- Statement Handler: This handler class is the class that deals with processing statement strings such as assignment and method call. Statement handler takes a string, converts it into objects, processes them based on which operation was called (the operation is either Handle Assignments or Handle Method Call) and stores information obtained from objects into the current GoM.

- Variable Handler: This class is responsible for storing all variables used inside a syntax tree. When there is an assignment statement or an argument passing expression, the Statement Handler object sends a string to the Variable Handler object to identify which object is being assigned now or which object is passing as an argument to a method call. Based on the result of the Variable Handler, DoSSO is able to get correct information about objects and their escape information.
- Object Table and Object Table Item: These structures are explained in section 3.3.
- Thread Escape: This structure stores the list of classes that directly or indirectly extend the Thread class, implement Runnable or Remote interfaces. It also stores the list of event thread methods.
- Variable: DoSSO stores objects used inside the syntax trees in terms of Variable objects. The structure of Variable class is as follows:

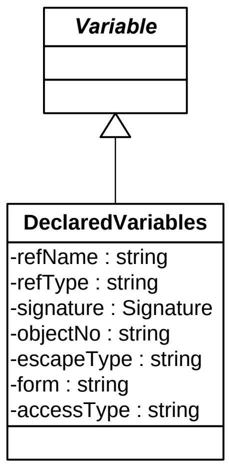


**Figure 15 Structure of Variable Class**

The Declared Variables class is used in all visitors to represent the objects used in the input program.

- Signature: This structure stores information about where either GoM or an object was created by giving the package name, class name, method name, types and number of formal parameters of a method.
- Graph Of Method: This structure is comprehensively explained in section 3.2

- Graph of Method Factory: The factory includes the list of all GoMs that were created during the analysis of the input program. When visitor needs a specific GoM, it extracts the Signature of the GoM and asks it from the factory. Factory checks whether there is a GoM matching with the Signature. If it matches, factory returns the GoM found; if not, factory creates a new GoM with the Signature and returns a newly created GoM.
- DBAdapter: This adapter is the class which is responsible for storing already analyzed GoMs into a persistent storage and for retrieving these GoMs for reuse.

# CHAPTER 5

# EXPERIMENTS AND CASE STUDY

To demonstrate the applicability of DoSSO, we have conducted several small experiments and a case study on a distributed and concurrent system with graphical user interface. In this chapter we first give results from experiments, then describe the structure of the concurrent system called Concurrent Editor on which the case study was performed. Next, we continue with explaining the steps undertaken to conduct the case study, and present the analysis results of DoSSO on the Concurrent Editor.

## 5.1 Experimental Results

Before conducting the case study, we experimented our tool on 15 various small input programs. Since these experiments were small, we were able to check whether DoSSO outputted correct and complete results or not. Before testing DoSSO on each experiment, we extracted the red and yellow objects manually. After each testing we compared DoSSO results with our expectations to validate the results. Experiments showed that DoSSO is able to give correct and complete results.

**Table 1 DoSSO Results obtained from Experiments**

| Experiment# | Lines Of Code | #REDActual | #REDDetectedByDoSSO |
|---|---|---|---|
| Experiment1 | 40 | 2 | 3 |
| Experiment2 | 49 | 3 | 4 |
| Experiment3 | 38 | 2 | 2 |
| Experiment4 | 48 | 3 | 3 |
| Experiment5 | 63 | 3 | 4 |
| Experiment6 | 74 | 3 | 3 |
| Experiment7 | 227 | 8 | 9 |
| Experiment8 | 142 | 4 | 5 |
| Experiment9 | 177 | 5 | 8 |
| Experiment10 | 175 | 5 | 7 |
| Experiment11 | 124 | 4 | 4 |
| Experiment12 | 350 | 10 | 10 |
| Experiment13 | 92 | 3 | 4 |
| Experiment14 | 158 | 5 | 6 |
| Experiment15 | 113 | 4 | 4 |

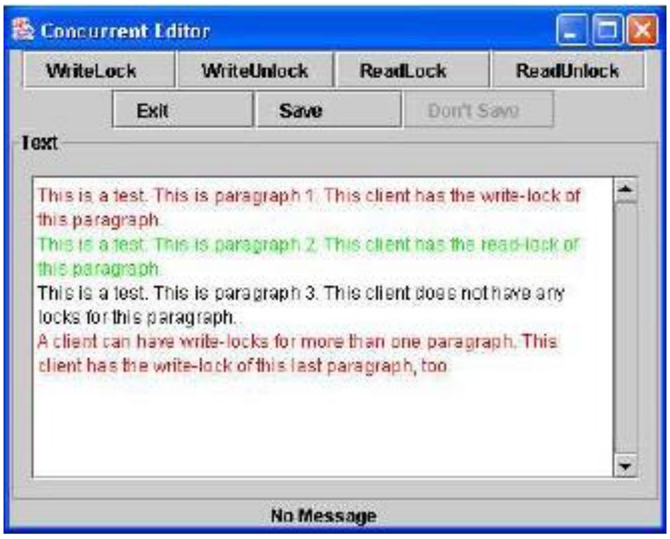
Table 1 shows the list DoSSO results obtained from testing 15 experimental input programs. The column “#REDActual” is the number of objects that we expected will be certainly RED at the end of analysis. Whereas, the column “#REDDetectedByDoSSO” is the number of objects that DoSSO reported as RED. The results in Table 1 show that DoSSO does not report the number of objects detected as RED less than the actual number of RED objects, meaning that DoSSO may report false positive. In these experiments we observed that DoSSO did not report false negative. For example in Experiment 12, we knew that 10 objects are risky shared objects and expected that these objects will be marked as RED. After the analysis, DoSSO successfully identified these 10 objects as RED.

However, we could not verify the results of the case study in the same way as with experiments because the case study program was more complicated than our simple experimental programs, and it was hard to obtain all list of objects that we thought would be definitely shared in the program. On the case study we checked whether the red objects reported are actually red instead and relied on previous experiments' results that DoSSO did not miss a red object.

## 5.2 Concurrent Editor

Concurrent Editor is a distributed text editor. The editor enables multiple users to edit a document that resides in a server at the same time. The document has to be consistent at all times. Whenever a client changes the document, the change is reflected on all clients in real time. After the editing, the document is saved if a consensus is reached among all clients.

Concurrent Editor is a real complicated application containing remote method invocations and accesses to shared data concurrently. It consists of 2800 lines of Java code. The application is composed of server and client nodes. The server node contains the original document, whereas client nodes contain the copies of the original document. It is possible for the document to be accessed and modified concurrently by client nodes. Figure 16 is the screenshot of the application [23].



This is a test. This is paragraph 1. This client has the write-lock of this paragraph.
This is a test. This is paragraph 2. This client has the read-lock of this paragraph.
This is a test. This is paragraph 3. This client does not have any locks for this paragraph.
A client can have write-locks for more than one paragraph. This client has the write-lock of this last paragraph, too.

**Figure 16 Concurrent Editor Screen Shot**

In Figure 16, there are several control buttons and a document with several paragraphs. As stated earlier, client nodes are able to access paragraphs concurrently. In order to do so, they need to send a request for either write or read access to the paragraph by clicking the `WriteLock` or `ReadLock` button, respectively. When a lock request is granted, the color of the paragraph changes and shows that now it is editable. Figure 16 shows that red color indicates a write lock permission, yellow indicates a read lock permission, and black indicates no access permission to a paragraph. A client node cannot modify the paragraph the cursor is on unless she has a write access to that paragraph. Multiple client nodes are not able to modify paragraphs at the same time unless they modify different paragraphs. If a client node has a read access to a paragraph, no other client may have a write access to that paragraph unless all read accesses to that paragraph are released. If a client node has a write access to a paragraph, no other client may have either write or read access to that paragraph unless the write access is released. To release write or read accesses, client nodes need to click `WriteUnlock` or `ReadUnlock` buttons, respectively.

## 5.3 Case Study Steps

In this section we describe the steps we have taken to conduct the case study.

**Step 1 – Preparation of the input program**

To prepare the input program means to collect the source codes of all method invocation and initiation expressions used inside the input program. For example, the Concurrent Editor package consists of 43 Java class files and they include many different types of objects such as Vector, Integer, HashMap, SWING components. In order to run DoSSO successfully, we needed source codes of classes that were used inside of 43 Java source files and that were not included in our permanent storage. After collecting all source codes, we ended up with 192 of Java source files. Nevertheless, a programmer may write his/her own stub classes for source files instead of collecting them. If programmer uses stub classes, then s/he should make sure that stub classes implement methods used in the source code. In our case, some classes inside of 192 were written as stub classes.

**Step 2 – Preprocessing the input program**

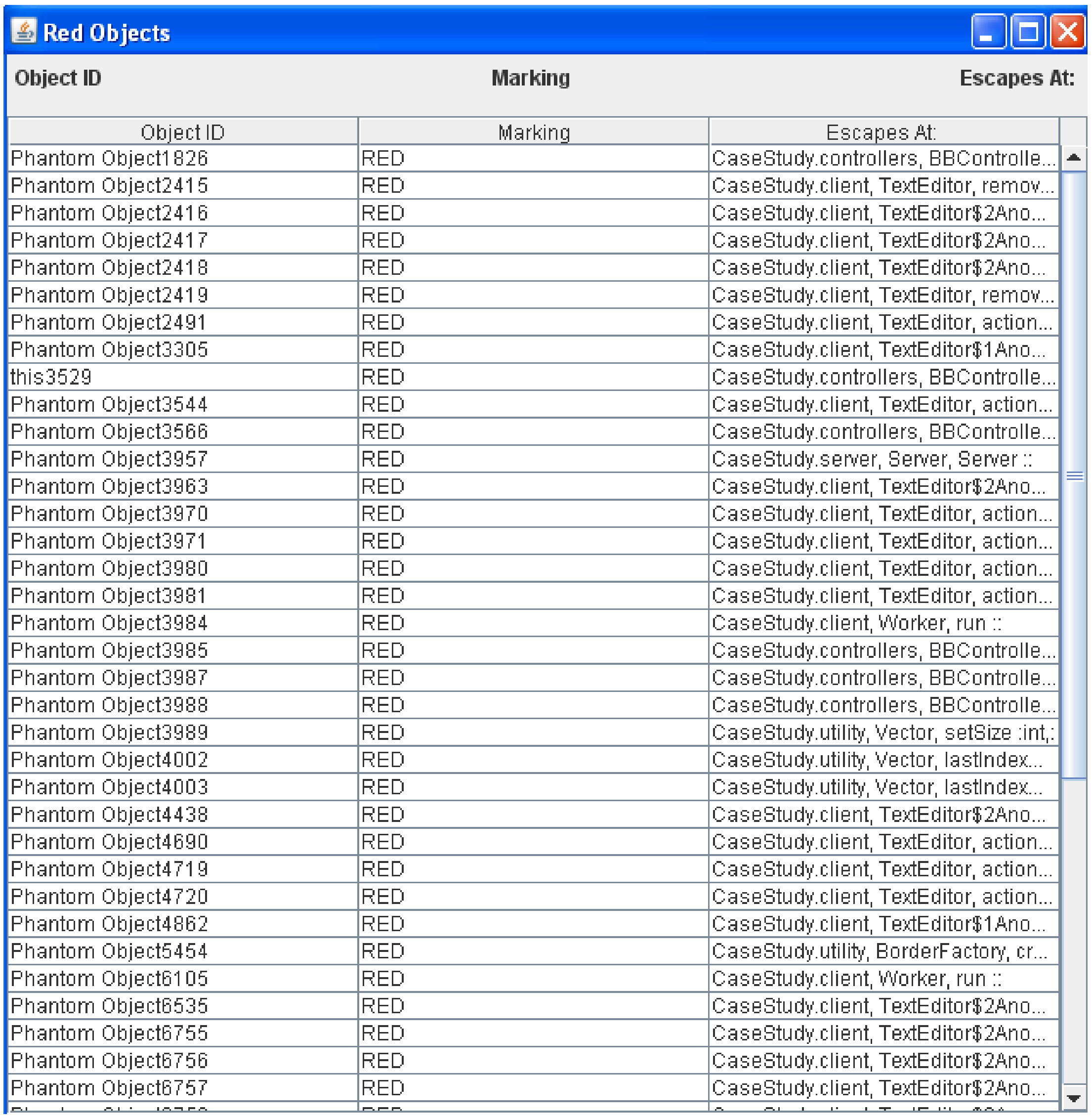
DoSSO requires the input program to be written in two level reference expressions. In this step we used Soot framework [22] to convert the Concurrent Editor implementation into a program with two level reference expressions.

**Step 3 – Executing DoSSO**

We executed the case study on the machine containing RAM of 3GB space, with CPU speed of 1.66GHz and it took about 3 minutes for DoSSO to run.

**Step 4 – Evaluating the results**

At the end of the analysis, DoSSO gives two different outputs. The first output is the RedObjects window which contains the list of objects that became shared in the program. This window outputs only risky shared objects. RedObjects window shown in Figure 17 includes information about object id, markings and the signature of the method where the object escapes. However, this single output is not sufficient to evaluate the results.



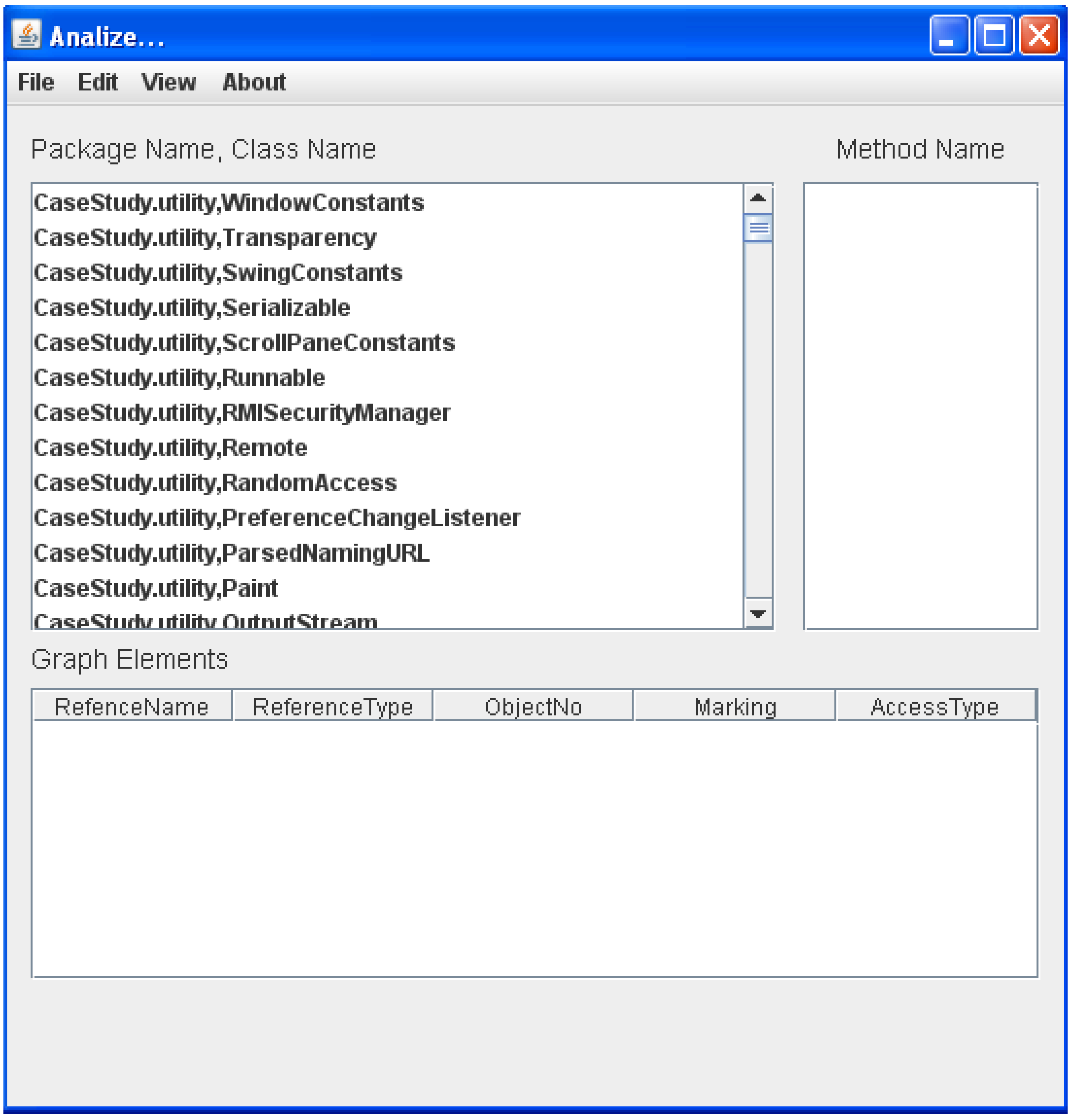
Red Objects

Object ID Marking Escapes At:

| Object ID | Marking | Escapes At: |
|---|---|---|
| Phantom Object1826 | RED | CaseStudy.controllers, BBControlle... |
| Phantom Object2415 | RED | CaseStudy.client, TextEditor, remov... |
| Phantom Object2416 | RED | CaseStudy.client, TextEditor$2Ano... |
| Phantom Object2417 | RED | CaseStudy.client, TextEditor$2Ano... |
| Phantom Object2418 | RED | CaseStudy.client, TextEditor$2Ano... |
| Phantom Object2419 | RED | CaseStudy.client, TextEditor, remov... |
| Phantom Object2491 | RED | CaseStudy.client, TextEditor, action... |
| Phantom Object3305 | RED | CaseStudy.client, TextEditor$1Ano... |
| this3529 | RED | CaseStudy.controllers, BBControlle... |
| Phantom Object3544 | RED | CaseStudy.client, TextEditor, action... |
| Phantom Object3566 | RED | CaseStudy.controllers, BBControlle... |
| Phantom Object3957 | RED | CaseStudy.server, Server, Server :: |
| Phantom Object3963 | RED | CaseStudy.client, TextEditor$2Ano... |
| Phantom Object3970 | RED | CaseStudy.client, TextEditor, action... |
| Phantom Object3971 | RED | CaseStudy.client, TextEditor, action... |
| Phantom Object3980 | RED | CaseStudy.client, TextEditor, action... |
| Phantom Object3981 | RED | CaseStudy.client, TextEditor, action... |
| Phantom Object3984 | RED | CaseStudy.client, Worker, run :: |
| Phantom Object3985 | RED | CaseStudy.controllers, BBControlle... |
| Phantom Object3987 | RED | CaseStudy.controllers, BBControlle... |
| Phantom Object3988 | RED | CaseStudy.controllers, BBControlle... |
| Phantom Object3989 | RED | CaseStudy.utility, Vector, setSize :int,: |
| Phantom Object4002 | RED | CaseStudy.utility, Vector, lastIndex... |
| Phantom Object4003 | RED | CaseStudy.utility, Vector, lastIndex... |
| Phantom Object4438 | RED | CaseStudy.client, TextEditor$2Ano... |
| Phantom Object4690 | RED | CaseStudy.client, TextEditor, action... |
| Phantom Object4719 | RED | CaseStudy.client, TextEditor, action... |
| Phantom Object4720 | RED | CaseStudy.client, TextEditor, action... |
| Phantom Object4862 | RED | CaseStudy.client, TextEditor$1Ano... |
| Phantom Object5454 | RED | CaseStudy.utility, BorderFactory, cr... |
| Phantom Object6105 | RED | CaseStudy.client, Worker, run :: |
| Phantom Object6535 | RED | CaseStudy.client, TextEditor$2Ano... |
| Phantom Object6755 | RED | CaseStudy.client, TextEditor$2Ano... |
| Phantom Object6756 | RED | CaseStudy.client, TextEditor$2Ano... |
| Phantom Object6757 | RED | CaseStudy.client, TextEditor$2Ano... |

**Figure 17 Red Objects Window**

The second output is a GoMs window which contains names of each class analyzed by DoSSO. For example, Figure 18 shows the package and class names of files analyzed by DoSSO on the left side of the GoMs window.

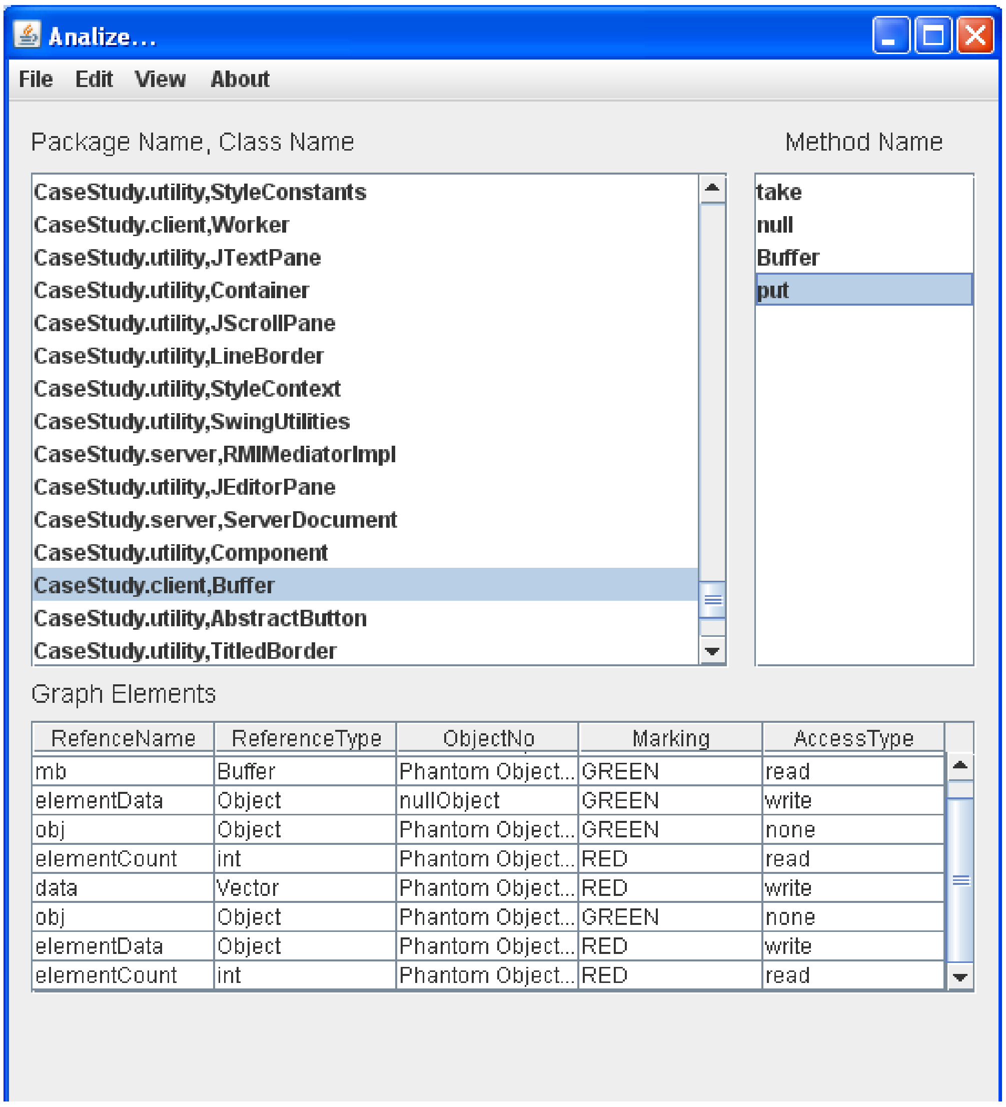


CaseStudy.utility,WindowConstants
CaseStudy.utility,Transparency
CaseStudy.utility,SwingConstants
CaseStudy.utility,Serializable
CaseStudy.utility,ScrollPaneConstants
CaseStudy.utility,Runnable
CaseStudy.utility,RMISecurityManager
CaseStudy.utility,Remote
CaseStudy.utility,RandomAccess
CaseStudy.utility,PreferenceChangeListener
CaseStudy.utility,ParsedNamingURL
CaseStudy.utility,Paint
CaseStudy.utility,OutputStream

| RefenceName | ReferenceType | ObjectNo | Marking | AccessType |
|---|---|---|---|---|

**Figure 18 GoMs Window – Output of Analyzed Classes**

Each analyzed class shown in Figure 18 contains at least one GoM information, depending on the number of methods the class contains. Meanwhile each GoM contains information about object references created and accessed inside the GoM. For example, Figure 19 shows GoMs of `Buffer` class, and Graph Elements inside the `put` method.



| Package Name, Class Name |
|---|
| CaseStudy.utility,StyleConstants |
| CaseStudy.client,Worker |
| CaseStudy.utility,JTextPane |
| CaseStudy.utility,Container |
| CaseStudy.utility,JScrollPane |
| CaseStudy.utility,LineBorder |
| CaseStudy.utility,StyleContext |
| CaseStudy.utility,SwingUtilities |
| CaseStudy.server,RMIMediatorImpl |
| CaseStudy.utility,JEditorPane |
| CaseStudy.server,ServerDocument |
| CaseStudy.utility,Component |
| CaseStudy.client,Buffer |
| CaseStudy.utility,AbstractButton |
| CaseStudy.utility,TitledBorder |

| Method Name |
|---|
| take |
| null |
| Buffer |
| put |

| RefenceName | ReferenceType | ObjectNo | Marking | AccessType |
|---|---|---|---|---|
| mb | Buffer | Phantom Object... | GREEN | read |
| elementData | Object | nullObject | GREEN | write |
| obj | Object | Phantom Object... | GREEN | none |
| elementCount | int | Phantom Object... | RED | read |
| data | Vector | Phantom Object... | RED | write |
| obj | Object | Phantom Object... | GREEN | none |
| elementData | Object | Phantom Object... | RED | write |
| elementCount | int | Phantom Object... | RED | read |

**Figure 19 GoMs Window – Detailed View of Analyzed Class**

The first row under the "Graph Elements" label, i.e. the headings of the table, in Figure 19 states the order of the properties of Graph Elements to be shown.

To evaluate the results, a programmer should use both of the outputs. Firstly, programmer chooses the objectID from the RedObjects window, and gets the signature of the method where the object became shared. Then from this signature, programmer extracts the class name and finds the class name in the GoMs window. Programmer also extracts the method name from the signature in order to look for the GoM with the same method name inside the class. Afterwards, programmer examines the GoM of the method reported in the RedObjects window. In this GoM, programmer is able to see the object references which refer to the objectID. These references are the variables that have to be protected with a synchronization mechanism.

The analysis results are stored in text files when DoSSO stops running. So, it is also possible to look for these text files separately. The list of red objects shown in Red Objects window is stored in Red Objects text file. The list of GoMs and their Graph Elements shown in GoMs window are stored in GoMs folder, where each class is stored in a separate text file. Sample screen shots of Red Objects file and GoMs folder are given in Appendix B.

## 5.4 Case Study Results

Before discussing the case study results obtained from DoSSO, we want to highlight one important point that we had observed from our analysis of case study results. Recall that our input program consisted of about 192 java source files, 43 of which were Concurrent Editor’s class files. The remaining 149 source files were either collected from Java libraries or were written as stub classes. The important point we had observed was that DoSSO results are dependent on how precise the input program is with respect to objects used inside the program, i.e. the more information programmer gives in the input program, and the more accurate results are produced by DoSSO. Because of this reason, we think that if our input program were given in more detailed way, we could obtain new list of shared objects in addition to our current findings. By “more detailed way” we mean that if we used original source code instead of stub classes when were preparing the input program, we could obtain more exact results.

The another important point is that when the input program contains stub classes, and DoSSO outputs inadequate number of shared objects, this situation does not mean that DoSSO reported false negative. It is possible that in the original code, the object becomes shared but since stub classes do not reflect the original source code in terms of implementation, certainly DoSSO will not able to detect this object, and will report it as GREEN. But if stub classes reflect the original implementation, then DoSSO detects shared objects successfully.

DoSSO has identified 53 objects as explicitly shared risky objects. In addition, DoSSO has detected 23 objects as shared due to RMI objects in the program and all of them were detected in the server side of the application. In Appendix A, we give the total list of objects that DoSSO concluded as risky shared objects and that are stored in RedObjects file. If the developer of Concurrent Editor identified shared objects manually, most probably the developer would miss some of objects among the 76 detected ones, and would not protect them with a locking mechanism. Then, not protecting all shared objects would cause problems after the application is developed. As the size of concurrent application grows the effort of identifying shared objects also grows. If the size of an application is huge, then it is very hard to manually identify all shared objects in the application as in our case study. Because of this fact, it will be advantageous for developer to use our tool to identify all shared objects automatically.

The objects DoSSO outputted as shared at the end of the analysis were the objects that were marked as RED during the analysis. However, at the end of the analysis we could see that there were too many objects (337 objects to be exact) marked as YELLOW. Programmer can check those YELLOW objects from files in the GoMs folder which contain objects used inside GoM, both shared and nonshared.

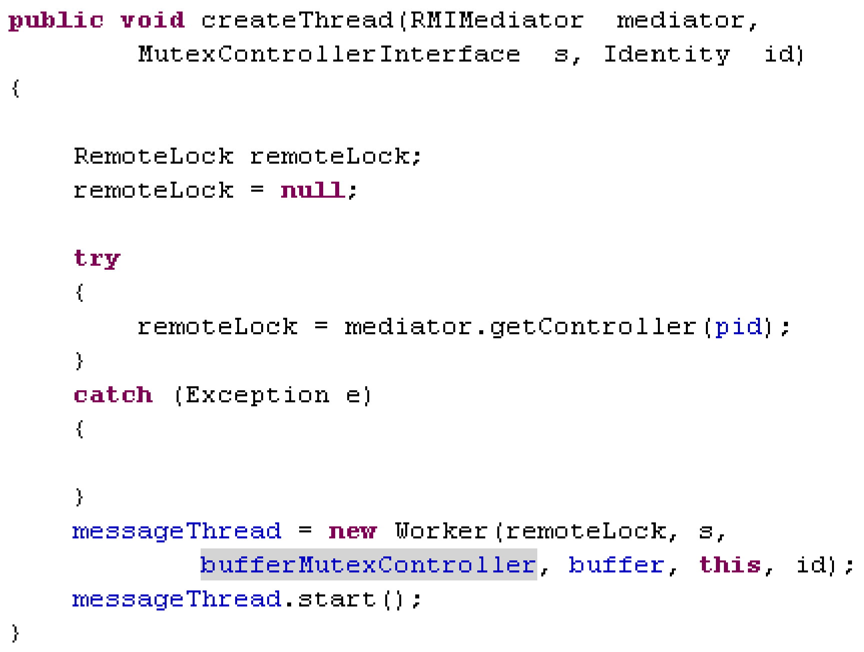
### 5.4.1 Observations on Case Study Results

In the rest of this section, we want to show how actually DoSSO concluded that the object is shared and marked the object as RED. For this reason, we will show three examples from the case study which will help to understand the steps DoSSO carried out to decide on the escape information of an object in detail.

- Observation – 1

We want to understand how PhantomObject 6888 in Appendix A became shared in the input program. The output in Appendix A shows that this object became shared at signature `CaseStudy.client, LockAttribute, createThread :Identity.` First we have to find the GoM in the GoMs window containing the same signature as above in order to find exactly which object reference became shared. The way to find the specific GoM in GoMs window is explained in section 5.2 thoroughly.

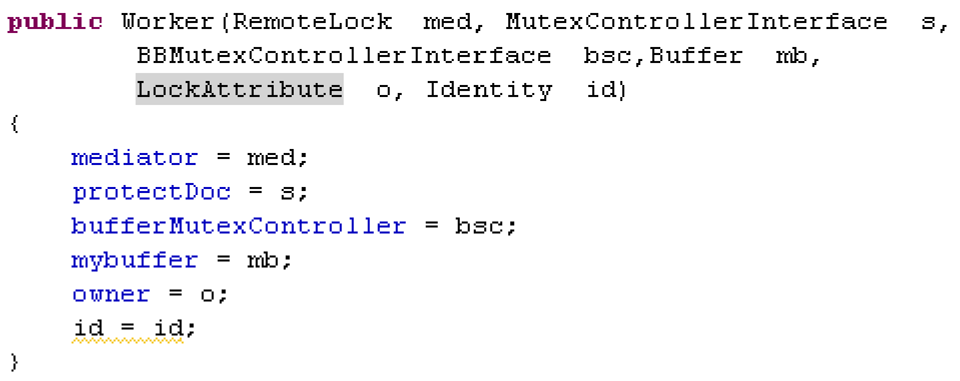
Figure 20 shows a code of the `createThread` method of the `LockAttribute` class. In the code, the new Thread object is created and 6 objects pass as an argument to the construction of this thread object. Right after the creation, the `messageThread` object starts. Since these 6 object which are `remoteLock`, `s`, `bufferMutexController`, `buffer`, `this` and `id` pass to another thread scope, they become shared and the objects they refer are marked as YELLOW in OT.



```
public void createThread(RMIMediator  mediator,
        MutexControllerInterface  s, Identity  id)
{

    RemoteLock remoteLock;
    remoteLock = null;

    try
    {
        remoteLock = mediator.getController(pid);
    }
    catch (Exception e)
    {

    }
    messageThread = new Worker(remoteLock, s,
            bufferMutexController, buffer, this, id);
    messageThread.start();
}
```

**Figure 20 The code of createThread method in LockAttribute class**

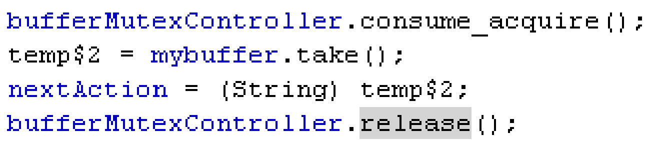
Meanwhile, in the constructor of Worker class, which is shown in Figure 21, we see that fields of the thread class are set to formal parameter values. Therefore when messageThread is instantiated in Figure 20, the `mediator`, `protectdoc`, `bufferMutexController`, `mybuffer`, `owner` and `id` fields will refer to the same object as `remoteLock`, `s`, `bufferMutexController`, `buffer`, `this` and `id`, respectively. Since they refer to the same object, at this point these fields become shared as well.



```
public Worker(RemoteLock  med, MutexControllerInterface  s,
        BBMutexControllerInterface  bsc,Buffer  mb,
        LockAttribute  o, Identity  id)
{
    mediator = med;
    protectDoc = s;
    bufferMutexController = bsc;
    mybuffer = mb;
    owner = o;
    id = id;
}
```

**Figure 21 The code of the constructor method of Worker class**

Right after the instantiation of the `messageThread` object, the `messageThread` starts. Figure 22 shows segment of code taken from the `run` method of Worker class. From this code segment, DoSSO detects that the `take` method is a setter method, and concludes that the state of `mybuffer` field is changed. At this line, DoSSO finds the object `mybuffer` refers to in OT, and if the marking of the object is not GREEN, DoSSO changes its marking to RED. In fact, from the GoM information of Worker Fields Graph, we found that `mybuffer` refers to the object Phantom Object 6888 which means DoSSO detected this object correctly.

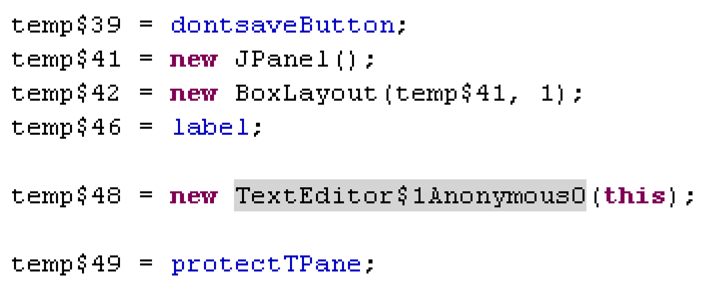


```
bufferMutexController.consume_acquire();
temp$2 = mybuffer.take();
nextAction = (String) temp$2;
bufferMutexController.release();
```

**Figure 22 Code segment of run method in Worker class**

- Observation – 2

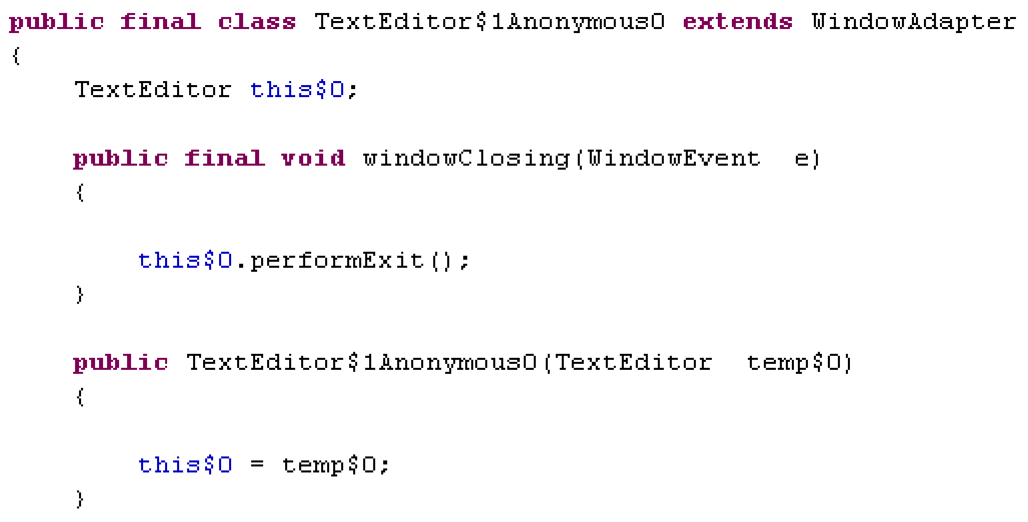
We want to find how PhantomObject 7232 in Appendix A became shared in the input program. The output in Appendix A shows that this object became shared at signature `CaseStudy.client, TextEditor, initGraphics: Vector`. From the GoM information of signature `CaseStudy.client, TextEditor, initGraphics: Vector` we found that `this$0` object reference of type `TextEditor` refers to PhantomObject 7232. Using this information we understand that `this` object in Figure 23 passed as an argument to the constructor method of `TextEditor$1Anonymous0` and the field `this$0` is set to refer to the same object as `this` refers to. The constructor implementation of class `TextEditor$1Anonymous0` is given in Figure 24.



```
temp$39 = dontsaveButton;
temp$41 = new JPanel();
temp$42 = new BoxLayout(temp$41, 1);
temp$46 = label;

temp$48 = new TextEditor$1Anonymous0(this);

temp$49 = protectTPane;
```

**Figure 23 Code segment of initGraphics method in TextEditor class**

When `this` object passes as an argument to the constructor of `TextEditor$1Anonymous0` object, `this` object becomes shared and marked as YELLOW because it passes to one of the event dispatch thread's method. Thus, the object `this` refers to is marked as YELLOW. In Figure 24, there is `windowClosing` method, where the state of `this$0` field changes because one of its setter methods – `performExit` is called. Because of this method, object that `this$0` refers to now becomes RED. In this way, Phantom Object 7232 is concluded as RED in OT.

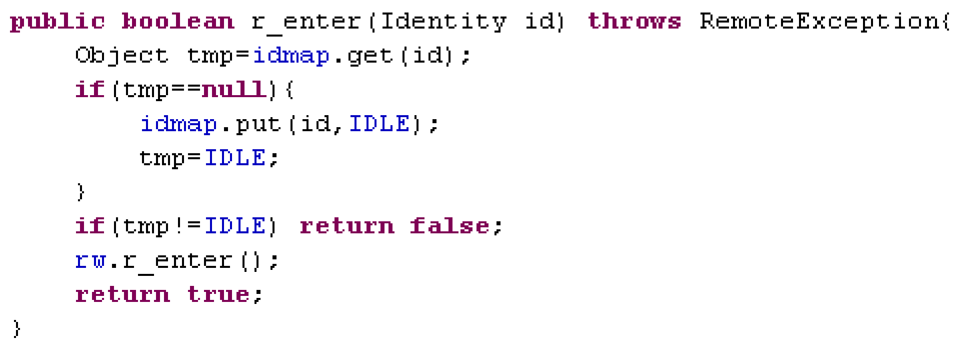


```
public final class TextEditor$1Anonymous0 extends WindowAdapter
{
    TextEditor this$0;

    public final void windowClosing(WindowEvent  e)
    {

        this$0.performExit();
    }

    public TextEditor$1Anonymous0(TextEditor  temp$0)
    {

        this$0 = temp$0;
    }
```

**Figure 24 Segment of code of TextEditor$1Anonymous0 class**

- Observation – 3

In this part, we show how DoSSO decides escape information of objects used inside RMI methods, which are considered as thread entry points in the analysis. For example consider the code in Figure 25.



```
public boolean r_enter(Identity id) throws RemoteException{
    Object tmp=idmap.get(id);
    if(tmp==null){
        idmap.put(id,IDLE);
        tmp=IDLE;
    }
    if(tmp!=IDLE) return false;
    rw.r_enter();
    return true;
}
```

**Figure 25 RMI method r_enter of RemoteLockImpl class**

In Figure 25 the remote method `r_enter` of `RemoteLockImpl` class is shown. DoSSO considers this method as explicit thread. In the method, `idmap`, `IDLE` and `rw` variables are fields of the class. `IDLE` and `rw` are only accessed but not modified by the remote method, thus DoSSO marks objects they refer to as

YELLOW. Whereas, `idmap` is modified at line `idmap.put(id, IDLE)` since `put` method is a setter method meaning that it sets one of the fields of `idmap`. At this line, DoSSO marks the object `idmap` refers as RED in OT.

# CHAPTER 6

# CONCLUSION and FUTURE WORK

In this section, firstly we will mention the limitations and advantages of DoSSO. Then we will give a conclusive summary of our work. Lastly we will state possible future work related with the DoSSO.

DoSSO successfully determines shared objects in the input program. However, as all other tools, it also has got its own advantages and limitations.

## 6.1 Limitations

- The first and the most important limitation of DoSSO is that it requires the source codes of all methods used inside the input program. The analysis expects all GoMs to be complete in terms of dependencies to other graphs. Requiring all source code might be a problem. For this reason we provide a storing utility for saving already analyzed GoMs. However, DoSSO still asks the programmer to carefully check all the source code given as an input.

- DoSSO does not handle recursive functions. Recursive function is a function in which it makes a call to itself.

- Another problem that DoSSO does not handle is an interface implementation and polymorphism problem. Since DoSSO performs a static analysis, it cannot distinguish among the types that implemented the same interface or that extended the same class; this recognition can be done only at run time.

- DoSSO handles only two level reference expressions. If the input program contains codes similar to ones given below, they have to be converted. We do this conversion through Soot [22] which is explained in section 4.1 in more detail.
    - `a.b.c.d();` - DoSSO accepts only two level reference expressions, as stated in chapter 3, meaning that programmer should ensure that an input program is free of multiple level expressions.
    - `foo(goo());` - variables passed as an argument to a method call have to be simple, i.e. programmer has to avoid using complex expressions like method call, assignment operation etc.
    - `new Thread();` - DoSSO stores objects with their reference names, hence it is essential for all objects to have a reference name at the initiation.

## 6.2 Advantages

- DoSSO is fully automated. It does not involve user interaction or interference while performing the analysis.

- DoSSO is able to detect shared objects due to RMI objects and event handling methods. To our best knowledge, none of previous works handle these two issues while detecting shared objects.

- Another advantage is that DoSSO supports the design for verification paradigm [11]. Based on the objects that DoSSO reported as potential shared objects, programmer may design his/her concurrent program accordingly, and may implement its design so that verification becomes easier later on.

- DoSSO saves the effort of developer in considering concurrency issues by providing shared objects before the program is developed. Instead of dealing with issues such as race condition after the program is developed, we suggest the prevention of such faults by reporting the risky objects.

## 6.3 Conclusion

Developing concurrent programs is hard, because programmer has to carefully identify shared objects and somehow protect them from being hazardously used. Identification of shared objects has come to be performed manually, which is an error prone process. We introduce a new tool which would resolve the problem - DoSSO, a fully automated tool that detects shared objects in the program. We believe that identifying all risky shared objects at the beginning of the development of a program eases the design and implementation of the system, and decreases the cost of programmer labor force.

We propose a new tool which performs a static analysis to find objects that become shared. We detect objects that are shared 1) when explicitly used together with Thread or Runnable objects, 2) when used jointly with RMI objects, and 3) when accessed inside the event handling methods. Our analysis performs five passes over the input program to detect shared objects. The abstract syntax tree of the input program obtained from the source code is traversed by five different visitors to obtain the following information: Java File information, Thread and RMI objects, event handling methods, shared objects due to RMI objects and explicit thread objects. We developed a new structure to store information obtained from each method definition node – GoM. GoMs store information about object references created and accessed inside the method scope. Escape information of objects are stored in Object Table (OT) structure. Based on the information stored inside the OT, DoSSO reports the list of shared objects in the program.

Based on our case study results, the presented tool, DoSSO, successfully identified shared objects automatically. In this way, we believe that it is a promising tool to aid the programmers in developing concurrent programs.

## 6.4 Future Work

In an object oriented program there is a hidden dependency between the methods of an object. A field of an object is updated in a method while it used in another method and these methods can be accessed by different threads. We call these fields as hidden shared fields. These hidden shared objects can be detected by analyzing relationships between getter and setter methods of an object when they are called in different thread scopes. In the future, an algorithm for finding hidden shared fields could be developed and integrated into DoSSO analysis.

Another future work is handling recursive functions in the analysis. Currently DoSSO does not handle recursive functions. Nevertheless, this limitation can be resolved by using fixpoint computation for recursions and can be accomplished in the future work of the project.

The other future work can be extending our tool with lockset algorithm to improve the precision of the results of the tool tailored for the programs that already contain the synchronization mechanism.

# REFERENCES

[1] *Therac - 25*. Retrieved February 16, 2009. Available: http://www.netcomp.monash.edu.au/cpe9001/assets/readings/www_uguelph_ca_~tgallagh_~tgallagh.html

[2] *Therac - 25*. Retrieved February 16, 2009. Available: http://www.computingcases.org/case_materials/therac/case_history/Case%20History.html

[3] *North America Blackout*. Retrieved February 16, 2009. Available: http://www.theregister.co.uk/2004/04/08/blackout_bug_report/

[4] M. Naik, *Effective static race detection for Java*, Unpublished doctoral dissertation, Stanford University, California, 2008.

[5] S. J. Hartley, *Concurrent Programming. The Java Programming Language*, Oxford University Press, 1998.

[6] D. Lea, *Concurrent Programming in Java. Design Principles and Patterns, Second Edition*, Addison – Wesley, 2000.

[7] B. Lewis and D. J. Berg, *Multithreaded programming with Java technology*, Prentice Hall, 2000.

[8] A. Betin-Can and T. Bultan, "Highly dependable concurrent programming using design for verification," *Formal Aspects of Computing,* vol. 19, pp. 243 -268, 2007.

[9] *Java 5.0 concurrency utilities*. Retrieved February 16, 2009. Available: http://java.sun.com/j2se/1.5.0/docs/guide/concurrency

[10] *Jsr – 166 concurrency utilities*. Retrieved February 16, 2009. Available: http://www.jcp.org/en/jsr/detail?id=166

[11] A. Betin-Can, T. Bultan, M. Lindvall, S. Topp, and B. Lux, "Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers," *Automated Software Engineering Journal,* vol. 14, pp. 129-178, 2007.

[12] M. B. Ari, *Principles of concurrent and distributed programming*, Prentice Hall Second Edition, 2006.

[13] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 237-252, 2003.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol. 15, pp. 391 – 411, 1997.

[15] J. Voung, R. Jhala, and S. Lerner, "Relay: Static race detection on millions of lines of code," in *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the 14th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 205 - 214, 2007.

[16] J. Choi, A. Loginov, and V. Sarkar, *Static datarace analysis for multithreaded object – oriented programs*, Technical Report, IBM Research Division, Thomas J. Watson Research Centre, New York, 2001.

[17] J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, "Escape analysis for Java," in *Conference on Object – Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 1-19, 1999.

[18] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for Java programs," *ACM SIGPLAN Notices*, vol. 34, pp. 187 – 206, 1999.

[19] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi, "Marmot: an optimizing compiler for Java," *Software – Practice and Experience*, vol. 30, pp. 199-232, 2000.

[20] *Java Tree Builder (JTB)*. Retrieved February 16, 2009. Available: http://compilers.cs.ucla.edu/jtb/jtb-2003/gjindex.html

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

[22] *Soot: a Java optimization framework*. Retrieved February 16, 2009. Available: http://www.sable.mcgill.ca/soot

[23] A. Betin – Can and T. Bultan, "Verifiable Concurrent Programming Using Concurrency Controllers," in *19th IEEE International Conference on Automated Software Engineering*, pp. 248 - 257, 2004.

[24] *Symbian OS*. Retrieved March 19, 2009. Available http://www.symbian.com/symbianos/os_smp.asp

[25] *Multithreading in Linux*. Retrieved March 19, 2009. Available http://whitepapers.techrepublic.com.com/abstract.aspx?docid=359061

# APPENDICES

# APPENDIX A

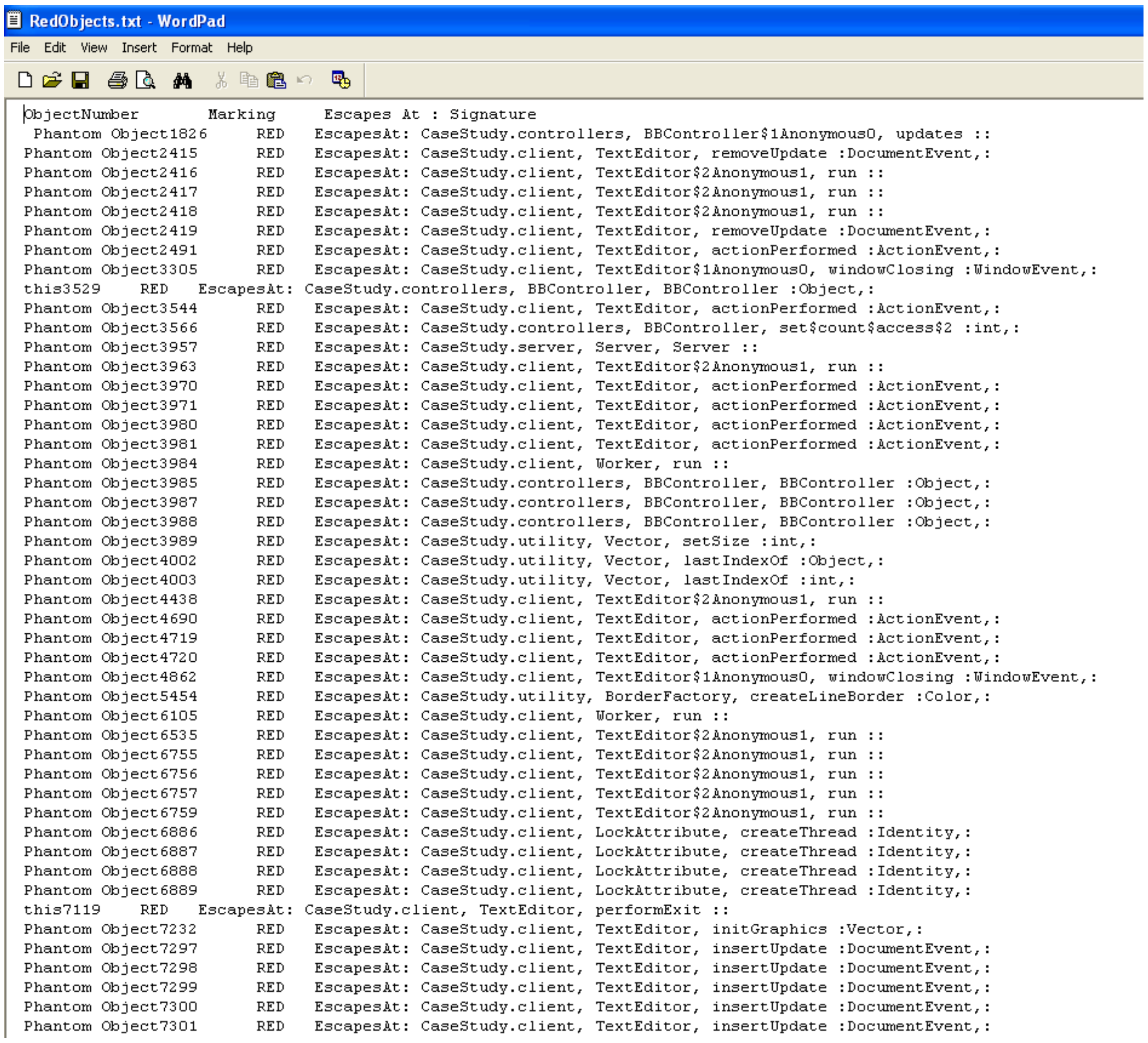
List of RED objects detected after conducting the case study.

| **Object Number** | **Marking** | **Escapes At: Signature** |
|---|---|---|
| Phantom Object1826 | RED | EscapesAt: CaseStudy.controllers, BController$1Anonymous0, updates :: |
| Phantom Object2415 | RED | EscapesAt: CaseStudy.client, TextEditor, removeUpdate :DocumentEvent,: |
| Phantom Object2416 | RED | EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run :: |
| Phantom Object2417 | RED | EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run :: |
| Phantom Object2418 | RED | EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run :: |
| Phantom Object2419 | RED | EscapesAt: CaseStudy.client, TextEditor, removeUpdate :DocumentEvent,: |
| Phantom Object2491 | RED | EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,: |
| Phantom Object3305 | RED | EscapesAt: CaseStudy.client, TextEditor$1Anonymous0, windowClosing :WindowEvent,: |
| this3529 | RED | EscapesAt: CaseStudy.controllers, BBController, BBController :Object,: |
| Phantom Object3544 | RED | EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,: |
| Phantom Object3566 | RED | EscapesAt: CaseStudy.controllers, BBController, set$count$access$2 :int,: |
| Phantom Object3957 | RED | EscapesAt: CaseStudy.server, Server, Server :: |
| Phantom Object3963 | RED | EscapesAt: CaseStudy.client, |

| | | |
|---|---|---|
| | | TextEditor$2Anonymous1, run :: |
| Phantom Object3970 | RED | EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,: |
| Phantom Object3971 | RED | EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,: |
| Phantom Object3980 | RED | EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,: |
| Phantom Object3981 | RED | EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,: |
| Phantom Object3984 | RED | EscapesAt: CaseStudy.client, Worker, run :: |
| Phantom Object3985 | RED | EscapesAt: CaseStudy.controllers, BBController, BBController :Object,: |
| Phantom Object3987 | RED | EscapesAt: CaseStudy.controllers, BBController, BBController :Object,: |
| Phantom Object3988 | RED | EscapesAt: CaseStudy.controllers, BBController, BBController :Object,: |
| Phantom Object3989 | RED | EscapesAt: CaseStudy.utility, Vector, setSize :int,: |
| Phantom Object4002 | RED | EscapesAt: CaseStudy.utility, Vector, lastIndexOf :Object,: |
| Phantom Object4003 | RED | EscapesAt: CaseStudy.utility, Vector, lastIndexOf :int,: |
| Phantom Object4438 | RED | EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run :: |
| Phantom Object4690 | RED | EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,: |
| Phantom Object4719 | RED | EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,: |
| Phantom Object4720 | RED | EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,: |
| Phantom Object4862 | RED | EscapesAt: CaseStudy.client, TextEditor$1Anonymous0, windowClosing :WindowEvent,: |
| Phantom Object5454 | RED | EscapesAt: CaseStudy.utility, BorderFactory, createLineBorder :Color,: |
| Phantom Object6105 | RED | EscapesAt: CaseStudy.client, Worker, run :: |
| Phantom Object6535 | RED | EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run :: |
| Phantom Object6755 | RED | EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run :: |
| Phantom Object6756 | RED | EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run :: |
| Phantom Object6757 | RED | EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run :: |
| Phantom Object6759 | RED | EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run :: |
| Phantom Object6886 | RED | EscapesAt: CaseStudy.client, LockAttribute, createThread :Identity,: |
| Phantom Object6887 | RED | EscapesAt: CaseStudy.client, LockAttribute, createThread :Identity,: |
| Phantom Object6888 | RED | EscapesAt: CaseStudy.client, LockAttribute, |

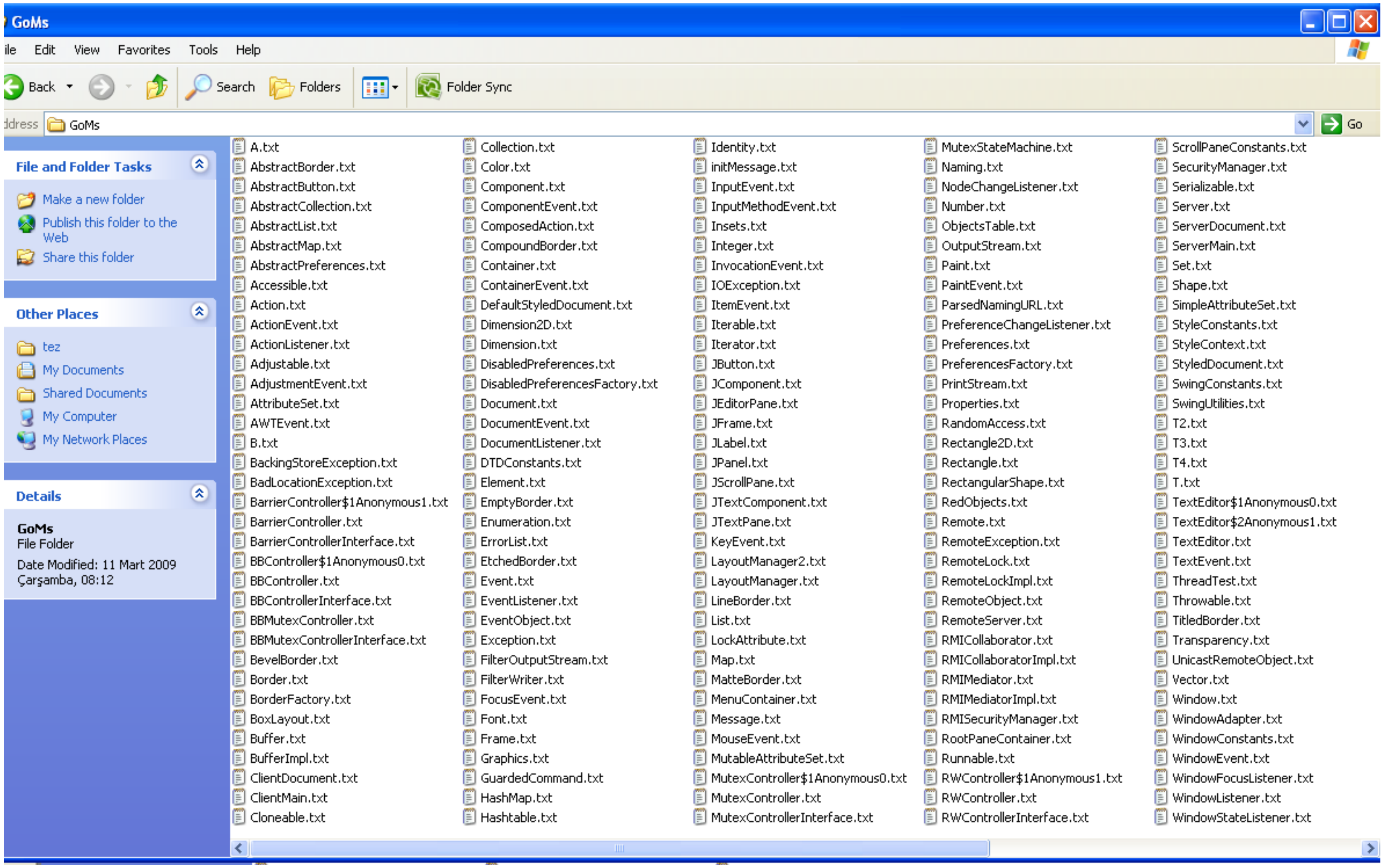
| | | |
|---|---|---|
| | | createThread :Identity,: |
| Phantom Object6889 | RED | EscapesAt: CaseStudy.client, LockAttribute, createThread :Identity,: |
| this7119 | RED | EscapesAt: CaseStudy.client, TextEditor, performExit :: |
| Phantom Object7232 | RED | EscapesAt: CaseStudy.client, TextEditor, initGraphics :Vector,: |
| Phantom Object7297 | RED | EscapesAt: CaseStudy.client, TextEditor, insertUpdate :DocumentEvent,: |
| Phantom Object7298 | RED | EscapesAt: CaseStudy.client, TextEditor, insertUpdate :DocumentEvent,: |
| Phantom Object7299 | RED | EscapesAt: CaseStudy.client, TextEditor, insertUpdate:DocumentEvent,: |
| Phantom Object7300 | RED | EscapesAt: CaseStudy.client, TextEditor, insertUpdate :DocumentEvent,: |
| Phantom Object7301 | RED | EscapesAt: CaseStudy.client, TextEditor, insertUpdate :DocumentEvent,: |
| Phantom Object7306 | RED | EscapesAt: CaseStudy.client, TextEditor, removeUpdate :DocumentEvent,: |
| Phantom Object7307 | RED | EscapesAt: CaseStudy.client, TextEditor, removeUpdate :DocumentEvent,: |
| Phantom Object7327 | RED | EscapesAt: CaseStudy.client, TextEditor, notify :Identity,: |
| Phantom Object7328 | RED | EscapesAt: CaseStudy.client, TextEditor, notify :Identity,: |
| Phantom Object7333 | RED | EscapesAt: CaseStudy.client, TextEditor, notify :Identity,: |
| Phantom Object7334 | RED | EscapesAt: CaseStudy.client, TextEditor, notify :Identity,: |

# APPENDIX B



```
ObjectNumber        Marking      Escapes At : Signature
 Phantom Object1826      RED   EscapesAt: CaseStudy.controllers, BBController$1Anonymous0, updates ::
Phantom Object2415      RED   EscapesAt: CaseStudy.client, TextEditor, removeUpdate :DocumentEvent,:
Phantom Object2416      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object2417      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object2418      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object2419      RED   EscapesAt: CaseStudy.client, TextEditor, removeUpdate :DocumentEvent,:
Phantom Object2491      RED   EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,:
Phantom Object3305      RED   EscapesAt: CaseStudy.client, TextEditor$1Anonymous0, windowClosing :WindowEvent,:
this3529     RED    EscapesAt: CaseStudy.controllers, BBController, BBController :Object,:
Phantom Object3544      RED   EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,:
Phantom Object3566      RED   EscapesAt: CaseStudy.controllers, BBController, set$count$access$2 :int,:
Phantom Object3957      RED   EscapesAt: CaseStudy.server, Server, Server ::
Phantom Object3963      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object3970      RED   EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,:
Phantom Object3971      RED   EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,:
Phantom Object3980      RED   EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,:
Phantom Object3981      RED   EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,:
Phantom Object3984      RED   EscapesAt: CaseStudy.client, Worker, run ::
Phantom Object3985      RED   EscapesAt: CaseStudy.controllers, BBController, BBController :Object,:
Phantom Object3987      RED   EscapesAt: CaseStudy.controllers, BBController, BBController :Object,:
Phantom Object3988      RED   EscapesAt: CaseStudy.controllers, BBController, BBController :Object,:
Phantom Object3989      RED   EscapesAt: CaseStudy.utility, Vector, setSize :int,:
Phantom Object4002      RED   EscapesAt: CaseStudy.utility, Vector, lastIndexOf :Object,:
Phantom Object4003      RED   EscapesAt: CaseStudy.utility, Vector, lastIndexOf :int,:
Phantom Object4438      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object4690      RED   EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,:
Phantom Object4719      RED   EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,:
Phantom Object4720      RED   EscapesAt: CaseStudy.client, TextEditor, actionPerformed :ActionEvent,:
Phantom Object4862      RED   EscapesAt: CaseStudy.client, TextEditor$1Anonymous0, windowClosing :WindowEvent,:
Phantom Object5454      RED   EscapesAt: CaseStudy.utility, BorderFactory, createLineBorder :Color,:
Phantom Object6105      RED   EscapesAt: CaseStudy.client, Worker, run ::
Phantom Object6535      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object6755      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object6756      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object6757      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object6759      RED   EscapesAt: CaseStudy.client, TextEditor$2Anonymous1, run ::
Phantom Object6886      RED   EscapesAt: CaseStudy.client, LockAttribute, createThread :Identity,:
Phantom Object6887      RED   EscapesAt: CaseStudy.client, LockAttribute, createThread :Identity,:
Phantom Object6888      RED   EscapesAt: CaseStudy.client, LockAttribute, createThread :Identity,:
Phantom Object6889      RED   EscapesAt: CaseStudy.client, LockAttribute, createThread :Identity,:
this7119     RED    EscapesAt: CaseStudy.client, TextEditor, performExit ::
Phantom Object7232      RED   EscapesAt: CaseStudy.client, TextEditor, initGraphics :Vector,:
Phantom Object7297      RED   EscapesAt: CaseStudy.client, TextEditor, insertUpdate :DocumentEvent,:
Phantom Object7298      RED   EscapesAt: CaseStudy.client, TextEditor, insertUpdate :DocumentEvent,:
Phantom Object7299      RED   EscapesAt: CaseStudy.client, TextEditor, insertUpdate :DocumentEvent,:
Phantom Object7300      RED   EscapesAt: CaseStudy.client, TextEditor, insertUpdate :DocumentEvent,:
Phantom Object7301      RED   EscapesAt: CaseStudy.client, TextEditor, insertUpdate :DocumentEvent,:
```

**Figure 26 Red Objects File**



File and Folder Tasks

- Make a new folder
- Publish this folder to the Web
- Share this folder

Other Places

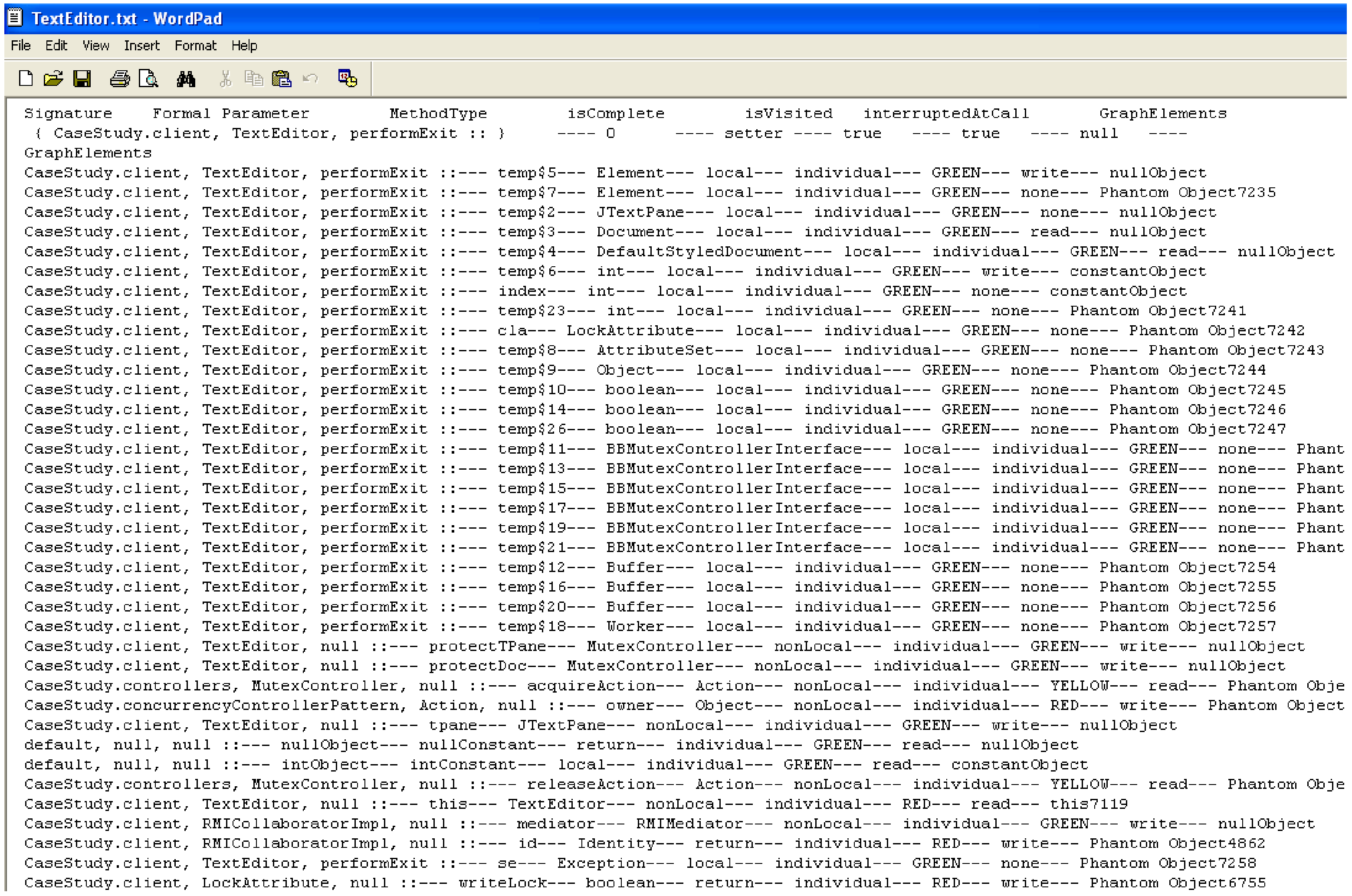
- tez
- My Documents
- Shared Documents
- My Computer
- My Network Places

Details

**GoMs**
File Folder
Date Modified: 11 Mart 2009 Çarşamba, 08:12

A.txt
AbstractBorder.txt
AbstractButton.txt
AbstractCollection.txt
AbstractList.txt
AbstractMap.txt
AbstractPreferences.txt
Accessible.txt
Action.txt
ActionEvent.txt
ActionListener.txt
Adjustable.txt
AdjustmentEvent.txt
AttributeSet.txt
AWTEvent.txt
B.txt
BackingStoreException.txt
BadLocationException.txt
BarrierController$1Anonymous1.txt
BarrierController.txt
BarrierControllerInterface.txt
BBController$1Anonymous0.txt
BBController.txt
BBControllerInterface.txt
BBMutexController.txt
BBMutexControllerInterface.txt
BevelBorder.txt
Border.txt
BorderFactory.txt
BoxLayout.txt
Buffer.txt
BufferImpl.txt
ClientDocument.txt
ClientMain.txt
Cloneable.txt
Collection.txt
Color.txt
Component.txt
ComponentEvent.txt
ComposedAction.txt
CompoundBorder.txt
Container.txt
ContainerEvent.txt
DefaultStyledDocument.txt
Dimension2D.txt
Dimension.txt
DisabledPreferences.txt
DisabledPreferencesFactory.txt
Document.txt
DocumentEvent.txt
DocumentListener.txt
DTDConstants.txt
Element.txt
EmptyBorder.txt
Enumeration.txt
ErrorList.txt
EtchedBorder.txt
Event.txt
EventListener.txt
EventObject.txt
Exception.txt
FilterOutputStream.txt
FilterWriter.txt
FocusEvent.txt
Font.txt
Frame.txt
Graphics.txt
GuardedCommand.txt
HashMap.txt
Hashtable.txt
Identity.txt
initMessage.txt
InputEvent.txt
InputMethodEvent.txt
Insets.txt
Integer.txt
InvocationEvent.txt
IOException.txt
ItemEvent.txt
Iterable.txt
Iterator.txt
JButton.txt
JComponent.txt
JEditorPane.txt
JFrame.txt
JLabel.txt
JPanel.txt
JScrollPane.txt
JTextComponent.txt
JTextPane.txt
KeyEvent.txt
LayoutManager2.txt
LayoutManager.txt
LineBorder.txt
List.txt
LockAttribute.txt
Map.txt
MatteBorder.txt
MenuContainer.txt
Message.txt
MouseEvent.txt
MutableAttributeSet.txt
MutexController$1Anonymous0.txt
MutexController.txt
MutexControllerInterface.txt
MutexStateMachine.txt
Naming.txt
NodeChangeListener.txt
Number.txt
ObjectsTable.txt
OutputStream.txt
Paint.txt
PaintEvent.txt
ParsedNamingURL.txt
PreferenceChangeListener.txt
Preferences.txt
PreferencesFactory.txt
PrintStream.txt
Properties.txt
RandomAccess.txt
Rectangle2D.txt
Rectangle.txt
RectangularShape.txt
RedObjects.txt
Remote.txt
RemoteException.txt
RemoteLock.txt
RemoteLockImpl.txt
RemoteObject.txt
RemoteServer.txt
RMICollaborator.txt
RMICollaboratorImpl.txt
RMIMediator.txt
RMIMediatorImpl.txt
RMISecurityManager.txt
RootPaneContainer.txt
Runnable.txt
RWController$1Anonymous1.txt
RWController.txt
RWControllerInterface.txt
ScrollPaneConstants.txt
SecurityManager.txt
Serializable.txt
Server.txt
ServerDocument.txt
ServerMain.txt
Set.txt
Shape.txt
SimpleAttributeSet.txt
StyleConstants.txt
StyleContext.txt
StyledDocument.txt
SwingConstants.txt
SwingUtilities.txt
T2.txt
T3.txt
T4.txt
T.txt
TextEditor$1Anonymous0.txt
TextEditor$2Anonymous1.txt
TextEditor.txt
TextEvent.txt
ThreadTest.txt
Throwable.txt
TitledBorder.txt
Transparency.txt
UnicastRemoteObject.txt
Vector.txt
Window.txt
WindowAdapter.txt
WindowConstants.txt
WindowEvent.txt
WindowFocusListener.txt
WindowListener.txt
WindowStateListener.txt

**Figure 27 GoMs Folder**



```
Signature    Formal Parameter        MethodType        isComplete        isVisited   interruptedAtCall       GraphElements
 { CaseStudy.client, TextEditor, performExit :: }     ---- 0      ---- setter ---- true    ---- true    ---- null    ----
GraphElements
CaseStudy.client, TextEditor, performExit ::--- temp$5--- Element--- local--- individual--- GREEN--- write--- nullObject
CaseStudy.client, TextEditor, performExit ::--- temp$7--- Element--- local--- individual--- GREEN--- none--- Phantom Object7235
CaseStudy.client, TextEditor, performExit ::--- temp$2--- JTextPane--- local--- individual--- GREEN--- none--- nullObject
CaseStudy.client, TextEditor, performExit ::--- temp$3--- Document--- local--- individual--- GREEN--- read--- nullObject
CaseStudy.client, TextEditor, performExit ::--- temp$4--- DefaultStyledDocument--- local--- individual--- GREEN--- read--- nullObject
CaseStudy.client, TextEditor, performExit ::--- temp$6--- int--- local--- individual--- GREEN--- write--- constantObject
CaseStudy.client, TextEditor, performExit ::--- index--- int--- local--- individual--- GREEN--- none--- constantObject
CaseStudy.client, TextEditor, performExit ::--- temp$23--- int--- local--- individual--- GREEN--- none--- Phantom Object7241
CaseStudy.client, TextEditor, performExit ::--- cla--- LockAttribute--- local--- individual--- GREEN--- none--- Phantom Object7242
CaseStudy.client, TextEditor, performExit ::--- temp$8--- AttributeSet--- local--- individual--- GREEN--- none--- Phantom Object7243
CaseStudy.client, TextEditor, performExit ::--- temp$9--- Object--- local--- individual--- GREEN--- none--- Phantom Object7244
CaseStudy.client, TextEditor, performExit ::--- temp$10--- boolean--- local--- individual--- GREEN--- none--- Phantom Object7245
CaseStudy.client, TextEditor, performExit ::--- temp$14--- boolean--- local--- individual--- GREEN--- none--- Phantom Object7246
CaseStudy.client, TextEditor, performExit ::--- temp$26--- boolean--- local--- individual--- GREEN--- none--- Phantom Object7247
CaseStudy.client, TextEditor, performExit ::--- temp$11--- BBMutexControllerInterface--- local--- individual--- GREEN--- none--- Phant
CaseStudy.client, TextEditor, performExit ::--- temp$13--- BBMutexControllerInterface--- local--- individual--- GREEN--- none--- Phant
CaseStudy.client, TextEditor, performExit ::--- temp$15--- BBMutexControllerInterface--- local--- individual--- GREEN--- none--- Phant
CaseStudy.client, TextEditor, performExit ::--- temp$17--- BBMutexControllerInterface--- local--- individual--- GREEN--- none--- Phant
CaseStudy.client, TextEditor, performExit ::--- temp$19--- BBMutexControllerInterface--- local--- individual--- GREEN--- none--- Phant
CaseStudy.client, TextEditor, performExit ::--- temp$21--- BBMutexControllerInterface--- local--- individual--- GREEN--- none--- Phant
CaseStudy.client, TextEditor, performExit ::--- temp$12--- Buffer--- local--- individual--- GREEN--- none--- Phantom Object7254
CaseStudy.client, TextEditor, performExit ::--- temp$16--- Buffer--- local--- individual--- GREEN--- none--- Phantom Object7255
CaseStudy.client, TextEditor, performExit ::--- temp$20--- Buffer--- local--- individual--- GREEN--- none--- Phantom Object7256
CaseStudy.client, TextEditor, performExit ::--- temp$18--- Worker--- local--- individual--- GREEN--- none--- Phantom Object7257
CaseStudy.client, TextEditor, null ::--- protectTPane--- MutexController--- nonLocal--- individual--- GREEN--- write--- nullObject
CaseStudy.client, TextEditor, null ::--- protectDoc--- MutexController--- nonLocal--- individual--- GREEN--- write--- nullObject
CaseStudy.controllers, MutexController, null ::--- acquireAction--- Action--- nonLocal--- individual--- YELLOW--- read--- Phantom Obje
CaseStudy.concurrencyControllerPattern, Action, null ::--- owner--- Object--- nonLocal--- individual--- RED--- write--- Phantom Object
CaseStudy.client, TextEditor, null ::--- tpane--- JTextPane--- nonLocal--- individual--- GREEN--- write--- nullObject
default, null, null ::--- nullObject--- nullConstant--- return--- individual--- GREEN--- read--- nullObject
default, null, null ::--- intObject--- intConstant--- local--- individual--- GREEN--- read--- constantObject
CaseStudy.controllers, MutexController, null ::--- releaseAction--- Action--- nonLocal--- individual--- YELLOW--- read--- Phantom Obje
CaseStudy.client, TextEditor, null ::--- this--- TextEditor--- nonLocal--- individual--- RED--- read--- this7119
CaseStudy.client, RMICollaboratorImpl, null ::--- mediator--- RMIMediator--- nonLocal--- individual--- GREEN--- write--- nullObject
CaseStudy.client, RMICollaboratorImpl, null ::--- id--- Identity--- return--- individual--- RED--- write--- Phantom Object4862
CaseStudy.client, TextEditor, performExit ::--- se--- Exception--- local--- individual--- GREEN--- none--- Phantom Object7258
CaseStudy.client, LockAttribute, null ::--- writeLock--- boolean--- return--- individual--- RED--- write--- Phantom Object6755
```

**Figure 28 GoM information of performExit method inside the TextEditor class**