Metrics in Java Projects

TAN, Shin Hwei

陈馨慧

Southern University of Science and Technology

Slides adapted from http://metrics2.sourceforge.net/

Lab Part 1: Cyclomatic Complexity

Cyclomatic Complexity (1)

- A measure of logical complexity
- Tells how many tests are needed to execute every statement of program

=Number of branches (`if`, `while`, `for`) + 1

Different tools tells you different values of Cyclomatic Complexity

- Original paper is not clear about how to derive the control flow graph
  - different implementations gives different values for the same code.
  - For example, the following code is reported with complexity 2 by the Eclipse Metrics Plugin, with 4 by GMetrics, and with complexity 5 by SonarQube:

```
int foo (int a, int b) {
        if (a > 17 && b < 42 && a+b < 55) {
                return 1;
        }
        return 2;
}
```

Recap: Cyclomatic Complexity (4)

- Number of predicates + 1
- Number of edges - number of nodes + 2
- Number of regions of the flow graph

Recap: Cyclomatic Complexity (5)
Testing view

- Cyclomatic complexity is the number of independent paths through the procedure
- Gives an upper bound on the number of tests necessary to execute every edge of control graph

Lab Exercise

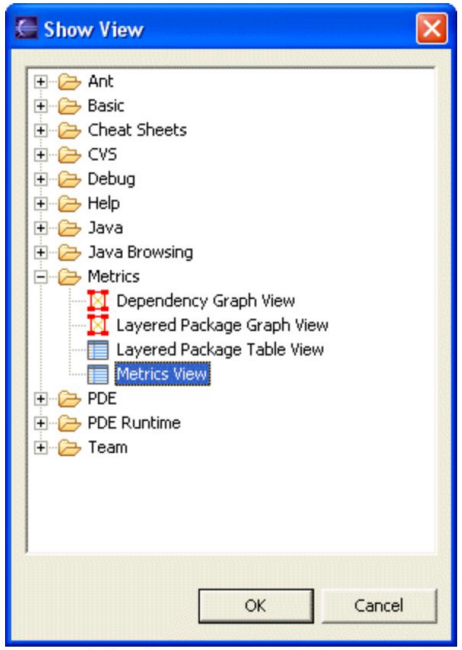
- Accept the invitation link: https://classroom.github.com/a/uD2YOLI2

- Install the Metrics plugin for either Eclipse or Intellij

Choice 1: Eclipse Metrics 2 Plugin

- Refer to: http://metrics2.sourceforge.net/
- Run Eclipse, go to Help menu -> Install New Software ... On the opening dialog click on the Add ... button. Add a new Remote site with the following url **http://metrics2.sourceforge.net/update/** and follow the instructions for installation.

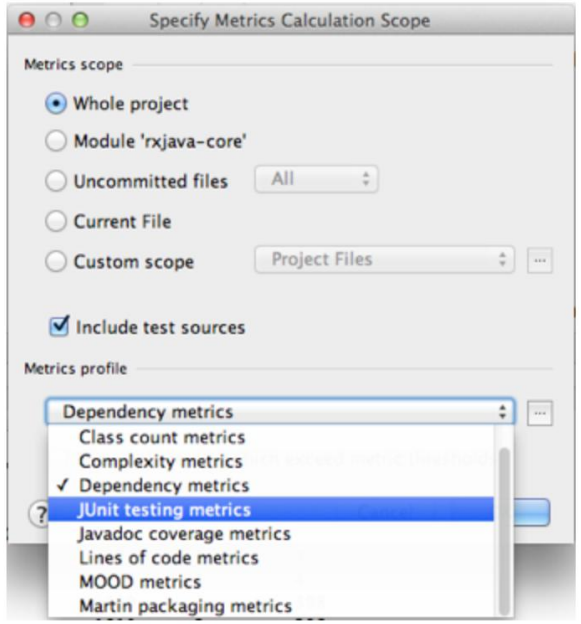
To start using the Metrics View, use Windows -> Show View -> Other and navigate to the Metrics View, as shown in the next image.



- Initially the resulting view will show a brief usage message because no metrics have been calculated yet. To start collecting metrics for a project, right click on the project and from the popup menu select "Metrics->Enable" (or alternatively, use the properties page)

Choice 2: Intellij MetricsReloaded Plugin

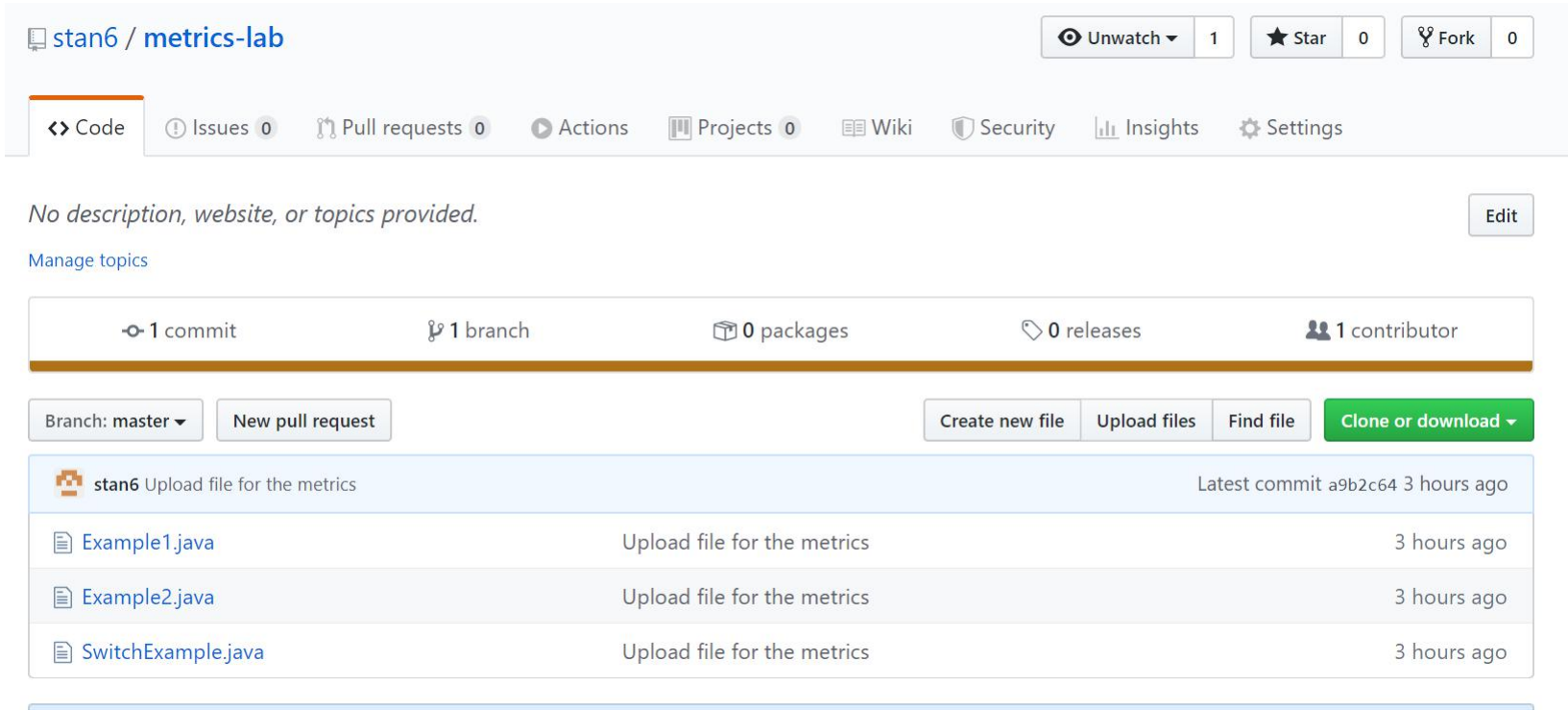
- Refer to: https://plugins.jetbrains.com/plugin/93-metricsreloaded to install the plugin
- Select the menu item Help | Find Action... and search for Calculate Metrics.... Try the Lines of code metrics profile first, if you haven't used MetricsReloaded before.



with a menu option under **Analyze|Calculate Metrics**. Metrics can be run for an ule, uncommitted files, current file, or a even a custom scope, making it extremely

Run the Metrics plugin on the example classes

- https://github.com/stan6/metrics-lab



Lab Exercise Part 1

- What is the Cyclomatic complexity for the class according to the plugin for the class below? Explain how to calculate the cyclomatic complexity (e.g., how many branches it have)
  1. SwitchExample.java
  2. Example1.java
  3. Example2.java
- Add a README.md with the following information:
  - Name:
  - Student id:
  - Answer the above question
  - Screenshot showing the results of the plugin

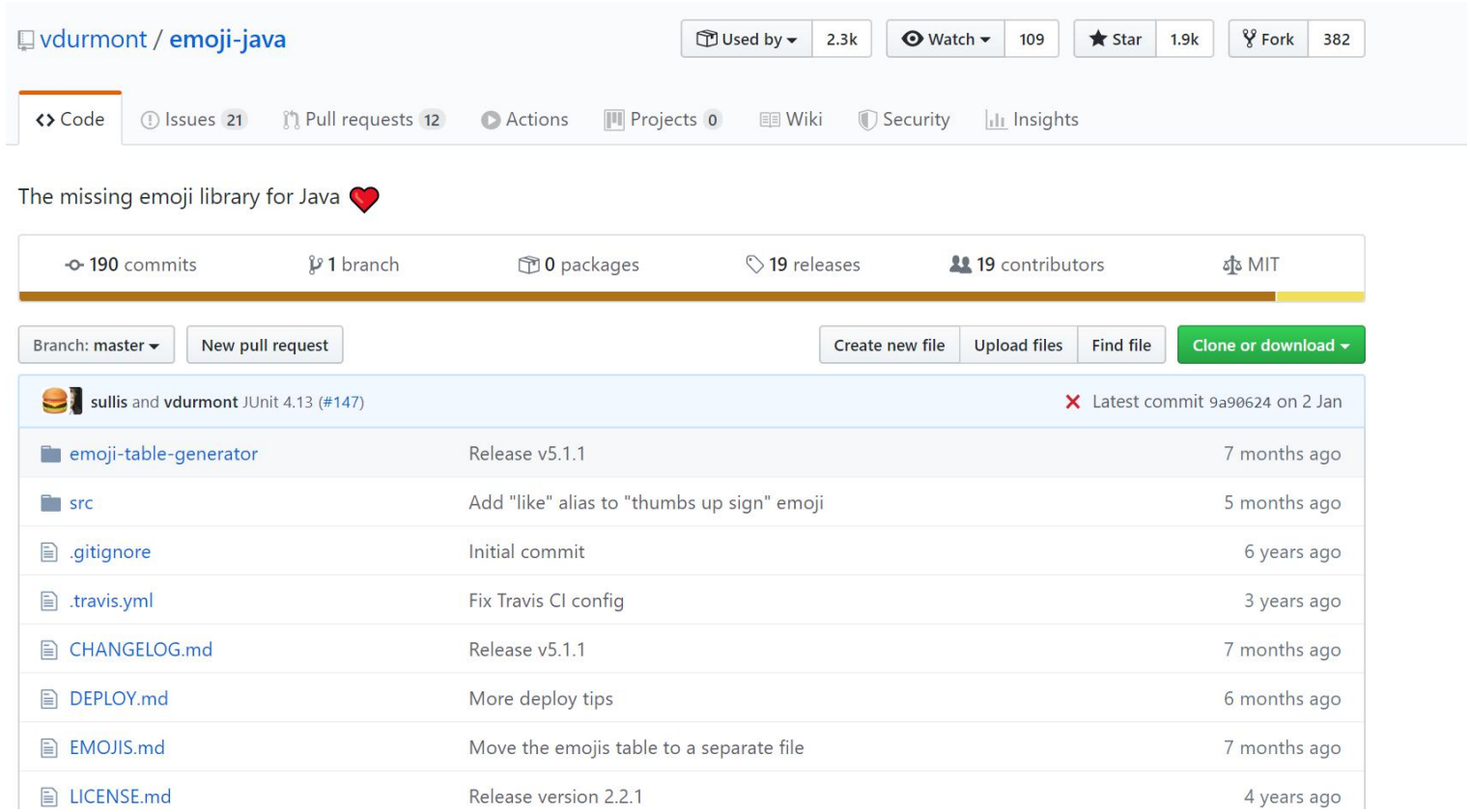
Lab Part 2: Project

Metrics for your project

Analyze Metrics for your project

My Example: https://github.com/vdurmont/emoji-java (A library for emoji)

Step 1: Import the project to your IDE (Eclipse/IntelliJ)
Step 2: Run the metrics on your project



Analyze your project: Lines of Code (LOC)?

Example: https://github.com/vdurmont/emoji-java (A library for emoji)

Look at the total lines of code:

- What is the total lines of code for your project?

Metric	Total	Mean	Std. Dev.	Maxim...	Resource causing Maximum
Number of Attributes (avg/max per type)	31	1.632	2.133	7	/emoji-java-master/src/main/j
Number of Static Attributes (avg/max per type)	6	0.316	1.126	5	/emoji-java-master/src/main/j
Number of Methods (avg/max per type)	111	5.842	9.224	38	/emoji-java-master/src/test/ja
Number of Static Methods (avg/max per type)	40	2.105	4.435	18	/emoji-java-master/src/main/j
Specialization Index (avg/max per type)		0.011	0.048	0.214	/emoji-java-master/src/main/j
Number of Classes (avg/max per packageFragment)	19	9.5	3.5	13	/emoji-java-master/src/main/j
Number of Interfaces (avg/max per packageFragment)	1	0.5	0.5	1	/emoji-java-master/src/main/j
Number of Packages	2				
Total Lines of Code	1419				
Method Lines of Code (avg/max per method)	874	5.788	5.287	31	/emoji-java-master/src/main/j

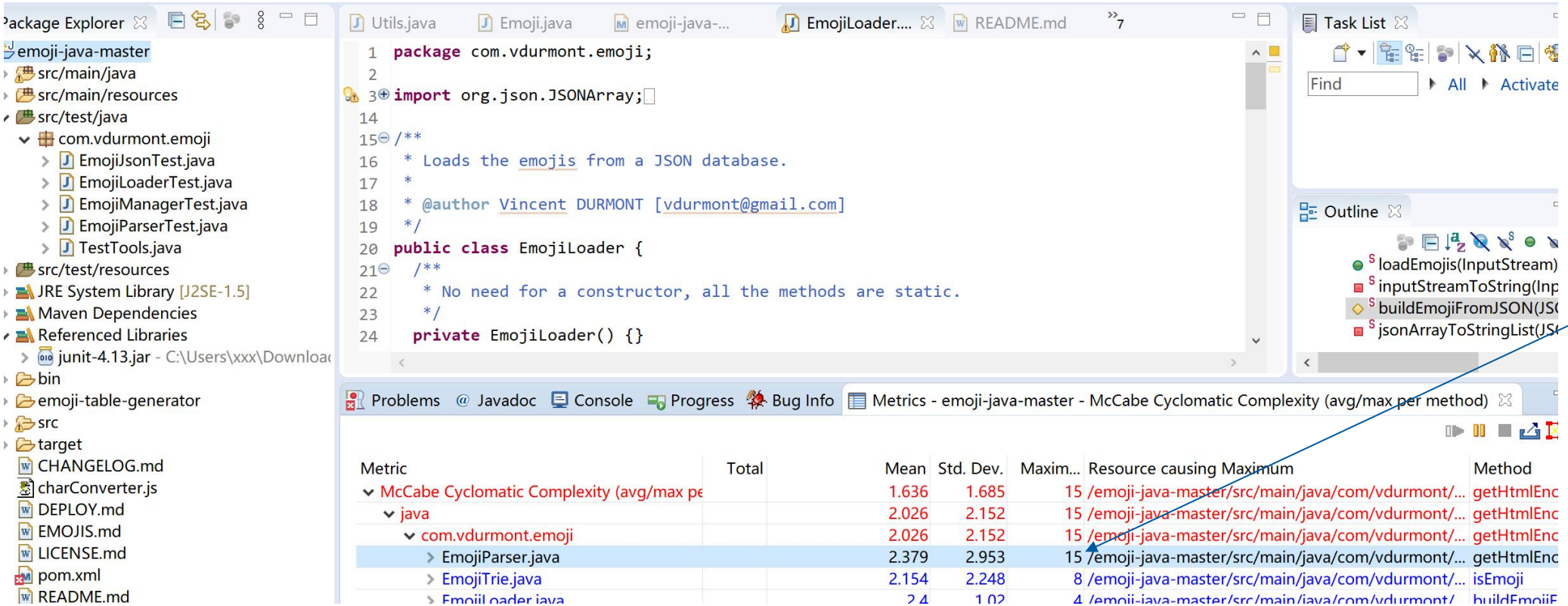
LOC is 1419 for my example

Analyze your project: Find the most complex class

Example: https://github.com/vdurmont/emoji-java (A library for emoji)

Look at the cyclometic complexity for each class:

- What is the most complex class in your project according to cyclometic complexity?



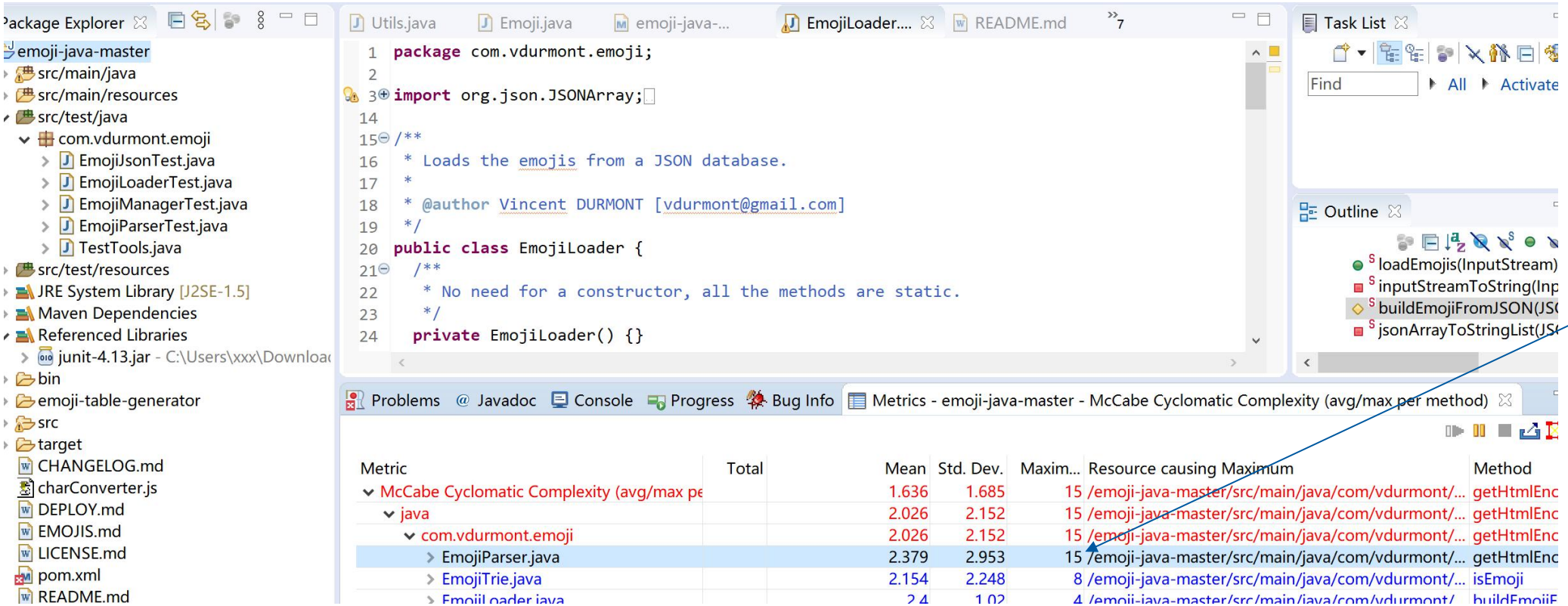
EmojiParser is the most complex class

Analyze your project: Find the most complex class

Example: https://github.com/vdurmont/emoji-java (A library for emoji)

Look at the cyclometic complexity for each class:

- What is the most complex class in your project according to cyclometic complexity?



EmojiParser is the most complex class

Cyclomatic Complexity (6): Metrics view

- McCabe found that modules with cyclomatic complexity greater than 10 were hard to test and error prone
- Check if your project has any method with cyclomatic complexity >10
- For example, EmojiParser.java has the following complex method

```
/** Finds the HTML encoded emoji in the given string starting at the given point, null otherwise */
 protected static AliasCandidate getHtmlEncodedEmojiAt(String input, int start) {
   …
 }
```

Why cyclomatic complexity is high for EmojiParser?

- High cyclomatic complexity because of (1) many branches, and (2) many complex conditions

```
protected static AliasCandidate getHtmlEncodedEmojiAt(String input, int start) {
    if (input.length() < start + 4 || input.charAt(start) != '&' || input.charAt(start + 1) != '#') return null;
    Emoji longestEmoji = null;
    int longestCodePointEnd = -1;
    char[] chars = new char[EmojiManager.EMOJI_TRIE.maxDepth];
    int charsIndex = 0;
    int codePointStart = start;
    do {
      int codePointEnd = input.indexOf(';', codePointStart + 3);  // Code point must be at least 1 char in length
      if (codePointEnd == -1) break;
      try {
        int radix = input.charAt(codePointStart + 2) == 'x' ? 16 : 10;
        int codePoint = Integer.parseInt(input.substring(codePointStart + 2 + radix / 16, codePointEnd), radix);
        charsIndex += Character.toChars(codePoint, chars, charsIndex);
      } catch (NumberFormatException e) {       break;
      } catch (IllegalArgumentException e) {       break;      }
      Emoji foundEmoji = EmojiManager.EMOJI_TRIE.getEmoji(chars, 0, charsIndex);
      if (foundEmoji != null) {
        longestEmoji = foundEmoji;
        longestCodePointEnd = codePointEnd;
      }
      codePointStart = codePointEnd + 1;
    } while (input.length() > codePointStart + 4 && input.charAt(codePointStart) == '&' &&
        input.charAt(codePointStart + 1) == '#' && charsIndex < chars.length &&
          !EmojiManager.EMOJI_TRIE.isEmoji(chars, 0, charsIndex).impossibleMatch());
    if (longestEmoji == null) return null;
    return new AliasCandidate(longestEmoji, null, start, longestCodePointEnd); }
```

Lab Exercise Part 2

- Answer the following questions by modifying the README.md that you have created in Lab Exercise part 1
  - Part 2: Metrics for my project
    - What is the Lines of Codes of your selected project? (If you selected two projects, you just need to select one of the projects to answer this question)
    - What is the maximum Cyclomatic Complexity of the classes of your project?
    - Do your project has any method with cyclomatic complexity >10? If yes, explain why the cyclomatic complexity is high for your project.

*You don't need to include any screenshot for this part

*You don't need to upload your project to the invitation link. Only need to answer questions in README.md

What to submit?

- README.md
  - Student name and student id
  - Answer for Part 1 (for the starter code)
  - Answer for Part 2 (for your project)