Group 6
Software Engineering SENG 300 Winter 2018
Department of Computer Science
University of Calgary

April 9, 2018

GROUP ASSIGNMENT#[3]

Choosing process

The process of choosing the 100 systems was based around the use of a criteria as well as where our interests lie. We split the decision to choose among our teammates and chose 10 system per person. When selecting the systems for analysis, we looked to cover a wide range of topics stretching amongst gaming, data processing, storage and analysis. Among the systems chosen, we included systems from medical fields, NASA, biological research and game development. We covered the entertainment industry as well as the practical systems used in the working industry. We looked into research systems that functioned similarly and chose a few outliers to compare and contrast between them. Having outliers aided in evaluating the significance of certain Java types and provided different ways of implementing different programs. This helped us see if there are substitutes available to using these types and how the developers could have put them to use. Furthermore, it was also important for us to ensure compilations of systems when choosing them. If projects were far from complete and not compiled by simple fixes(for example adding a jar), we discarded the system.

In addition to the criteria introduced above, the criteria helped choose model representatives which do a respectable job at representing the large pool of systems out in the public. The systems who pass the criteria give effective information for analysis. The goal of analysis is to reflect on the significance of certain Java Types and their place in the Java language. This goal called for the consideration of certain aspects as follows. Is the system representative of all systems out in the public? It is not possible to ensure that a particular system is a model representative, so we must narrow the margin of error through the use of solid examples. An example is one that would have been supported past its release for the next number of years. Another aspect of a reliable choice would be one which sees use, with regular use being optimal. Githubs interface allows for the evaluation of such aspects with its vast user base. The feedback and contributions to a project are a great reflection of its reliability. In conclusion, the systems were selected based on user feedback, reliability, history of and fixing of bugs, and the usage of such systems. These qualities are what determines a model representative, which covers for the vast population of systems. In conclusion, there are many systems to choose from, with our criteria we were able to narrow down the pool and further narrowed the pool based on our varying interests.

Systems which were not able to be run by quick fixes then confirmed the presence of bugs within our tool, which is something which is always welcomed when it arises. This shows our tool isnt perfect for every system out there and being able to catch these instances are things we want to be able to do.

Data Collection Process

Following the selection process mentioned earlier, the analysis tool developed from the two previous iterations were modified in order to count and display the numbers of nested, local, anonymous and other types. Each (10) individual member from our group ran the same analysis tool on the

source code from ten different projects. The same analysis tool was used in order to eliminate potential inconsistencies that would arise from using different tools that were developed from previous iterations. The usage of this tool was intended to count the occurences of these specific Java types to aid in drawing conclusions from them. Counting the occurrences of the types we are concerned with creates a means for comparison which displays the significance of a type based on the amount it is used. Upon running the tool on a programs source code, Each individual took the output from each of their ten Java projects and recorded it into an Excel spreadsheet for comparing and contrasting the values with other Java projects and drawing an overall conclusion based on a sample of unbiased 100 Java projects. At this point, any bugs that were found in the analysis tool after cross-referencing it with a different tool used within the group were addressed before the analysis process. Following the data collection and recording process, a relative answer can be drawn based on our analysis of the data concerning the significance of nested, local, and anonymous Java types in industry or outside of an academic setting. During the analysis process, a few speculations were made such as low usage of these java types is associated with lower significance where a high usage could reflect the practicality of a certain type.

The different tool used to cross-reference results was made independently of the tool used to make the final analysis. The reason for this was to catch any mistakes that mightve been made along the way, all the results we were to attain had to be accurate and another tool confirmed either, the presence of errors in another tool or the accuracy of another tool. Emphasis should be made on unconnected creation of these tools, as it served the purpose to debug and produce the final product. This is not to assume that the second tool was also completely bug free, no quite the opposite, we are aware that nothing can be perfect but as stated before this minimized the errors occurring in the final product by making sure as many as possible were caught by both tools.

Analysis Process

After gathering our data, we crunched some numbers to find how often nested, local and anonymous classes were used. Comparing all 100 projects against each other we found that:

1. Nested types were:

- used between 0.3% - 8.6% compared to other types.
- On average 2.30% compared to other types.
- With outliers that were greatly out of that range e.g. 0.03% and 15.6%.

  Nested types seem to be quite frequent throughout all of the 100 programs. It appears to be positively correlated with the number of lines of java code (i.e. with the complexity of the given system). Complexity of the given system can be estimated by overall usage of any type. Primitive types are used most often. It appears that the more primitives we have the more nested types we have as well. They also seem to be more present in systems such as music players as these are focused more on frontend (GUIs, visuals, events). They are declared more often than local types as those seem to be very rare. Compared to the anonymous types, no correlation appears to be present in this case.

2. Local types were:

- Used between 0.08% - 6.2% compared to other types.

- On average 0.83% compared to other types.
- With outliers that were out of range e.g. 13.3%.

  Anonymous types were quite rare at least in our systems. The highest number we found in a single program was 17. However, thats an exception as most of our systems had none. It also seems to be the case that they are rare because most of our systems have ample dependencies and thus have a lot of their code outsourced.

3. Anonymous types were:

- Used between 0.0% - 9.4% compared to other types.
- On average 1.16% compared to the other types.
- With outliers that were out of range e.g. 14.2%.

  Anonymous types dont seem to be related to any other types. In fact the only property they seem to be related to is the size/complexity of the program and even that is not accurate 100% of the time. It seems that different programmers have different preferences when it comes to anonymous types. It makes their systems more concise but more difficult to get oriented in. Nonetheless, the larger the system gets, the more likely they are to appear. Also there is no correlation with the usage of other types.

4. In total the above three were:

- Used between 0.0% - 16.3% compared to other types.
- On average 4.29% compared to other types.
- With outliers that were out of range e.g.17.1%.

  Based on these results, nested, local, and anonymous declaration types are rarely used in comparison to other declaration types (non-nested, non-local, non-anonymous type declarations).

  Speculations based on the data

  Whereas some types make up approximately the same number in the sample, for example, the interface and class declaration and annotations types, primitive types almost always had the highest number of declarations in a system. A reason why primitive types are so common is that they are efficient and directly hold the value instead of just references. In addition, primitive types also cost less space in memory and have a fixed size, which makes memory allocation easier while manipulating the data.

  Taking a close look at the values for nested, local and anonymous types, we see that the values of these were directly impacted by their application within the specific project. Lets look at this closer:

(a) Nested speculations:
Among others a Dex To Java Decompiler, Protocol Library, JUnit 4 and a Calendar System were projects which had nested types with amounted to a percentage of 8% or higher compared to other types. The commonality between these is that they are either libraries or assistance tools. Assistance Tools simply being something to make life easier but in reality could be done by hand but itd be more tedious e.g. converting Dex To Java or keeping track of things on a calendar.
The reason why these in particular would have a greater number of nested types we would assume is due to built in libraries that already occur in java. The tools would simply need to get access to certain methods/classes in predefined or user-defined interfaces and classes to create these projects. Another reason is the purpose of security, most of these projects have members within their classes that should not be accessed elsewhere e.g. assuming we have 2 instances of a calendars that we want to keep completely separate, on a particular day lets assume May 17, calendar1 should not be able to view whats on the schedule for calendar2 and vice versa.
Nested inner classes are also used for the purpose of testing, e.g. creation of stubs, these systems could possibly have more testing involved compared to other systems because they need to be accurate. These systems will either be run on a daily basis or are systems in which when run cannot afford to make mistakes. For example with the calendar, we cant have a situation where a calendar misinforms a surgeon of his scheduled appointments and he missed an operation on a patient in dire need.
We also see nested anonymous classes here, which could be used in the instance of a Calendar to handle events on the GUI associated with it, or Overriding methods to do the same task with different inputs, which is very likely as we have seen with our assignments with the ASTParser. A lot of the time there was overriding of visit methods within an anonymous class and the same tasks might definitely need to be done for the Dex to Java Decompiler. These classes can be used as arguments to method or when instantiating objects.
Another reason could involve static inner classes which just allow the developer to access members and methods without actually having to instantiate the classes which could be useful to speed the process of accessing overridden methods in that class or member values.

Local speculations:

Consideration of nested, local and anonymous types in practice

Nested types can be simply defined as a class that is defined within another class. Local and Anonymous types are a special type of nested types. Suffice to say, these types share properties and some slight differences. They are encapsulated within an enclosing class and have access to the members of the said enclosing class. This allows for easier logical grouping of classes and makes code more readable.

In practice, nested types are simply used for logical grouping while local and anonymous classes are used in more specific situations. Local types allow more than one instance of the type, while anonymous allow for declarations of additional fields and methods. Their usage seems to vary but it also appears that when developing a larger program, developers use them to make their systems more concise, readable and generalized. They provide additional functionality and versatility to systems that other types simply cannot. They allow developers to encapsulate methods in a help class that are useless outside of the class it is nested within. This is useful in preventing data leaks. If used in a proper way, they can be very worthwhile.

Accuracy of Analysis Tool

A few bugs were found in our analysis tool, when we initially ran the tool through the project we were not counting any anonymous class declarations which was quickly fixed by visiting the AnonymousClassDeclaration node and incrementing the counter. After running it a second time, we found that we were missing annotations which was also fixed by visiting the appropriate nodes using the ASTVisitor and incrementing the annotation counter. One major bug that we found was that the tool was throwing a NullPointerException error for some of the projects, this was also handled by adding try-catch clauses and also including the appropriate counters since some of the NullPointerExceptions originated from the visit methods from the ASTVisitor, specifically due to getFullyQualifiedName(), where not all types had a fully qualified name. After debugging, the projects were then ran through the tool once again in order to ensure that the data we collected were the same as before, if the count values did not match for the three earlier systems that were re-ran by each team member, then all the selected projects were ran through the tool once more and the data in the excel spreadsheet was updated in order to ensure accuracy.

Accuracy of Results

We created test suites to test our tools using regular as well as edge cases to ensure it works the way it should. We determined that our tool produces accurate results with the edge cases and regular cases that we used to test it with, but concluded that there is no way to know for certain if the results for the bigger system is exact. We applied our past experience to note that some of the data that showed heavy usage of primitives must be accurate since it corresponded to our programming experiences as well. However, some of the systems we looked at were large which made it difficult to estimate the number we would get, therefore rendering us incapable to determine whether or not our tool is producing the exact output we are looking for.

To know whether the results are correct, one approach would be to manually count the declarations we are looking for in our systems and check whether we get the same numbers. However, this process is very long and tedious and is likely to be riddled with human errors.

Generalization of results to industry trends

Based on the analysis of our results, large Java systems that are used in industry tend follow a general trend such that they do not incorporate nested, local, and anonymous types as often compared to other types within their programs. The results provide a basic idea of the general structures that larger systems tend to follow in a very broad sense. Although we can not make the conclusion that they are not used at all in the industry (since we have counted some of these types), it is safe to say that the practicalities of these types are more common in an academic setting than in practice. Since we see this trend in Java-based systems, it is possible that these trends carry into other non-Jare likely representative in other programming languages as well, for example in different large non-Java systems, we may find that there are more primitive types, non-nested, non-local, and non-anonymous types than nested, local, and anonymous types. In terms of generalization, the results that we obtained suggested that in typical large object oriented systems, these systems tend to have less than twenty percent of their types cva object oriented systems that allow for nested, local, and anonymous types to exist. The results obtained comprised of nested, local and anonymous types.

Stability of Nested, Local and Anonymous types

Measuring stability through the evolution of of nested, local and anonymous types across multiple versions of the same system, it can be concluded that these types are relatively stable. When put to use, these types see little to no change based solely on how they function. If a system were to eliminate the use of these types, it would not be due to buggy functioning and rather the evolution of the entire system and the favouring of an overhaul of the source code.

Reworking our analysis procedure to include multiple versions would aid in verifying the stability of these types. As opposed to putting our focus on the current version of the systems, looking at their previous iterations would give a better look at the evolution of the software. This insight on the history of the system helps draw conclusions and make assumptions about how stable these types are. Suppose the types are unstable, it is safe to assume that they are subject of change. Tracing back to previous versions and counting the use of these types would reveal their value and helps back to perception of them being stable.

We could change the analysis tool to record the names of the nested, local and anonymous types instead of just their counts. That way if a particular type were no longer used it could be compared to previous versions and determined to be unstable. If the list of types stays relatively the same with similar counts, it can be concluded that it is stable

Submitted by Group 6 on April 9, 2018.