DOI: 10.15439/2019F286
ISSN 2300-5963 ACSIS, Vol. 18

Search for the Memory Duplicities in the Java Applications Using Shallow and Deep Object Comparison

Richard Lipka
NTIS - New Technologies for the Information Society
Faculty of Applied Sciences
University of West Bohemia
Univerzitni 8, Plzen, 323 00, Czech Republic
Email: lipka@kiv.zcu.cz

Tomas Potuzak
Department of Computer Science and Engineering
Faculty of Applied Sciences
University of West Bohemia
Univerzitni 8, Plzen, 323 00, Czech Republic
Email: tpotuzak@kiv.zcu.cz

***Abstract*—In high-level object languages, such as Java, a problem of unnecessary duplicates of instances can easily appear. Although there can be a valid reason for maintaining several clones of the same data in the memory, often it indicates that the application can be refactored into a more efficient one. Unnecessary instances consume memory, but in case of Java applications can also have a significant impact on the application performance, as they might prolong the time needed for the garbage collection. In this paper, we are presenting a method and a tool that allows detecting duplicity in the heap dump of a Java application, based on the shallow and deep object comparison. The tool allows to identify the problematic instances in the memory and thus helps programmers to create a better application. On several case studies, we also demonstrate that the duplicates appear not only in the student projects and similar programs that often suffer from poor maintenance but also in commonly available Java tools and frameworks.**

I. Introduction

JAVA language was designed to provide fully automated memory management and to shield programmers from errors caused by the memory leaks. The developers are often encouraged to design the data models based on the real world structures and not to think too much about the internal representation of the data they are using. This should lead to greater efficiency of programming, but at the same time hardware resources are often used inefficiently. Consequently, instead of memory leaks typical in C-language programs, programmers are creating different constructions that clog up the operational memory and can lead to unnecessary slowdowns of the application (or even to the termination of the application due to insufficient memory) due to excessive garbage collection.

In general, this problem is known as the memory bloat, and there are many different aspects of it [1]. We had previous experience with fixing an application that was suffering heavily from wasting memory [2], so our main goal was to create a tool that would allow us to easily identify the problematic objects retained in the Java heap. One of the issues we have encountered is (often multiple) duplication of the identical instances in the memory. Especially for less experienced programmers, it might be difficult to identify the problem. We hope that our tool might help them to find the unnecessary objects in the memory of their programs. In the same time, we wanted to investigate how often the similar problem arises in other applications generally available in the Java community.

A. Memory Bloat

There is no generally accepted definition of the memory bloat, but many examples are known both from the literature and from the real applications. In [1] Mitchell describes 15 anecdotal examples of the memory issues that might arise, classified into four main groups, and shows how Java Virtual Machine deals with them and how programmers might or might not make its work more difficult. Anecdote 12 mentions data duplication created during the communication between Java application and the outside environment. Mitchell also describes how different types of objects can have a significant impact not only on the memory consumption but also on the application speed, as the Java Virtual Machine has to perform garbage collections. Depending on the number of retained objects, it can significantly slow down the application.

The problem of the object duplication is partially solved also in Java Virtual Machine itself, currently only with the `String` class. Since Java version 8.20 [3] Java contains an implementation of the string deduplication – as long as `Strings` are managed by the virtual machine, they are created in the separate part of the memory. When a new `String` shall be created, it is first checked whether there already is an instance with the same content. If so, only a reference on the existing instance is provided. As `Strings` are immutable in Java, this is a safe way of dealing with them - all operations manipulating with `Strings` are in fact creating new instances, so when a programmer wants to change the content of the instance, a new, different instance with new data is provided.

This behaviour is possible mainly due to the simple nature of the `String` object – only an array of characters needs to be checked. However, not only `Strings` instances are duplicated in Java programs. Our intention is not to provide a more general, runtime method for the deduplication, as the deep object comparison can be quite time-consuming. We only intend to provide a tool that will allow to analyze the program memory and to discover possible duplicates. We also do not claim that the duplicate objects can be automatically merged just because they contain the same data - this decision has to be made by a programmer with a deep insight into the application. However, if there are multiple identical instances of one class, it can be a strong indicator that the application can be refactored into a more efficient one.

B. Motivation

Our motivation comes from two sources. First, we are often dealing with software created by students, which usually contains many types of problems and we were not able to find a tool that would be able to show how many duplicates are present in the memory of student programs. Standard profilers such as *VisualVM* can be helpful, but not suitable for this type of analysis. Furthermore, we wanted to see if this problem is present not only in the work of inexperienced programmers but also in software that is more widespread and freely available.

The remainder of the paper is organized as follows: Section II deals with the related work, focused mainly on the problem of the equality and comparison of the objects. Section III explains how the equality of the objects is implemented in our tool. In section IV we are showing the algorithm used for the duplicity analysis in our implementation. The method of validation of our implementation, as well as several results of duplication analysis of several Java application, is presented in section V. The last section concludes the paper and discusses possible future work.

II. Related Work

There are two main areas relate to our work. The first one is the problem of the memory efficiency of Java applications. This problem is discussed quite extensively and many different bad practices or problematic patterns were described in last years. Especially with the incresing interest in the embededd systems, the need for the memory efficient software grows [4].

The most common problem is the detection of the memory leaks, described for example in [5] or in [6]. As the Java is a language with garbage collection, the classical memory leaks with inaccessible memory are rare in it, and the works we are mentioning are focusing more on the detection of the objects with large overhead [5]. The typical example might be collections that contain mostly `null` elements. Another approach is the design of the more efficient ways of the garbage collection [6]. As one of the main sources of the performance issues and memory wasting is automated ORM (Object-Relational Mapping) when used incorrectly, it is also possible to find approaches focusing on the analysis of the ORM performance antipatterns and fixing them [7].

Other approaches are trying to find a way how to evaluate overall memory health of the Java programs. One of the most useful works on this topic is [8], where the problematic structures are described in high detail and even a metric based on the ratio of the useful data and structure overhead is proposed to evaluate memory health. These ideas are further expanded in [1] and [9], where the memory impact of the complexity of the domain model is discussed, as each reference occupy some space. Several different antipatterns are presented, along with the proposed solutions. Unfortunately, the patterns described in this work are not discovered automatically, the authors rely mostly on the manual analysis of the data structures. Another approach to the memory optimization is based on the efficient memory partitioning [10].

One important inspiration for our work is [11], where both static and dynamic analysis is proposed as a tool for evaluating the usage of Java collections (or other structures that allows manipulation with a large number of elements). Another approach for the dynamic analysis of the collections efficiency is presented in the [12] – the technique proposed here aims not only to use collections more efficiently, but also to choose the most suitable collection for the application, based on the runtime analysis.

The second is more focused on the instances comparison or even automatic detection of the possibility of replacing one instance with another. In [13], a post-mortem analysis of the Pharo programs is proposed in order to determine if there are some redundancies in the suitable classes with inclination to the redundancy (such as `Point` or `String` classes), aiming to replace the redundant ones with one instance. This paper also contains an extensive description of different ways how to define an object equivalence. The possibility of the replacement of one instance with another is also examined in [14]. In this case, not only a comparison of the objects is performed, but authors also propose to instrument the original program in order to observe the usage of the candidates for the merging and automatically determine if such merging is possible without influencing the program behaviour.

As the search for duplicates in the whole namespace is time demanding task with a high complexity, some publications are focusing only on the classes which are known to contain duplicates very often. In [15], the methods used for the `String` deduplication is described in high detail. Similarly, the description of the approaches to get rid of the `String` duplicates is described in [3]. As this is typically performed at runtime, there is a great need to make these algorithms as efficient as possible. In [16], different methods for the decision whether the deduplication should be performed or not and their impact on the program performance is demonstrated.

III. Object Equality

As was mentioned in previous section, there are multiple ways of how the object equality can be defined. As our application is working with the heap dump, we did not focus on the fast pre-analysis using some form of the hash code of the objects, such as in [17], but immediately on the analysis of the attributes of the objects, similar as described in [18]. In contrast with [18], we are not working directly with objects

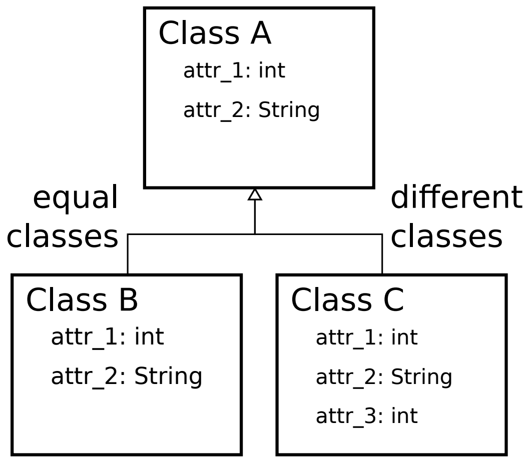

Fig. 1. Class equality in inheritance chain

in memory, but with their serialized form in the heap dump. So for the comparison, we cannot use in any way methods that are implemented in the objects, such as `equals()` or `compareTo()`.

A. Class Equality

The first aspect to consider is the class of the compared objects. If we are comparing instances with the same class, it makes sense to analyze the data field by field. If the values of all fields are identical, we can consider the instances identical as well. However, it is also possible to consider inheritance. The inheritance in Java cannot remove any field from the successor, only to add additional ones or to add or override its methods. In case that the predecessor contains the same fields as the successor, we can consider them identical from the data point of view as well. Of course, there is a question if such objects can be merged into one, but that is something that can not be decided automatically during the heap analysis. Thus, we consider instances identical in case all their fields are identical even when they belong to different classes, as long as they are part of the same inheritance chain and no additional fields are added in the successors (as you can see in Fig. 1). When such field is added, even if its value is `null`, it is considered different due to the different class definition and the matching of fields is not performed at all. You can notice that including inheritance chains in the comparison, in fact, broadens the definition of what are identical objects and can lead to a higher number of identified duplicates.

B. Fields Equality

When fields are compared, it is simple to deal with the primitive types, but more complicated to deal with the references (see Section III. C). Although `String`s are references, Java deals with them in a special way. This enables us to treat them in a special way as well. As they are stored in a special part of memory and due to the string deduplication, we often do not need to analyze the actual content of the `String`, only to see if the reference leads to the same instance. When references are different, the actual data of the `String` have to be compared. This can happen, as the string deduplication does not work for all `String`s in Java application and the instance of the `String` can thus appear both in the regular heap space and in the area reserved for the `String`s.

C. References Equality

Another aspect is dealing with the reference fields. The previously described method is suitable for the objects that are composed only of the primitive data types and `String`s. However, in Java, most objects contain also references on other objects. In such a case, there are two different points of view. The shallow approach would consider two references identical only when they are pointing to the same instance. This can be checked very easily, as in the heap dump, the references are represented only as a `long` number, so we only need to compare those.

In order to obtain a broader set of results, we have also implemented a deep comparison approach. It means that, in addition to the identical reference numbers, two references will be considered identical when they point to two different instances that are internally identical. Again, this approach leads to a higher number of identified duplicates, but also significantly increases the complexity of the comparison, as it has to be used in a recursive way – the referred instances can point to other instances and so on. This also means that a stopping condition has to be defined, in order to deal with cycles and to improve the performance of the comparison.

The references can create an arbitrary graph, but it can be always reduced to a finite tree structure when the analysis is stopped after each node of the graph is visited once. Another option is to define a depth, in which the analysis should end. If the instances fields contents are identical till the required depth is reached, the instances themselves are considered identical as well.

The last aspect we need to describe is the dealing with arrays and collections. Java offers `List` and `Set` interfaces and several implementations that can be used to store a large number of data. In case when the deep object comparison is used, no special approach is needed and the structures are identical only when they contain identical instances in the same order. It would be possible to broaden equality definition even more and ignore the order of the instances within the array, but that would require even more complex calculation and specific implementation for each Java collection.

IV. Duplication Finder

Our implementation of the duplication finder is created in the Java language. As we need to deal with a heap dumps in binary HPROF format, which is created by means of JVM, we were looking for a tool that would allow us to process the data easily and we used a Hprof Heap Dump parser library [19]. This library allows us to load the data from heap dump and reconstruct the content of each instance. It also provides access to the class descriptions so the data can be correctly interpreted.

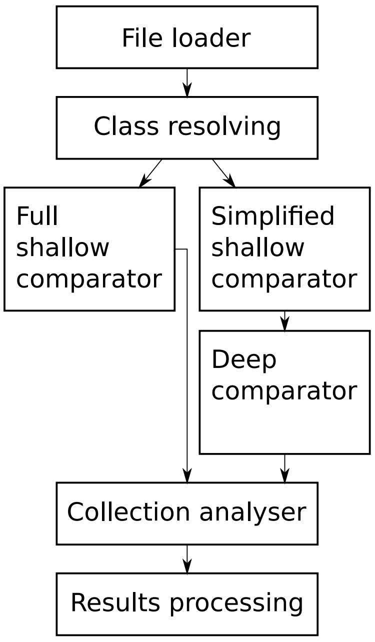

Fig. 2. Tool pipeline

A. Tool Architecture

In order to allow easy modification of the analysis process (for example to be able to switch between shallow and deep comparison or to add additional modules searching for other memory anti-patterns), the tool is designed as a pipeline. Our tool sequentially reads the heap dump file and produce a basic representation of the loaded data. The overall behaviour is represented in Fig. 2.

These data are joined with the corresponding class descriptions and then further processed according to the class description (so the loaded byte streams are converted for example to long numbers or to `String`s for easier processing). When the actual class of each instance is analyzed, it is also possible to decide if the analysis should stop or continue depending on the class or package name – this allows us to limit the duplicity search only on the certain classes in the memory dump and thus save some time during the analysis.

B. Problem Identification

The prepared data are then passed to a module that is responsible for the duplicity analysis. The matching algorithm iterates over all loaded instances and stores them according to their properties in the two-level structure. At first, instances are divided according to their classes (as was described in the previous section – so the different classes can be considered equivalent if they are part of the same inheritance chain and they share the same set of fields). Then, within each class, the equivalent instances are grouped according to the values of their fields (see Fig. 3). Depending on the settings, the shallow or the deep comparison of the objects is performed in this phase.

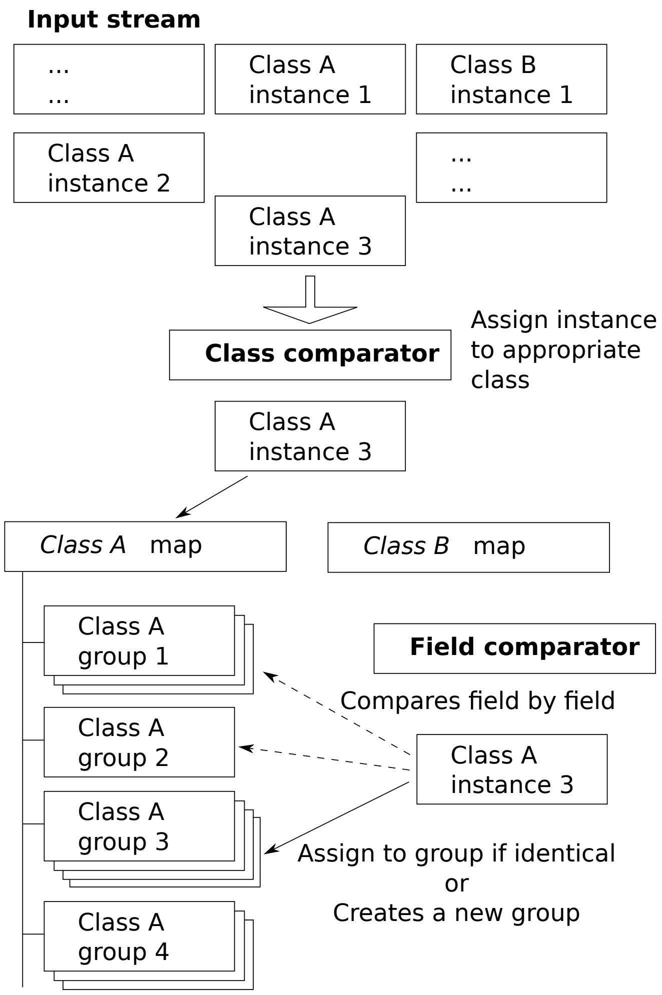

Fig. 3. Data structures for the equality classification

The shallow comparison is quite straightforward – the instances retrieved from the heap dump are compared field by field, including the references (see Fig. 4). If the instances are identical on this level, they are considered equal and became part of the group within the class. Each additional instance is compared against all existing groups and either is added to one of the groups or a new group is created if the new instance is unique.

Deep comparison is more complicated and more time demanding. The algorithm is similar to the deep object comparison algorithm we have described in [20], but for this purpose modified and made faster. There are two main differences – the first one is that we are now working with the heap dump data and not with the instances that are in memory of the executed application. The second is that we do not need to explore the structures of instances completely - the first occurrence of a difference is sufficient to claim that the compared objects are not identical. It would be possible to use the original version of the algorithm as well, but it would make the whole comparison process even more time demanding.

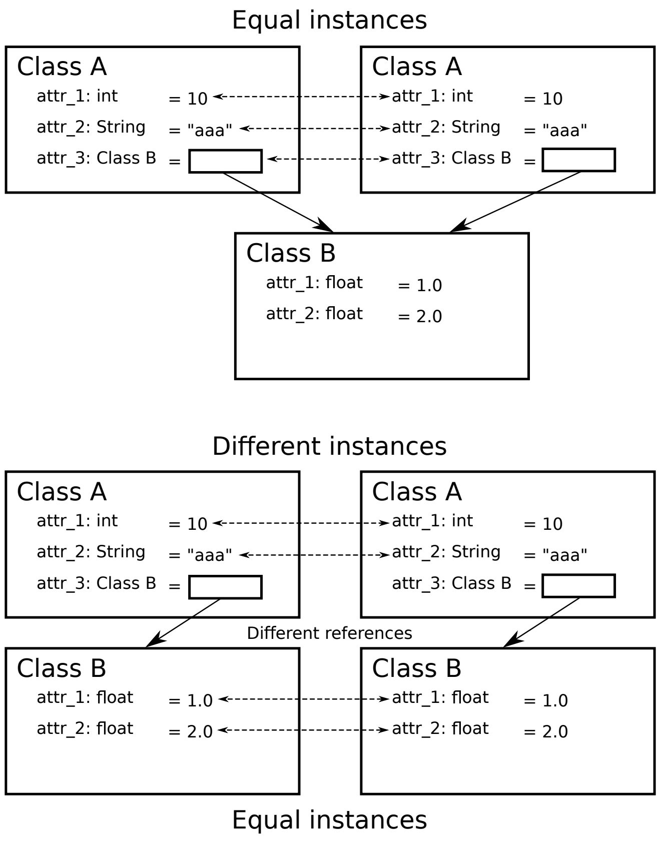

Fig. 4. Instances equality with Shallow comparison

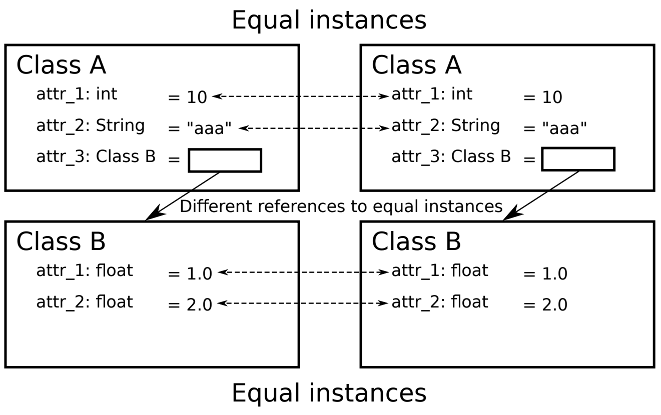

Fig. 5. Instances equality with Deep comparison

In order to perform the deep comparison, a graph representation of both compared objects has to be created. In order to do so, the whole memory dump have to be processed (we have to be able to resolve references to create an object graph), so it is performed after the shallow comparison is finished. But for the purpose of the deep comparison, a modified version of shallow comparison is done. In this case, only primitive data types are compared to determine if the instances are identical and the references are ignored. This, of course, means that the instances that differ only in references will be considered identical during this modified shallow comparison. The reason for this is to allow faster evaluation of the deep comparison. These "duplicates" are not reported in the result of the algorithm, but only used in order to evaluate the equality of the referred instances during the deep comparison faster.

Each node of the graph corresponds to an instance and each edge corresponds to the reference. As we expect that, in most cases, the instances will not be identical, the graph is constructed on demand, as a modification of Breadth First Search (BFS) algorithm. When all fields of compared instances are identical, the references are resolved one by one (see Fig 5). As the deep comparison is actually performed after the shallow one, in the first step algorithm checks if the referred objects were identical during the shallow comparison. If not, the comparison can be terminated, as we are not looking for all differences between instances, only one difference is sufficient to consider them unique. If the referred instances belong to the same shallow equivalent class, other references of the original object are analyzed. Only when all referred instances are considered identical, the algorithm starts constructing the next level of the graph.

Finally, the module for the analysis of the lists and arrays is used. For this analysis, we considered using the idea of collection health described for example in [8]. Collections are considered to be healthy (from the memory utilization point of view) only if they carry a sufficient amount of data. Programmers in Java often create unnecessary large pre-allocated structures, which are never sufficiently utilized, so the collection contains a significant number of `null` references. Such references, despite not carrying any data, still occupy memory space. For example, default instance of the `ArrayList` is created with 10 empty slots and often only small fraction is used (more specifically, the array is allocated when the first element is inserted to it, when the collection is created empty, the array is not created immediately).

This analysis is currently limited only on the offsprings of the `AbstractList` class, which are based on the array. As each collection requires a different approach and implementation, we focused only on the array types and not on the linked structures or maps. The analytical module is looking for two parameters. First, it calculates the ratio of the space occupied by the collection to its size. All collections that have this ratio below 0.5 are marked as underutilized. Second, the content is analyzed for duplicity and, if the collection is filled with identical elements, it is marked as problematic as well.

C. *Result Reporting*

The last part of the processing is the results reporting. As the tool is so far only operated from the command line, the results are presented only in the text form in the standard output stream. The result report contains names of all classes that have at least one set of duplicate instances (including information about the package and the inheritance chain), the serialized form of fields of duplicate instances and also the list of collections that contain one of the marked problems.

D. Complexity

The complexity of comparing each instance with each other to determine if they are identical is implemented with quadratic complexity with respect to the number of elements to compare. It can be expressed as $O(n^2)$, where n is a number of elements to compare. However we do not really need to compare each element with each other across the whole heap dump - we need to compare elements only within the same class. In such case, complexity is still quadratic, but in a form $O(k \cdot n_k^2)$ where k is the number of equivalent classes and n_k is a number of the instances within each class. This makes the comparison more feasible, the number of instances within a class is more manageable value. We have to note here that the time even for the shallow comparison itself can differ significantly according to the number of the fields in compared classes. In case of the deep comparison, it depends heavily on the complexity of the compared structures. This have a significant impact on algorithm performance as well. Theoreticly, it would be possible to achieve linear complexity by calculating hash for each instance and compare only the hash codes, but the calculation of hash for the deep comparison would still require a complete reconstruction of the data structures, even in cases when they differ in first few attributes (and the comparison will quickly find differences), so we decided against this approach.

V. Validation and Results

In order to validate the functionality of our tool, we have at first created a simple test application with a known number of duplicates and half empty collections. The purpose was just to figure out whether the tool will be able to find all injected problems. Then we have continued with tests performed on several real-life applications, to investigate whether the duplicates occur in the real world software.

A. Testing Application

The testing application is able to generate an arbitrary number of instances of simple objects, containing numerical and string attributes, as well as simple reference structures. It uses only two simple classes, `Child` and `Parent` that can refer to each other. The testing application was run several times with the different number of the created instances, in order to verify the time complexity of the algorithm. In each test, there were only 5 deep duplicates.

The heap dump was obtained using `jmap` [21] tool, using command

```
jmap −dump:live , file=<file −path> <pid>
```

in order to ensure that only "living" objects (objects that would survive next garbage collection) are obtained and listed in the heap dump. The time measurements were done on the PC with Windows 10, SSD drive, Intel Core i7-4930K CPU, 3.40 GHz and 32 GB RAM. No parallelization was used at the moment. The analysis was limited only to the package with our data classes, other instances were ignored. The times were obtained as an average from 5 executions.

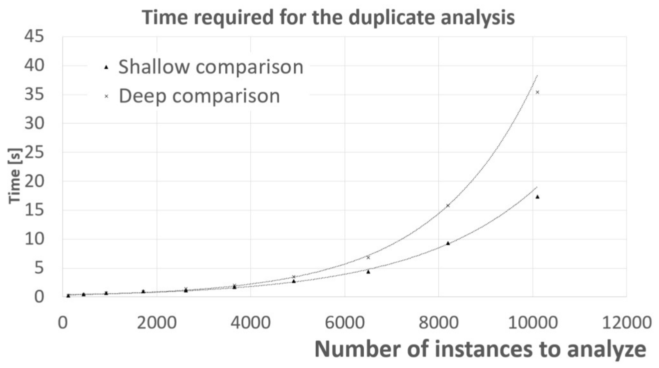

Fig. 6. Times required for analysis by deep comparison and shallow comparison algorithms

In Table I, the results for the testing application are presented. From these results, the deep and shallow comparisons can be compared both in their abilities and consumed computation time. In this simple case, our deep comparison algorithm was able to find all duplicates, while the shallow comparison version was not able to identify duplicates that were using different referred instances with identical data. On the other hand, the time required for the deep comparison grows significantly faster even when there are no significantly more deep duplicates (see Fig. 6). Running the deep analysis thus makes sense only if there is a strong suspicion that there might be such kind of problem. As expected, time requirements appear to grow quadratic in relation to the number of instances that are there for comparison, although the measurements, especially for the small number of instances might be influenced by the overhead of the processing.

B. Real World Programs

We have also attempted to run the algorithm on several real-world programs, to investigate whether if the problem with memory duplicates will be present in them. We have chosen four different programs – *Spring Boot framework* with a basic hello-world application, *Eclipse* with several projects open, *IntelliJ Idea* and *TomEE* with running map visualization server. Due to the time complexity of the deep analysis, we used only a shallow comparison in all four examples.

1) Spring Boot: The *Spring Boot framework* [22] in version 2.1.4 was used with only the very basic "Hello World" application. The dump of the whole framework has approximately 27 MB. For the purposes of the experiment, we have decided to limit our testing only to the classes from the `org` namespace.

In Table II, the results of the `org` namespace and its parts are summarized. One important find is that some duplicates were in fact found, but there is not a great number of them. Mostly, the duplicities were only pairs, but for example, the `Signature` class had 38 identical instances and class `DefaultFlowMessageFactory` 34 instances. Both classes only contain short `Strings` with basic framework settings, but their presence in the analyzed application shows that our tool is able to work with the real software.

TABLE I
RESULTS FOR THE SIMPLE TESTING APPLICATION

Instance count	Dump size [MB]	Injected duplicates	Found duplicates (shallow)	Found duplicates (deep)	Duration shallow [ms]	Duration deep [ms]
120	2.1	20	15	20	221	227
440	2.6	30	25	30	493	401
920	3.0	40	35	40	725	705
1710	3.6	50	45	50	987	912
2620	4.0	60	55	60	1121	1409
3650	4.4	70	65	70	1753	2021
4910	5.4	80	75	80	2769	3517
6500	5.9	90	85	90	4379	6781
8200	6.4	100	95	100	9334	15789
10100	6.9	110	105	110	17351	35419

TABLE II
ANALYSIS OF CLASSES IN SPRING BOOT FRAMEWORK

Package name	Classes	Instances	Found duplicates	Duration [ms]
`org`	2416	9093	347	14759
`org.springframework`	1555	6053	329	8214
`org.springframework.boot`	380	1506	27	4229
`org.springframework.core`	196	1585	5	4425
`org.springframework.web`	296	239	37	4108
`org.springframework.boot.web`	75	27	1	4002

Furthermore, as the Spring Boot framework is often used and well maintained, we did not expect to find many problems in it.

The second thing to notice is the ratio of classes and instances within one class. In the whole `org` package the ratio is approximately 1 : 4. For many classes, there are only a few instances. During drill down to the sub-packages, the ratio changes. For example in the package `org.springframework.boot.web` the ratio is even reversed – more classes were loaded from the namespace than was actually used to create instances.

2) Eclipse: We have analyzed *Eclipse* [23] in version 4.10.0 (build 20181214-0600). The IDE was only started and in the moment of heap dump collecting was not performing any particular task. The size of the Eclipse heap dump was approximately 92 MB. Measurements were performed for the packages `org`, `com`, `java`, `sun`, and `ch`. The results are summarized in Table IV.

This is the largest heap we have processed and, again like in the heap of the *Spring framework*, no large problem is present. However, the tool demonstrates that it is capable handling not only trivial examples but also larger datasets. The most duplicated was the class `org.eclipse.swt.widgets.TypedListener` with 444 identical instances based on the shallow comparison. Many of the discovered classes contained large fragments of the XML configuration of the tool (like `org.eclipse.sisu.plexus.ConfigurationImpl` with 16 identical instances containing 750 characters each). The results also show the rapid growth of the required computational time for larger datasets, with the analysis of `java` package taking more than 6 hours. In this package, 75 ms on average was required for analysis of each instance. In comparison in the smallest package `ch`, only 13 ms on average were required.

3) IntelliJ Idea: Along with Eclipse, we also tried to perform analysis of IntelliJ Idea in version 2018.3. The dump of this IDE was smaller than in the case of the Eclipse, approximately 74 MB. Only packages `org`, `com`, and `sun` were available within. In a similar way as in the previous case, the IDE was not performing any particular task, it was just started. The results are summarized in the Table IV.

In this case, despite the lower number of the classes with duplicates, a large number of identical instances was found. The class `org.jdom.Text` contained several instances with many clones, the largest group had 11577 identical instances. All these clones contained only several unprintable characters (typically end of the line and some other character) and obviously were part of the loaded DOM of some data the IDE was requiring after starting. In this case, the tool demonstrates that it is capable of discovering large clusters of the identical instances. However further analysis of the source texts of the *IntelliJ Idea* would be required to determine if there is a way to mitigate this type of the duplicity. Other duplicity classes (with only several clones) contained for example the text of the library licenses.

4) TomEE with Visualisation Server: The last example we tried to analyze was Apache *TomEE* [22] server in version 7.1.5, with the running application dealing with the visualization of the power grid. *TomEE* is a version of the Tomcat server, with additional modules useful for building enterprise applications. In this case, we have decided to focus not on the classes from the technology itself, but on the domain objects from the visualization server. As previous examples showed, the frameworks that are intensively used will probably contain fewer issues than the applications that should be executed within them.

The size of the heap dump of the server was approximately 370 MB, significantly larger than the previous files. When the dump was collected, the server was working with 4 users at the moment, so 4 sessions with data models were loaded. We were focused on the proprietary package `cz.zcu.laps.pnp.domain`, which contains domain data of the application. The data were organized in the form of a graph, composed of the nodes representing elements of the power distribution network and power lines between them. Each user is able to work only with one model at one time. As the graph structure was maintained by the different package (*JGraphT* library), the nodes have no references to other objects except enums, so the only shallow analysis was required.

The package contained 48 different classes and 49096 instances. Further analysis showed that the instances are distributed only among 6 classes of the domain model. The shallow analysis of this namespace took 3.22 hours on the same machine. In this case, the structure of the results was quite different – in each class, multiple triplets of identical instances were discovered.

Further manual analysis of the result showed that the problem, in this case, was in the different sessions. As we went through the triplets, it became obvious that they are part of the same graph – in fact, the duplicates were not only the nodes of the graph, which were discovered by our tool, but the graphs themselves. The reason for this was that two of the users were visualizing the same graph and the server maintained a copy of all the data for both sessions and also – as a form of the cache memory – a third copy not related to any session. As only the visualization was required from the server, it would be possible (in this particular case) to merge all data and maintain only one copy for every user who needs it. This issue is similar to the problems described in [7], as the graphs are also mainly products of the ORM. However, in this case, we cannot really speak about the ORM antipattern, the problem is more in the design of the data structures and lack of sharing data between users in the moments when it is possible.

However, our tool was not able to discover this issue directly, as no package from *JGraphT* was analyzed, so there was no overview of the whole structure during the analysis. This shows obvious limitation when only part of the namespace is analyzed – the data structures that are keeping data are not part of the analysis and even if the deep comparison is used, the identical structures will not be visible.

TABLE III
ANALYSIS OF CLASSES IN THE ECLIPSE IDE

Package name	Classes	Instances	Found duplicates	Duration [ms]
`org`	9647	141970	756	5007822
`com`	919	27906	865	90271
`java`	1155	313405	39	23596884
`sun`	929	28092	20	91228
`ch`	244	539	5	7335

TABLE IV
ANALYSIS OF CLASSES IN THE INTELLIJ IDEA IDE

Package name	Classes	Instances	Found duplicates	Duration [ms]
`org`	2016	157743	283	8425230
`com`	7687	77927	261	1290908
`sun`	1119	15620	31	26023

Furthermore, for such big structures as the graphs in our case (49096 instances are in fact only data in 5 graphs, without the overhead of the *JGraphT* library), the deep analysis would be quite time demanding – especially if parts of the graph would be shared and only some parts would be changed. On the other hand, this type of the situation – data shared or cloned on the server among several sessions – can be an example when the analysis of the duplicities is useful.

VI. CONCLUSION AND FUTURE WORK

In the paper, we have presented an algorithm and tool designed to search duplicates in the memory space of the Java applications. The analysis is based on the exploration of heap dump and comparing primitive fields of the objects with the same class. We have implemented both shallow and deep comparisons and demonstrated their functionality on the sample application. According to the results, the algorithm is capable of finding the duplicates that are present in the memory for both simple objects from the testing application and also in the data obtained from four real-world applications. The quadratic complexity of the algorithm, along with the need to compare deep structures, prevents processing a large number of instances, but even on the real heap dumps the algorithm was able to perform the analysis within hours. As this type of the analysis is not something that needs to be performed often, but mainly in the situation when there is a problem with resource consumption, we believe that the tool is practically utilizable.

Main goal of our immediate future work is to implement the parallelization of the task. The shallow comparison should be simply parallelizable, as when all instances from the heap are loaded to the memory of the analyzer, the comparisons need to be performed only within each class and thus can be distributed among the working threads.

As for deep object comparison, the problem is more difficult there, as all instances need to be present in the memory when a graph of referred objects should be constructed. However, if sufficient memory is available, the task can be still distributed, as each worker can obtain the whole copy of the heap and then work on the analysis of objects within a particular class. The question remains if the communication between such workers would allow making the process faster, for example, if the sub-graphs of the compared objects are already processed and the information about their equality is available. This approach would require to determine a sequence, in which objects should be compared and evaluated, in order to have simpler objects processed before the complex one.

VII. Acknowledgment

This research was supported by the project LO1506 (PUNTIS) of the Czech Ministry of Education, Youth and Sports under the program NPU I

References

[1] N. Mitchell, E. Schonberg, and G. Sevitsky, “Four trends leading to java runtime bloat,” *IEEE Software*, vol. 27, no. 1, pp. 56–63, Jan 2010.

[2] K. Jezek and R. Lipka, “Antipatterns causing memory bloat: A case study,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 306–315.

[3] P. Liden. (2017) String deduplication in g1 (acccessed: 13 may 2019). [Online]. Available: http://openjdk.java.net/jeps/192

[4] K. Hadj Salem, Y. Kieffer, and S. Mancini, “Formulation and Practical Solution for the Optimization of Memory Accesses in Embedded Vision Systems,” in *PROCEEDINGS OF THE 2016 FEDERATED CONFERENCE ON COMPUTER SCIENCE AND INFORMATION SYSTEMS (FEDCSIS)*, ser. ACSIS-Annals of Computer Science and Information Systems, Ganzha, M and Maciaszek, L and Paprzycki, M, Ed., vol. 8, PTI; IEEE. 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE, 2016, Proceedings Paper, pp. 609–617, Federated Conference on Computer Science and Information Systems (FedCSIS), Gdansk, POLAND, SEP 11-14, 2016.

[5] G. Xu and A. Rountev, “Precise memory leak detection for java software using container profiling,” in *2008 ACM/IEEE 30th International Conference on Software Engineering*, May 2008, pp. 151–160.

[6] M. Jump and K. S. McKinley, “Cork: Dynamic memory leak detection for garbage-collected languages,” *SIGPLAN Not.*, vol. 42, no. 1, pp. 31–38, Jan. 2007. [Online]. Available: http://doi.acm.org/10.1145/1190215.1190224

[7] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, “Detecting performance anti-patterns for applications developed using object-relational mapping,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1001–1012. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568259

[8] N. Mitchell and G. Sevitsky, “The causes of bloat, the limits of health,” *SIGPLAN Not.*, vol. 42, no. 10, pp. 245–260, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1297105.1297046

[9] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O’Sullivan, T. Parsons, and J. Murphy, “Patterns of memory inefficiency,” in *Proceedings of the 25th European Conference on Object-oriented Programming*, ser. ECOOP’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 383–407. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032497.2032523

[10] D. Langr and I. Simecek, “On Memory Footprints of Partitioned Sparse Matrices,” in *PROCEEDINGS OF THE 2017 FEDERATED CONFERENCE ON COMPUTER SCIENCE AND INFORMATION SYSTEMS (FEDCSIS)*, ser. Federated Conference on Computer Science and Information Systems, Ganzha, M and Maciaszek, L and Paprzycki, M, Ed., PTI; IEEE. 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE, 2017, Proceedings Paper, pp. 513–521, Federated Conference on Computer Science and Information Systems (FedCSIS), Prague, CZECH REPUBLIC, SEP 03-06, 2017.

[11] G. Xu and A. Rountev, “Detecting inefficiently-used containers to avoid bloat,” *SIGPLAN Not.*, vol. 45, no. 6, pp. 160–173, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806616

[12] O. Shacham, M. Vechev, and E. Yahav, “Chameleon: Adaptive selection of collections,” *SIGPLAN Not.*, vol. 44, no. 6, pp. 408–418, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1543135.1542522

[13] A. Infante and A. Bergel, “Object equivalence: Revisiting object equality profiling (an experience report),” *SIGPLAN Not.*, vol. 52, no. 11, pp. 27–38, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3170472.3133844

[14] D. Marinov and R. O’Callahan, “Object equality profiling,” in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, ser. OOPSLA ’03. New York, NY, USA: ACM, 2003, pp. 313–325. [Online]. Available: http://doi.acm.org/10.1145/949305.949333

[15] K. Nasartschuk, M. Dombrowski, K. B. Kent, A. Micic, D. Henshall, and C. Gracie, “String deduplication during garbage collection in virtual machines,” in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON ’16. Riverton, NJ, USA: IBM Corp., 2016, pp. 250–256. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049877.3049904

[16] K. Nasartschuk, K. B. Kent, S. A. MacKay, A. Micic, and C. Gracie, “Improving garbage collection-time string deduplication,” in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON ’17. Riverton, NJ, USA: IBM Corp., 2017, pp. 113–119. [Online]. Available: http://dl.acm.org/citation.cfm?id=3172795.3172809

[17] D. F. Bacon, S. J. Fink, and D. Grove, “Space- and time-efficient implementation of the java object model,” in *Proceedings of the 16th European Conference on Object-Oriented Programming*, ser. ECOOP ’02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 111–132. [Online]. Available: http://dl.acm.org/citation.cfm?id=646159.680023

[18] N. Grech, J. Rathke, and B. Fischer, “Jequalitygen: Generating equality and hashing methods,” *Sigplan Notices - SIGPLAN*, vol. 46, pp. 177–186, 10 2010.

[19] E. Aftandilian. (2018) Hprof heap dump parser (acccessed: 13 may 2019). [Online]. Available: https://github.com/eaftan/hprof-parser

[20] T. Potuzak and R. Lipka, “Deep object comparison for interface-based regression testing of software components,” in *Proceedings of the 2018 Federated Conference on Computer Science and Information Systems, FedCSIS 2018, Poznań, Poland, September 9-12, 2018.*, 2018, pp. 1053–1062. [Online]. Available: https://doi.org/10.15439/2018F51

[21] Oracle. (2019) The jmap utility (acccessed: 13 may 2019). [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr014.html

[22] P. Software. (2019) Spring boot 2.1.4 (acccessed: 13 may 2019). [Online]. Available: https://spring.io/projects/spring-boot

[23] I. Eclipse Foundation. (2019) Eclipse ide 2018âĂŠ12 (acccessed: 13 may 2019). [Online]. Available: https://www.eclipse.org/downloads/