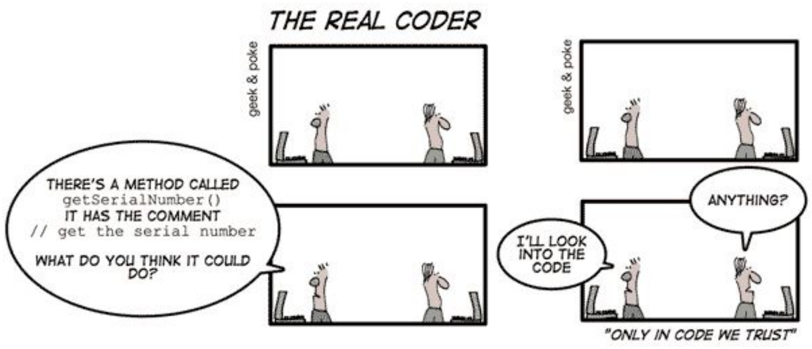
# Corollary: a Java Documentation and Verification Tool

Jed Chen and Glenn Zhang

**Motivation**

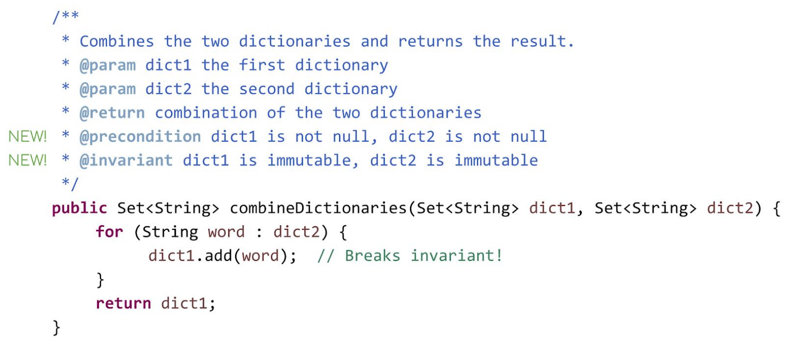
The purpose of this project is to help people write better documentation and be more certain of the correctness of the documentation in Java. Many programmers currently find it difficult to define what is necessary to include in their comments. Sometimes, programmers may forget to document their edge cases. Other times, they forget to implement them. People that read code may find commenting styles to differ between two different classes, making it difficult to understand. Lombok Project aimed to identify many common usages of variables within Java with tags such as @NonNull and @Data. This could help in some cases, but ultimately we are looking for a solution not to help write code, but to verify it outside of runtime.


THE REAL CODER
geek & poke
geek & poke
THERE'S A METHOD CALLED getSerialNumber() IT HAS THE COMMENT // get the serial number WHAT DO YOU THINK IT COULD DO?
ANYTHING?
I'LL LOOK INTO THE CODE
"ONLY IN CODE WE TRUST"

In Java, there are Javadocs, which is a form of documentation that includes parameters, exceptions, returns, and some other tags. With these, it is feasible to write good documentation. We wish to enhance the Javadocs format by adding tags that would help people understand how to use a piece of code: preconditions, invariants, and postconditions. By using specific keywords within these tags, there will be no ambiguity as to what these conditions are. In addition to this, the software will ensure that the code the programmer wrote follows these tags. These two features combined will make a structured form of documentation that directly connects documentation and correctness. In conclusion, we wish to add Javadocs tags that would represent preconditions, invariants, and postconditions that can be verified through our software, Corollary.

**Approach**

The high level approach is to prepare a script. Using static analysis, the script will take the program as an input and indicate whether the program is working as intended according to the Javadoc tags. If the program fails to follow the Javadoc tags, the script will return error codes and messages to inform of the programmer where the errors occurred and possible solutions to the errors. This allows the documentation to have a larger definite relationship to the code that it describes. Though there have been other approaches that achieve similar goals of verification like Lombok, they lack in establishing a wider variety of information that programmers try to convey.



```
      /**
       * Combines the two dictionaries and returns the result.
       * @param dict1 the first dictionary
       * @param dict2 the second dictionary
       * @return combination of the two dictionaries
NEW!   * @precondition dict1 is not null, dict2 is not null
NEW!   * @invariant dict1 is immutable, dict2 is immutable
       */
      public Set<String> combineDictionaries(Set<String> dict1, Set<String> dict2) {
          for (String word : dict2) {
              dict1.add(word);  // Breaks invariant!
          }
          return dict1;
      }
```

Example of tags @precondition and @invariant

**Limitations, Risks, and Challenges**

Our approach requires programmers to follow certain formats when writing their documentation, which may discourage many from adapting to this system. If this project becomes too obscure to use, the assistance it provides in its translation between code and documentation diminishes. Another limitation the project has is the limit of the length of documentations. Documentations typically are kept short for readability, but the verification aspect of this project is stronger with more lines.

If the style of documentation turns out to be too difficult to follow for people, the documentation contributes little to nothing to communicate the properties of code. We would like to be able to cover a large number of types of invariants, but we will start from easily verifiable ones (e.g. x is greater than y) and move to more difficult ones (e.g. list has no repeats).

Putting this into an IDE would be nice, but with so many different ones, it is difficult to push out a software to all of them. We will stick to reading a text file (with a reach goal of implementing it in some IDE). Similarly, different programming languages have different syntax and nuances, making it difficult to create a software that can work on multiple languages. For that reason, we have decided to focus on writing the software for Java, which we are most familiar with.

Hours: 8 hours (Combined)