Software Architecture (SA)

- A good SA is essential for easy software maintenance (modification of the software product after delivery to correct faults, to improve performance or other attributes) and adding new features.
- Our focus: the aspect of SA that deals with architecture of the modules (packages and classes) and their interconnections.

Motivating Questions….

- How difficult can (even a simple) a code change be?
  - Does it cause a cascade of subsequent changes in dependent modules? (**Rigidity**)
  - Does it break software in multiple places – even in areas not conceptually related to the original code change? (**Fragility**)
- How easy is it to reuse a software module?
  - What is the overhead in separating the desirable parts from undesirable parts? Is the overhead so much that rewriting the software would be easier? (**Immobility**)
- Is the right way the hard way?
  - Do the design constraints make it hard to follow the right way, and instead an improperly designed "hack" is easy to implement. (**Viscosity**)

References

- Software Package Metrics
  - Design Principles and Design Patterns, Robert C. Martin[1], 2000. Article available at: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
  - Wikipedia: http://en.wikipedia.org/wiki/Software_package_metrics
  - Book "**Agile Software Development:** Principles, Patterns, and Practices": http://books.google.com/books/about/Agile_Software_Development.html?id=0HYhAQAAIAAJ

[1]Martin is a software professional since 1970 and an international software consultant since 1990. In 2001, he initiated the meeting of the group that created agile software development from extreme programming techniques.
See http://en.wikipedia.org/wiki/Robert_Cecil_Martin

Package Coupling Principles

- Applications are networks of interrelated packages.
- The interrelationship should obey certain principles for efficient software maintenance.
  - "Package Coupling Principles"
- We capture packages interrelations by package dependency metrics.
- Package X is dependent on package Y if X cannot be compiled without Y.

Package Coupling Principles

- Package X is dependent on package Y if
  - Some code in X read/writes into a field declared in some class in Y
  - A class in X extends/implements a class in Y
  - Some code in X calls a method in Y

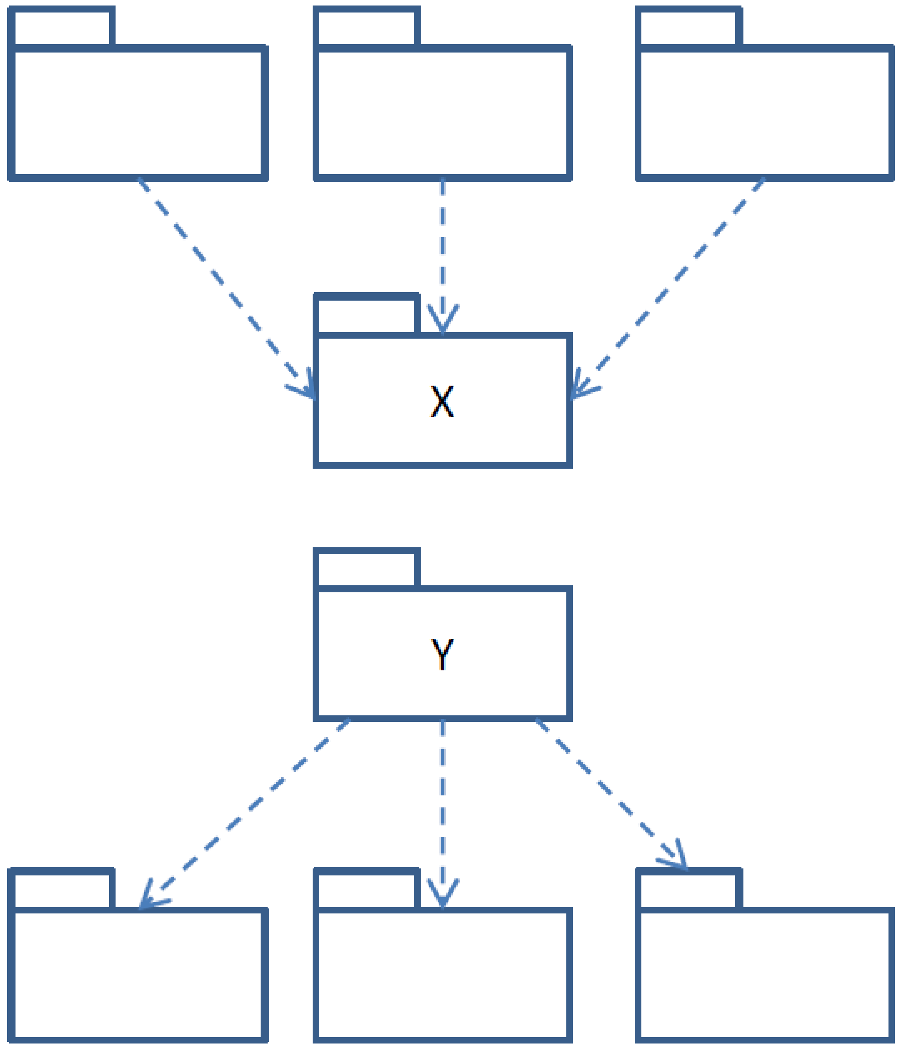
The Acyclic Dependencies Principle

- *The dependencies between packages must not form cycles.*
  - Code-changes are usually localized to a package because a good design groups related classes together in a package. The changed package should be compiled and tested with the packages it depends on. If there is a cyclic dependency, a change in any one package in the cycle requires the test suite to be built with all the packages in the cycle – clearly not a desirable thing.

The Stable Dependencies Principle (SDP)

- A package should be stable if it has many packages dependent on it.
  - Because it requires a great deal of work to reconcile any changes in the stable package with all the dependent packages.
  - For such a package should be stable, it should not be heavily dependent on other packages.

Dependent vs. Independent Packages

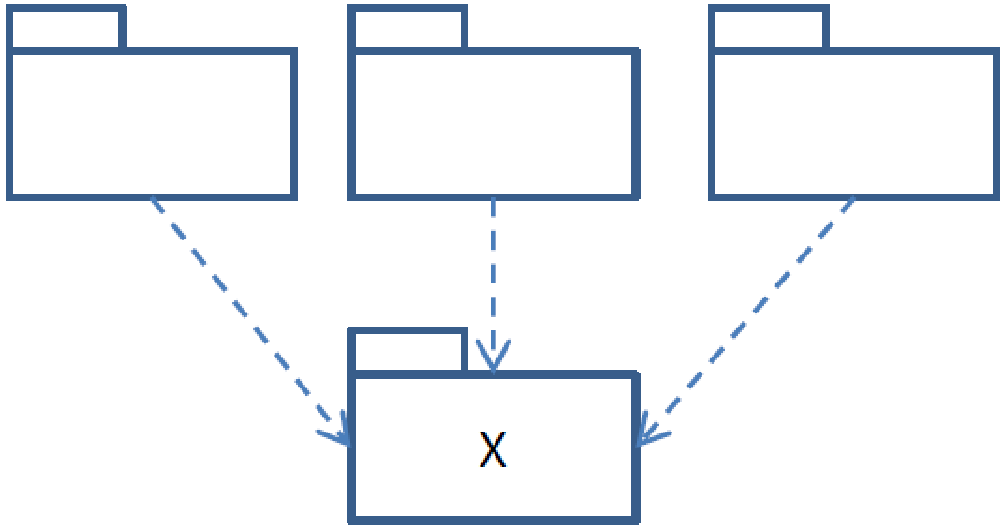


- X is independent
  - It has three packages dependent on it.
  - X depends upon nothing, so it has no external influence to make it change. We say it is *independent.*
- Y is dependent
  - It has no packages depending on it.
  - Y depends on three packages, so changes may come from three external sources. We say that Y is *dependent*.

Stability Metrics

- Efferent Coupling (Ce): The number of classes outside the package that the classes inside the package depend upon (outgoing dependencies).
- Afferent Coupling (Ca): The number of classes outside the package that depend upon the classes inside the package (incoming dependencies).
- Instability (I): Ce/(Ca+Ce)
  - Range [0,1]; 0: very stable, 1: very unstable.
- SDP can be rephrased as: “Depend upon packages whose instability metric is lower than yours.
- We can envision the packages structure of our application as a set of interconnected packages with unstable packages at the top, and stable packages on the bottom. In this view, all dependencies point downwards.

The Stable Abstractions Principle (SAP)

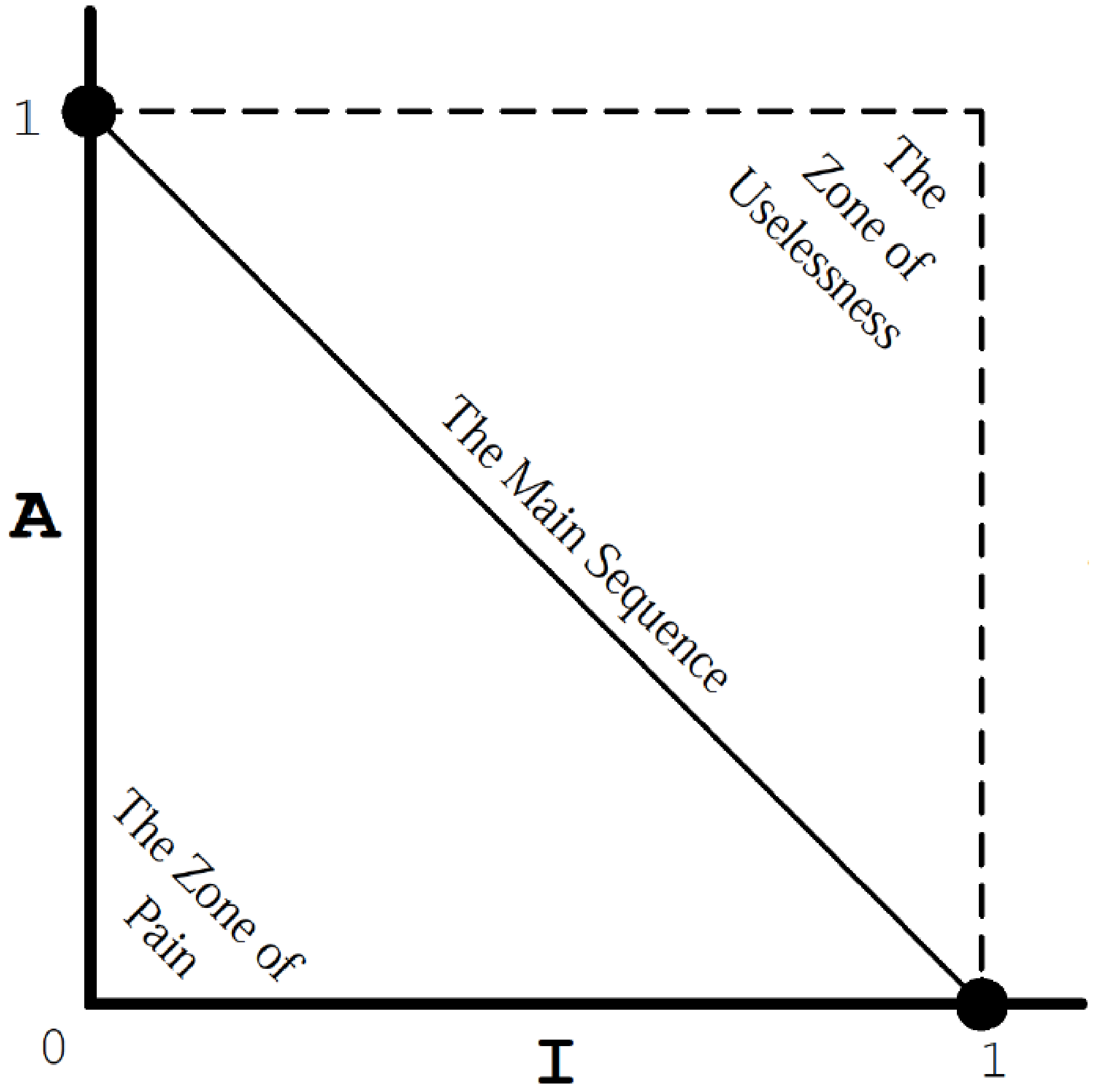


- SAP: Heavily used packages should have a high-degree of abstraction.
- A high-degree of abstraction implies that packages are easy to change or extend.

Abstractness Metrics

- *Nc:* Number of classes in the package.
- *Na:* Number of abstract classes in the package.
- Abstractness (A) = *Na/Nc*
  - Range [0,1]; 0: no abstract classes, 1: all classes are abstract
- SAP can be restated as: I should increase as abstractness decreases. That is, concrete packages should be unstable (flexible) while abstract packages should be stable.

Abstractness vs Instability



- Ideally a package should lie on "The Main Sequence" line
- Distance (D): |A+I-1|
  - Range [0,1]; 0: package directly on the main sequence, 1: farthest away from the main sequence.

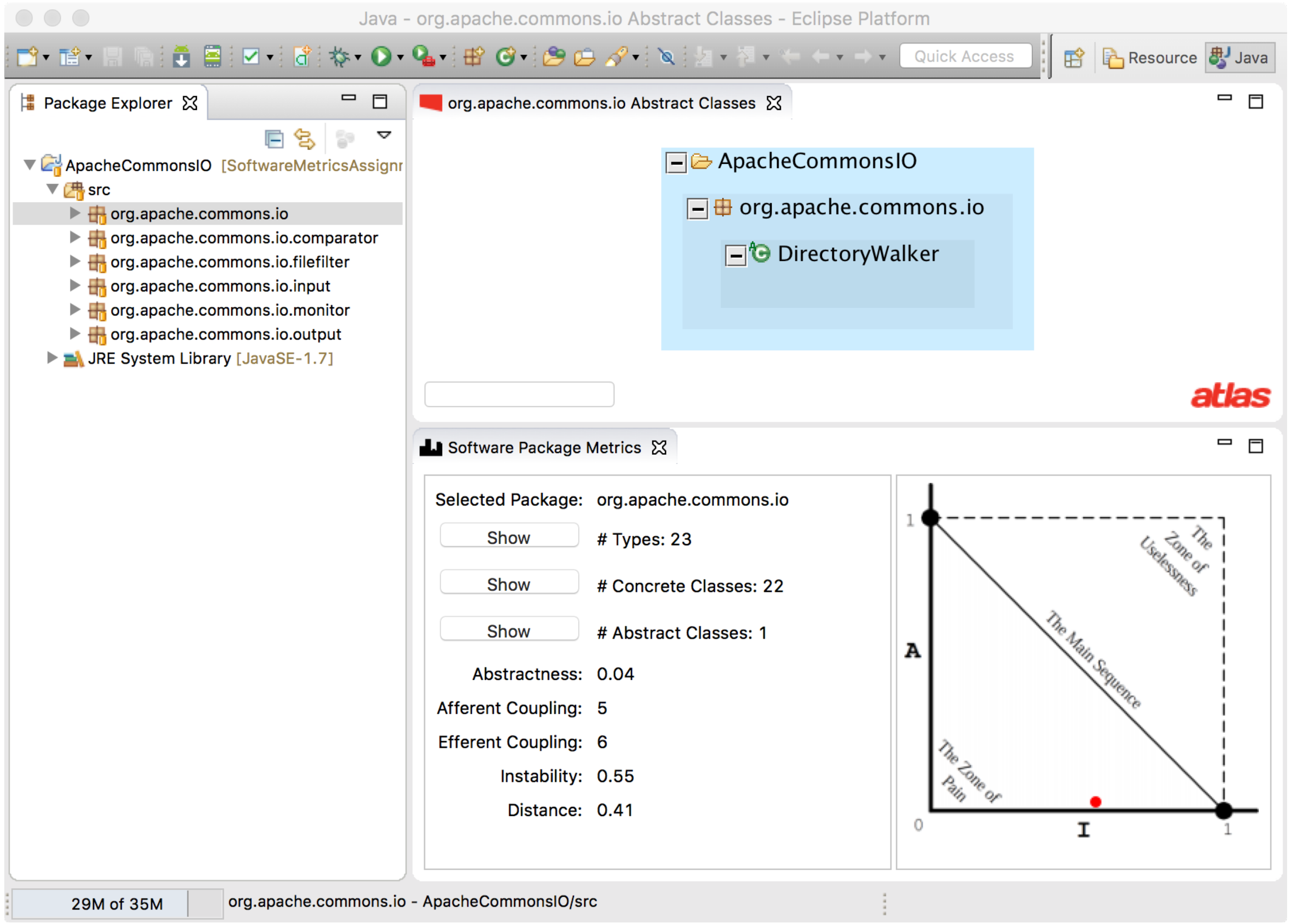
Reflections on Software Metrics

- These metrics are meant to provide a measure of how good the software architecture is.
- But, they are imperfect, and reliance upon them as the sole indicator of a sturdy architecture would be foolhardy.

Classroom Activity

- Develop a program analysis tool to compute the previously described software metrics
  - Implement TODOs in base project: https://github.com/benjholla/SoftwareMetricsAssignment
- Evaluate the Apache Commons IO library

Classroom Activity



Java - org.apache.commons.io Abstract Classes - Eclipse Platform

Quick Access
Resource
Java
Package Explorer
ApacheCommonsIO [SoftwareMetricsAssignr
src
org.apache.commons.io
org.apache.commons.io.comparator
org.apache.commons.io.filefilter
org.apache.commons.io.input
org.apache.commons.io.monitor
org.apache.commons.io.output
JRE System Library [JavaSE-1.7]
org.apache.commons.io Abstract Classes
ApacheCommonsIO
org.apache.commons.io
DirectoryWalker
atlas
Software Package Metrics
Show
Show
Show
1
A
0
I
1
The Zone of Uselessness
The Main Sequence
The Zone of Pain
29M of 35M

Selected Package: org.apache.commons.io

Types: 23

Concrete Classes: 22

Abstract Classes: 1

Abstractness:	0.04
Afferent Coupling:	5
Efferent Coupling:	6
Instability:	0.55
Distance:	0.41

org.apache.commons.io - ApacheCommonsIO/src

Classroom Activity

- With regard to the Distance (D) from the Main Sequence is the org.apache.commons.io package better or worse compared to the org.apache.commons.io.filefilter package?
- Do you feel these metrics accurately capture the quality of this code?

Classroom Activity: Solution

```
public static Q getClasses(Q graph){
    return graph.nodesTaggedWithAny(XCSG.Java.AbstractClass, XCSG.Java.Class);
}

public static Q getInterfaces(Q graph){
    return graph.nodesTaggedWithAll(XCSG.Java.Interface);
}

public static Q getAbstractClasses(Q graph){
    return graph.nodesTaggedWithAny(XCSG.Java.AbstractClass, XCSG.Java.Interface);
}

public static Q getConcreteClasses(Q graph) {
    return getClasses(graph).difference(getAbstractClasses(graph));
}
```

Classroom Activity: Solution

```
public static double getAbstractness(Node pkg) throws ArithmeticException {
    // Step 1) Get the types declared under the package
    Q packageTypes = getPackageTypes(pkg);
    // Step 2) From the discovered class nodes select the abstract classes
    Q abstractClasses = getAbstractClasses(packageTypes);
    // Step 4) Return the ratio of the number of abstract classes (and interfaces) in the package to the total number of types in the package
    return (double) countNodes(abstractClasses) / (double) countNodes(packageTypes);
}
```

Classroom Activity: Solution

```
public static Q getAfferentCouplings (Node pkg){
    // Step 1) Get the methods declared under the package
    Q packageMethods = getPackageMethods(pkg);
    // Step 2) Create a subgraph of CALL edges from the universe
    Q callEdges = Common.universe().edgesTaggedWithAny(XCSG.Call).retainEdges();
    // Step 3) Within the calls subgraph get the predecessors of the package methods (calling methods)
    Q callingMethods = callEdges.predecessors(packageMethods);
    // Step 4) Return the packages of the calling methods
    return getPackagesOfProgramElements(callingMethods);
}
```

Classroom Activity: Solution

```
public static Q getEfferentCouplings (Node pkg){
    // Step 1) Get the methods declared under the package
    Q packageMethods = getPackageMethods(pkg);
    // Step 2) Create a subgraph of CALL edges from the universe
    Q callEdges = Common.universe().edgesTaggedWithAny(XCSG.Call).retainEdges();
    // Within the calls subgraph get the successors of the package methods (called methods)
    Q calledMethods = callEdges.successors(packageMethods);
    // Step 4) Return the packages of the called methods
   return getPackagesOfProgramElements(calledMethods);
}
```

Classroom Activity: Solution

```
public static double getInstability(Node pkg) {
    double ce = (double) countNodes(getEfferentCouplings(pkg));
    double ca = (double) countNodes(getAfferentCouplings(pkg));
    return ce / (ca + ce);
}

public static double getDistance(Node pkg) {
    double a = getAbstractness(pkg);
    double i = getInstability(pkg);
     return Math.abs(a + i - 1);
}
```

Classroom Activity: Solution

Package	Types	Concrete Classes	Abstract Classes	Abstractness	Afferent Coupling	Efferent Coupling	Instability	Distance	Properties
org.apache.commons.io	23	22	1	0.04	5	6	0.55	0.41	Least Abstract
org.apache.commons.io.comparator	10	9	1	0.1	4	3	0.43	0.47	
org.apache.commons.io.filefilter	25	22	3	0.12	3	6	0.67	0.21	Least Stable
org.apache.commons.io.input	27	24	3	0.11	6	3	0.33	0.56	Most Stable
org.apache.commons.io.monitor	5	4	1	0.2	3	4	0.57	0.23	Most Abstract
org.apache.commons.io.output	19	18	1	0.05	4	3	0.43	0.52	

- With regard to the Distance (D) from the Main Sequence is the org.apache.commons.io package better or worse compared to the org.apache.commons.io.filefilter package?
  - According to the metric, the distance D from the main sequence indicates org.apache.commons.io is worse designed than org.apache.commons.io.filefilter, since the org.apache.commons.io.filefilter package is significantly closer to 0 than the org.apache.commons.io package.

Classroom Activity: Solution

Package	Types	Concrete Classes	Abstract Classes	Abstractness	Afferent Coupling	Efferent Coupling	Instability	Distance	Properties
org.apache.commons.io	23	22	1	0.04	5	6	0.55	0.41	Least Abstract
org.apache.commons.io.comparator	10	9	1	0.1	4	3	0.43	0.47	
org.apache.commons.io.filefilter	25	22	3	0.12	3	6	0.67	0.21	Least Stable
org.apache.commons.io.input	27	24	3	0.11	6	3	0.33	0.56	Most Stable
org.apache.commons.io.monitor	5	4	1	0.2	3	4	0.57	0.23	Most Abstract
org.apache.commons.io.output	19	18	1	0.05	4	3	0.43	0.52	

- Do you feel these metrics accurately capture the quality of this code?
  - Depends. Code metrics are an approximation that capture some properties of the software. Keep in mind that Apache Commons IO is a library. What would happen if we consider the libraries interactions with the JDK? What would happen if we consider the libraries interactions with an application using the library?