Aufgabe 1b)

CheckStyle:

- Da CheckStyle lediglich die Programmstruktur analysiert können hier nicht sehr leicht false positives bzw. false negatives gefunden werden. Insbesondere ist es schwierig false negatives zu finden, da der festgelegte Programmierstil in CheckStyle für solche Fehler eine sehr geringe Anfälligkeit besitzt. Ein Beispiel hierfür wäre ein mit Leerzeichen eingerückter Codeabschnitt. Ist das Codefragment zwar an der richtigen Stelle, jedoch lediglich mit Leerzeichen eingerückt, könnte CheckStyle hier eine Fehlermeldung geben, da der Algorithmus die Implementierung fälschlicherweise für falsch ansieht, da hier mit einem TAB eingerückt werden sollte. Ein false positive könnte vorliegen, falls die Anforderungen für den Programmierstil nicht konkret festgelegt werden und somit kein eindeutiges Analyseverfahren vorhanden ist. Dies impliziert mögliche false positives, da Teile des Programmiercodes (bspw. korrekte Methodenkopfdefinition) dem letzteren Kriterium des Prüfalgorithmus entsprechen und somit eine vorherig als falsche anerkannte Formatierung des Methodenkopfes plötzlich als richtig akzeptiert wird.

JDepend:

- Leider ist es uns die Arbeit mit JDepend schwer gefallen. Da wir alle die aktuellste Version von Eclipse benutzen (Neon M3 Package - http://www.eclipse.org/downloads/packages/release/Neon/M3). Da JDepend die Programmstruktur als solches und nicht isolierte Syntax- oder Stilprobleme wie die anderen Probleme löst, ist es ebenfalls schwierig wie bei CheckStyle false positives bzw. false negatives zu finde. Daher betrachten wir hier die Priorisierung der einzelnen Packages im Hinblick auf die Abstraktheit und Instabilität. Ein false positive wäre unserer Meinung nach eine falsche Interpretation von JDepend im Hinblick auf auf den Zusammenhang zwischen Abstraktheit und Instabilität, sodass ein Package zu hoch eingestuft wird. Ein false negative wäre dementsprechend eine zu geringe Priorisierung des Packages.

Aufgabe 2a + b)

CheckStyle:

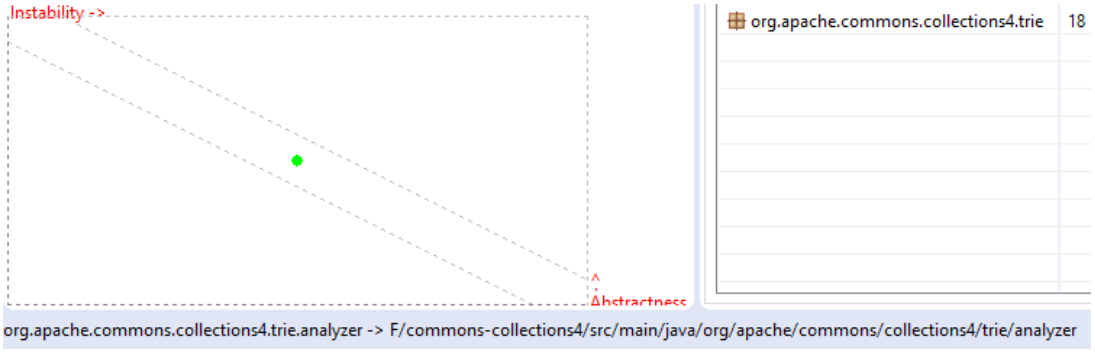
- Projekt: commons-deamon → src/main/java → org.apache.commons.daemon.support → DaemonConfiguration.java
- Line 52:
  - “{“ steht in der falschen Zeile.
  - Dies könnte behoben warden, indem man die geschweifte Klammer in die richtige Zeile einrückt.
- Line 56:
  - Multiple markers at this line
    - member def modifier bei Einrücktiefe 4 nicht an korrekter Einrücktiefe 2
    - Modifier 'static' weicht von der empfohlenen Modifier-Reihenfolge aus der Java-Sprachdefinition ab.

- Die Berichtigung dieser Fehlermeldung würde entsprechend der Beschreibung durch korrektes Einrücken und empfohlener Verwendung vom Modifier static erfolgen.

- Line 107:
  - Kind von method def bei Einrücktiefe 8 nicht an korrekter Einrücktiefe 4
  - Korrekte Einrücktiefe anpassen (von 8 auf 4)

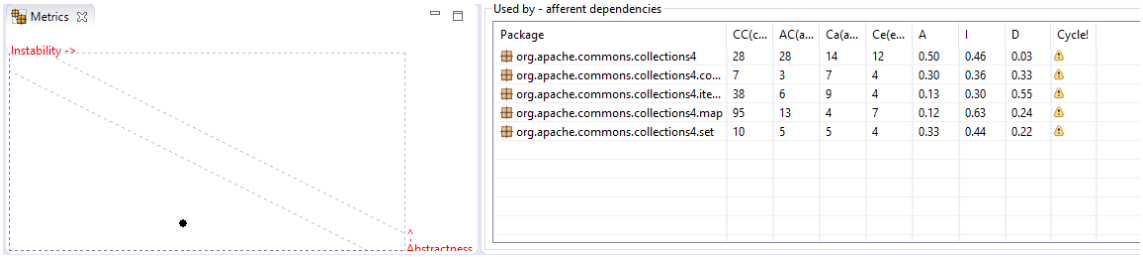
JDepend

- Da JDepend nicht einwandfrei auf unseren Rechnern funktioniert konnten wir hier leider gemäß Aufgabe 2a keine Fehler finden und gemäß 2b keine Verbesserungsvorschläge. Würde JDepend korrekterweise die Zyklen zwischen den Paketen anzeigen. Wäre hier unser Ansatz Beispiele zu geben wie man diese Zyklen auflösen kann um somit Performanz des Codes zu steigern.
- Im Folgenden geben wir Beispiele für 3 Fälle (Projekt: commons-collections4):
  1. Ausgeglichen:



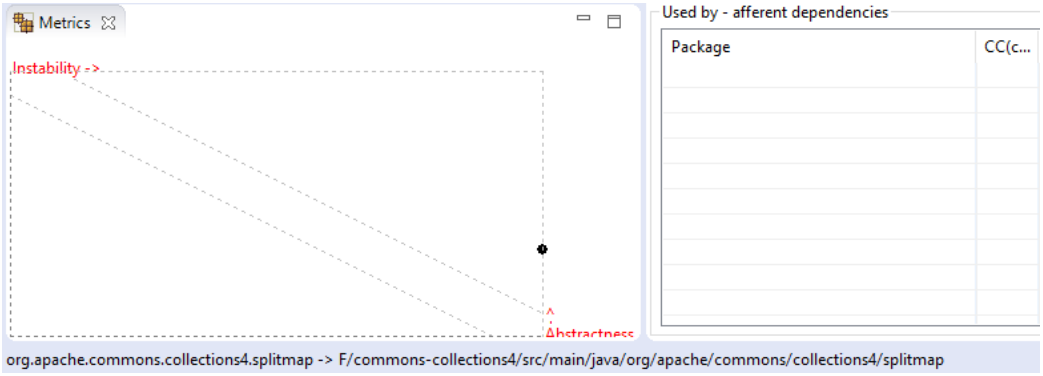
Hier ist eine geringe Anzahl an Zyklen vorhanden und somit Stabilität zur Laufzeit gewährt. Es handelt sich außerdem nicht um ein false negative oder positive.

  2. Nicht Ausgeglichen (linkslastig) – org.apache.commons.collections4.list



Ziel wäre Abhängigkeiten zu reduzieren und den Punkt in Richtung Annahmebereich zu verschieben, dies könnte durch lediglich Verändern der Instabilität oder der Abstractness erfolgen. Man könnte auch beide Komponenten adjustieren um in den Annahmebereich zu gelangen. Somit würde man Zyklen minimieren und die Stabilität zur Laufzeit erhöhen. Es handelt sich hierbei erneut nicht um ein false negative bzw. positive.

  3. Nicht Ausgeglichen (rechtslastig) – org.apache.commons.collections4.splitmap



Um hier Stabilität zur Laufzeit zu gewähren, wollte man Instabilität reduzieren und die Anzahl der afferent dependencies erhöhen. Auffälig ist, dass die meisten Packages sich links vom Ablehnungsbereich finden. Die Wahrscheinlichkeit auf ein false positive bzw. false negative ist demnach hier sehr hoch. Jedoch konnten wir die beiden Fälle auch hier ausschließen, da die Abhängigkeiten korrekt analysiert wurden.