EECE 437 Code Review

Abbas Zeitoun
201500542

Thursday, March 2, 2017

1 Numeric Information

The following numeric information excludes the 3 "main" files used for testing the code (one "main" file per problem):

- Number of files: 50
- Number of classes: 47 (Three of the files were incomplete and left empty)
- Lines of code (Including white space, excluding comments)[1]: 2538
- Lines of comments[2]: 23
- Ratio of comments to code: $23/2538 = 0.9\%$
- Maximum level of inheritance[3]: 3
- Readability rating: 8.5
  The code was readable and understandable, even for someone that has not seen Java's generics before. The scarcity of comments leaves the code uncluttered, especially in places where the functionality of the code can be seen immediately by reading the implementation. It would have been a smoother read, though, if some comments were added to

[1] Lines including in-line comments at the end were counted as both a line of code and a line of comments.

[2] Code that was commented out as a means of excluding it but not deleting it completely was not counted as a line of comments.

[3] Assumes that the root of every class hierarchy tree is at level 0 and ignores the default inheritance from java.lang.Object for non-inheriting classes i.e. java.lang.Object is effectively considered at inheritance level -1.

clarify the slightly more complex parts of the code (e.g. The part where TACeilingInt and TACeilingDouble are wrapped together in the same class TACeiling).

- Coding style consistency rating: 7
  The coding style was not entirely consistent, but it seems like my classmate was on the right track. Even though the style was not consistent across all classes and functions, it is obvious that some style patterns are being repeated across some classes and functions. This is possibly due to my classmate changing her style as she became more knowledgeable of what is most readable throughout the course of the assignment. Since the assignment is incomplete as of the date of writing this review, it cannot be ascertained whether or not the complete assignment would have an entirely consistent coding style, but I tend to lean towards the probability of it having one.

2 On Form

- The name of the directory does not reflect the functionality of the contents of the directory.

- The checked-in items only consist of source code files, README files, and subdirectories to organize the content into three parts corresponding to the three problems of the assignment. The checked-in items correctly exclude any executables or object files.

- The different units of code were separated by placing each class in its own .java file.

- Class names, method names and file names are chosen consistently and reflect functionality or structure. However, variable names are not chosen as consistently. In some cases, variables are given a full name. In other cases (like some constructors for example), they are given one-letter names, and their meanings have to be deduced from their usage. Even though in some cases the meanings are apparent after some minor code inspection, it is better to use full, meaningful variable names wherever possible and to use unambiguous abbreviations of some words if the variable name is too long.

3 On Operation

- The code compiles. The only (minor) issue is in the main2.java file that requires renaming the public class from "main" to "main2" before compiling.

- The code runs and produces expected results. However, some parts of the code were incomplete and therefore some functionality is missing. The README file mentions some of these missing functionalities by stating some limitations of the current design:

  – TAConstant currently does not support non-integer primitive types but requires some modifications to support those.

  – TAArray currently is only specialized for integers, doubles and booleans (through TAArrayInt, TAArrayDouble, and TAArrayBool respectively). The absence of mentions of other types indicates that more complex types are not currently supported.

  Other issues not mentioned in the README file include:

  – The non-specialized versions of operators do not support taking other non-specialized operators as operands (e.g. TALessThan (TAMinus, TAMinus)). However, it should be noted that the code does support passing specialized descendants of these operators to other non-specialized operators (e.g. TALessThan(TAMinusInt, TAMinusInt))).

  – Implementations of TAArrayAccess, TAArrayBool, and TAArrayDouble were absent (the files were left empty).

  – Due to the way the class hierarchy was structured (See section 4), not all classes can be passed as elements to TAPair. This is because TAPair only takes as elements instances of classes that are descendants of TAObject, and not all classes were made descendant of TAObject. Resolving this issue is easy, however, and it is very likely that it was caused accidentally to begin with, not by design.

  (See sections 7 and 8 for explanations and/or suggestions regarding some of these missing functionalities.)

4 On Content

- The classes in the code can be divided as follows[4]:
  - TAInt, TADouble, and TABool: primitive types; inherit from TAObject.
  - TAArray (and its descendants TAArrayInt, TAArrayDouble, and TAArrayBool)[5] and TAPair: composite types; also inherit from TAObject.
  - TAConstant: a constant integer value; inherits from TAObject.
  - TAAnd, TAOr, TANot, TAXor, and TAImplies: formulae; inherit from TAObject.
  - TAMinus, TAPlus, TADivide, and TAMultiply (each having a TAOperationInt descendant and a TAOperationDouble descendant): terms; inherit from TANumericFunction, which inherits from TAObject.
  - TACeiling and TAFloor (each having a TAOperationInt descendant and a TAOperationDouble descendant): terms; inherit from java.lang.Object directly.
  - TAEqual, TALessThan, and TAGreaterThan (each having a TAOperationInt descendant and a TAOperationDouble descendant; TAEqual having an additional TAOperationBool descendant): formulae; inherit from java.lang.Object directly.

  Additionally, an interface hierarchy was designed and implemented to allow the above classes to be divided into the four mentioned spaces. In particular, the hierarchy is rooted at TAValue and has three descendants: TAIntValue, TABoolValue, and TADoubleValue. Each of the above classes implements one or more of these interfaces based on the type of the value that they should return. For example, formulae implement TABoolValue because they return boolean values.

- Most of the code functions properly, as reflected in the three main methods: constructors, list methods, and evaluate methods. The list method correctly prints the name of the function then calls the list method of the operands, and this recursive calling procedure results in the proper printing of the string equivalent of the statement. Similarly,

[4]The classification excludes TAException, which inherits from java.lang.Exception.

[5]Even though TAArrayBool and TAArrayDouble were not implemented, it is obvious that they are meant to be descendants of TAArray–similar to TAArrayInt.

the evaluate method correctly calls the evaluate method of the operands before calculating the result. Furthermore, the constructors, coupled with the TAValue interface hierarchy, enforce at compile time that the parameters passed to the above term and formulae classes as operands have the proper return type.

- As for the type coupling issue, it seems like the decision was to go with strings to represent the types in most cases. The only exception to this was in TAArray, in which my classmate used the values of the primitive types themselves as a way of choosing which kind of array to instantiate. The idea was smart because that delegated the instantiation process to the constructors and went around the need to process the string itself. However, this method still has its shortcomings as it does not scale too well given that the number of primitive types is limited. Alternatively, the idea of using strings to represent the types is more easily extendable: indicating that an object is an array of a certain type, for example, is as easy as concatenating a bracket '[' to the end of the type of the elements in the array. A similar approach can be followed to deal with types of pairs.

- As of the time of writing of this review, the only constant values that are supported are integers. This was done through the implementation of a TAConstant class that encapsulates the integer value.

5 Further Considerations

- The submission does not put the problem into context, but possible uses of the code can be seen easily, even if not mentioned. This is because such a mathematical system can be used in almost anything that requires math, and therefore it seems fitting to try to keep it as general as possible instead of tailoring it to one or two applications only.

- The submission does not state principal design decisions.

6 Design Strengths

- The submission uses two hierarchies: one for classes and one for interfaces. This allows the classes that are similar in structure or that logically belong together to inherit from a common parent class while

also supporting different return types where needed for sibling classes. Additionally, having a return type interface hierarchy makes it easier to extend the system later on by adding more operations or types: the newly added operation or type simply has to implement the existing interface, and it would be compatible with all existing functions and operations.

- For operators that can have different return types (e.g. TAPlus can return either an integer or a double), the submission includes specialized versions of the operators, one for every return type. This allows for compile-time type checking when nesting these specialized operators within other operators. Note that operators cannot support compile-time type checking for nested non-specialized operators, for by definition, the non-specialized operators must be general enough to support several return types. This means that the return type of the non-specialized operators cannot be known before runtime, and thus, to my knowledge, it is not possible to support both compile-time type checking and to support nesting non-specialized operators at the same time.

- Operators contain references to their operands, not copies/clones of them. This allows the operator to always have access to the most recent value of its operands, and thus allows it to evaluate the correct result after its operands' values are updated.

7 Deviations from the Requirements

- The list method does not print a space after listing the second operand. This makes the resulting string a lot cleaner and easier to read for people. It also makes the listed string slightly more compact. However, it is still advisable to ensure that the omitted space is not being used for any particular task (for example during the process of de-serializing the string back into classes) before omitting it.

- Non-specialized versions of operators cannot be nested within other operators.This is due to prioritizing compile-time type checking (See section 6 for more details).

8 Advice and Recommendations

The following list of recommendations is not meant to be comprehensive but was created based on whatever suggestions came to mind. It is meant to provide a starting point from which the code can be improved and does not ensure that the code will be perfect after the recommendations have been followed, as there is always room for further optimizations and improvements. The recommendations are, in no particular order:

- The placement of braces needs to be more consistent. In particular, alternating between placing braces on their own lines and writing code on the same line the braces open and/or close on reduces from the cleanliness of the code.

- The amount of whitespace between items declared within a class needs to be less varied. Having too much whitespace in some places and just one line of whitespace in other places gives off the impression that something is missing, when in reality nothing is.

- It makes more sense to place the member variables of a class at the beginning and the methods of the class afterwards, as usually the programmer will start reading from the top, and it is less confusing to see the declarations of the variables before their usage inside the class's methods.

- Where possible, variable names should be descriptive of their functions or purpose.

- Comments should be added to clarify the non-trivial or potentially ambiguous parts of the code.

- In most cases, siblings under a parent class should be similar in structure or function. If they differ too much from one another, it might be a good idea to classify them under intermediate classes. Specifically, direct descendants of TAObject should be grouped up under intermediate classes similar to TANumericFunction. Additionally, classes that inherit from java.lang.Object should be made to inherit from TAObject or one of its descendants for consistency (The exceptions to this obviously being TAValue and TAException).

- While I am not aware of the ideal way of representing types in such a system, I would recommend the use of strings over the use of the values of the primitive types themselves, simply because of the former's ability to support more complex types.

- Code in the checked-in files that is not meant to be used and that is not meant to be a comment should be deleted rather than commented out.

- The names of directories should be changed to reflect the functionality of their contents.