Object Oriented Design Quality Metrics

References

- Analyze java package metrics in a graph database
- Calculate metrics
- jqassistant
- notebook walks through examples for integrating various packages with Neo4j
- OO Design Quality Metrics
- Neo4j Python Driver

Incoming Dependencies

Incoming dependencies are also denoted as "Fan-in", "Afferent Coupling" or "in-degree". These are the ones that use the listed package.

If these packages get changed, the incoming dependencies might be affected by the change. The more incoming dependencies, the harder it gets to change the code without the need to adapt the dependent code (“rigid code”). Even worse, it might affect the behavior of the dependent code in an unwanted way (“fragile code”).

Since Java Packages are organized hierarchically, incoming dependencies can be count for every package in isolation or by including all of its sub-packages. The latter one is done without top level packages like for example "org" or "org.company" by assuring that only packages are considered that have other packages or types in the same hierarchy level ("siblings").

Table 1a

- Show the top 20 Java Packages with the most incoming dependencies
- Set the "incomingDependencies" properties on Package nodes.

artifactName	fullQualifiedPackageName	packageName	incomingDependencies	incomingDependenciesWeight	incomingDependentTypes	incomingDependentInterfa

Table 1b

- Show the top 20 Java Packages including their sub-packages with the most incoming dependencies
- Set the property "incomingDependenciesIncludingSubpackages" on Package nodes.

artifactName	fullQualifiedPackageName	packageName	incomingDependencies	incomingDependenciesWeight	incomingDependentTypes	incomingDependentInterfa

Table 1c

- Show the top 20 Typescript modules with the most incoming dependencies
- Set the property "incomingDependencies" on Module nodes if not already done.

	fullQualifiedModuleName	moduleName	incomingDependencies	incomingDependenciesWeight	incomingDependentAbstractTypes	incomingDependentAbstractTypeV
0	/home/runner/work/code-graph-analysis-pipeline...	router	21	42	0	
1	/home/runner/work/code-graph-analysis-pipeline...	react-router	19	42	0	
2	/home/runner/work/code-graph-analysis-pipeline...	react-router-dom	2	3	0	
3	/home/runner/work/code-graph-analysis-pipeline...	react-router-dom-v5-compat	0	0	0	
4	/home/runner/work/code-graph-analysis-pipeline...	server.tsx	0	0	0	
5	/home/runner/work/code-graph-analysis-pipeline...	react-router-native	0	0	0	

Outgoing Dependencies

Outgoing dependencies are also denoted as "Fan-out", "Efferent Coupling" or "out-degree". These are the ones that are used by the listed package.

Code from other packages and libraries you’re depending on (outgoing) might change over time. The more outgoing changes, the more likely and frequently code changes are needed. This involves time and effort which can be reduced by automation of tests and version updates. Automated tests are crucial to reveal updates, that change the behavior of the code unexpectedly (“fragile code”). As soon as more effort is required, keeping up becomes difficult (“rigid code”). Not being able to use a newer version might not only restrict features, it can get problematic if there are security issues. This might force you to take “fast but ugly” solutions into account which further increases technical dept.

Since Java Packages are organized hierarchically, outgoing dependencies can be count for every package in isolation or by including all of its sub-packages. The latter one is done without top level packages like for example "org" or "org.company" by assuring that only packages are considered that have other packages or types in the same hierarchy level ("siblings").

Table 2a

- Show the top 20 Java Packages with the most outgoing dependencies
- Set the "outgoingDependencies" properties on Package nodes.

artifactName	fullQualifiedPackageName	packageName	outgoingDependencies	outgoingDependenciesWeight	outgoingDependentTypes	outgoingDependentInterface

Table 2b

- Show the top 20 Java Packages including their sub-packages with the most outgoing dependencies
- Set the property "outgoingDependenciesIncludingSubpackages" on Package nodes.

artifactName	fullQualifiedPackageName	packageName	outgoingDependencies	outgoingDependenciesWeight	outgoingDependentTypes	outgoingDependentInterface

Table 2c

- Show the top 20 Typescript modules with the most outgoing dependencies
- Set the "outgoingDependencies" properties on Module nodes if not already done

	fullQualifiedModuleName	sourceName	outgoingDependencies	outgoingDependenciesWeight	outgoingDependentAbstractTypes	outgoingDependentAbstractTypeWe
0	/home/runner/work/code-graph-analysis-pipeline...	react-router-dom	145	362	0	
1	/home/runner/work/code-graph-analysis-pipeline...	server.tsx	40	62	0	
2	/home/runner/work/code-graph-analysis-pipeline...	react-router-native	24	40	0	
3	/home/runner/work/code-graph-analysis-pipeline...	react-router	14	24	0	
4	/home/runner/work/code-graph-analysis-pipeline...	react-router-dom-v5-compat	0	0	0	
5	/home/runner/work/code-graph-analysis-pipeline...	router	0	0	0	

Instability

$$Instability = \frac{Outgoing\ Dependencies}{Outgoing\ Dependencies + Incoming\ Dependencies}$$

Instability is expressed as the ratio of the number of outgoing dependencies of a module (i.e., the number of packages that depend on it) to the total number of dependencies (i.e., the sum of incoming and outgoing dependencies).

Small values near zero indicate low *Instability*. With no outgoing but some incoming dependencies the Instability is zero which is denoted as maximally stable. Such code units are more rigid and difficult to change without impacting other parts of the system. If they are changed less because of that, they are considered stable.

Conversely, high values approaching one indicate high *Instability*. With some outgoing dependencies but no incoming ones the *Instability* is denoted as maximally unstable. Such code units are easier to change without affecting other modules, making them more flexible and less prone to cascading changes throughout the system. If they are changed more often because of that, they are considered unstable.

Since Java Packages are organized hierarchically, *Instability* can be calculated for every package in isolation or by including all of its sub-packages.

Table 3a

- Show the top 20 Java Packages with the lowest *Instability*

- Set the property "instability" on Package nodes.

artifactName	fullQualifiedPackageName	packageName	instability	instabilityTypes	instabilityInterfaces	instabilityPackages	instabilityArtifacts	p.outgoingDepende

Table 3b

- Show the top 20 Java Packages including their sub-packages with the lowest *Instability*
- Set the property "instabilityIncludingSubpackages" on Package nodes.

artifactName	fullQualifiedPackageName	packageName	instability	instabilityTypes	instabilityInterfaces	instabilityPackages	instabilityArtifacts	p.outgoingDepende

Table 3c

- Show the top 20 Typescript modules with the lowest *Instability*
- Set the property "instability" on Module nodes if not already done

	projectName	fullQualifiedModuleName	moduleName	instability	instabilityAbstractTypes	instabilityModules	instabilityPackages	module.outgoingDependencies
0	react-router-dom-v5-compat	/home/runner/work/code-graph-analysis-pipeline...	react-router-dom-v5-compat	0.000000	0.0	0.000000	0.000000	0
1	router	/home/runner/work/code-graph-analysis-pipeline...	router	0.000000	0.0	0.000000	0.000000	0
2	react-router	/home/runner/work/code-graph-analysis-pipeline...	react-router	0.424242	0.0	0.625000	0.333333	14
3	react-router-dom	/home/runner/work/code-graph-analysis-pipeline...	react-router-dom	0.986395	0.0	0.909091	0.666667	145
4	react-router-dom	/home/runner/work/code-graph-analysis-pipeline...	server.tsx	1.000000	0.0	1.000000	1.000000	40
5	react-router-native	/home/runner/work/code-graph-analysis-pipeline...	react-router-native	1.000000	0.0	1.000000	1.000000	24

Abstractness

$$Abstractness = \frac{abstract\ classes\ in\ category}{total\ number\ of\ classes\ in\ category}$$

Package *Abstractness* is expressed as the ratio of the number of abstract classes and interfaces to the total number of classes of a package.

Zero *Abstractness* means that there are no abstract types or interfaces in the package. On the other hand, a value of one means that there are only abstract types.

Since Java Packages are organized hierarchically, *Abstractness* can be calculated for every package in isolation or by including all of its sub-packages.

Table 4a

- Show the top 30 packages with the lowest *Abstractness*
- Set the property "abstractness" on Package nodes.

artifactName	fullQualifiedPackageName	packageName	abstractness	numberAbstractTypes	numberTypes

Table 4b

- Show the top 30 packages with the highest *Abstractness* and number of Java Types

artifactName	fullQualifiedPackageName	packageName	abstractness	numberAbstractTypes	numberTypes

Table 4c

- Show the top 30 packages including their sub-packages with the highest package depth and lowest *Abstractness*
- Set the property "abstractnessIncludingSubpackages" on Package nodes.

artifactName	fullQualifiedPackageName	packageName	abstractness	numberAbstractTypes	numberTypes	numberOfIncludedSubPackages	maxSubpackageDepth

Table 4d

- Show the top 30 packages including their sub-packages with the highest package depth and highest *Abstractness*

artifactName	fullQualifiedPackageName	packageName	abstractness	numberAbstractTypes	numberTypes	numberOfIncludedSubPackages	maxSubpackageDepth

Table 4e

- Show the top 30 Typescript modules with the lowest *Abstractness*
- Set the property "abstractness" on Module nodes if not already done.

	projectName	fullQualifiedModuleName	moduleName	abstractness	numberAbstractTypes	numberTypes
0	react-router-dom-v5-compat	/home/runner/work/code-graph-analysis-pipeline...	react-router-dom-v5-compat	0.000000	None	None
1	router	/home/runner/work/code-graph-analysis-pipeline...	router	0.000000	None	None
2	react-router-dom	/home/runner/work/code-graph-analysis-pipeline...	server.tsx	0.333333	None	None
3	react-router-dom	/home/runner/work/code-graph-analysis-pipeline...	react-router-dom	0.352941	None	None
4	react-router-native	/home/runner/work/code-graph-analysis-pipeline...	react-router-native	0.384615	None	None
5	react-router	/home/runner/work/code-graph-analysis-pipeline...	react-router	0.600000	None	None

Distance from the main sequence

The *main sequence* is a imaginary line that represents a good compromise between *Abstractness* and *Instability*. A high distance to this line may indicate problems. For example is very *stable* (rigid) code with low abstractness hard to change.

Read more details on that in OO Design Quality Metrics and Calculate metrics.

Table 5a

- Show the top 30 packages with the highest distance from the "main sequence"

artifactName	fullQualifiedName	name	distance	abstractness	instability	elementsCount

Table 5b

- Show the top 30 packages including their sub-packages with the highest distance from the "main sequence"

artifactName	fullQualifiedName	name	distance	abstractness	instability	elementsCount

Table 5c

- Show the top 30 Typescript modules with the highest distance from the "main sequence"

	artifactName	fullQualifiedName	name	distance	abstractness	instability	elementsCount
0	react-router-dom-v5-compat	/home/runner/work/code-graph-analysis-pipeline...	./index.ts	1.000000	0.000000	0.000000	0
1	router	/home/runner/work/code-graph-analysis-pipeline...	./index.ts	1.000000	0.000000	0.000000	0
2	react-router-native	/home/runner/work/code-graph-analysis-pipeline...	./index.tsx	0.384615	0.384615	1.000000	13
3	react-router-dom	/home/runner/work/code-graph-analysis-pipeline...	./index.tsx	0.339336	0.352941	0.986395	34
4	react-router-dom	/home/runner/work/code-graph-analysis-pipeline...	./server.tsx	0.333333	0.333333	1.000000	6
5	react-router	/home/runner/work/code-graph-analysis-pipeline...	./index.ts	0.024242	0.600000	0.424242	5

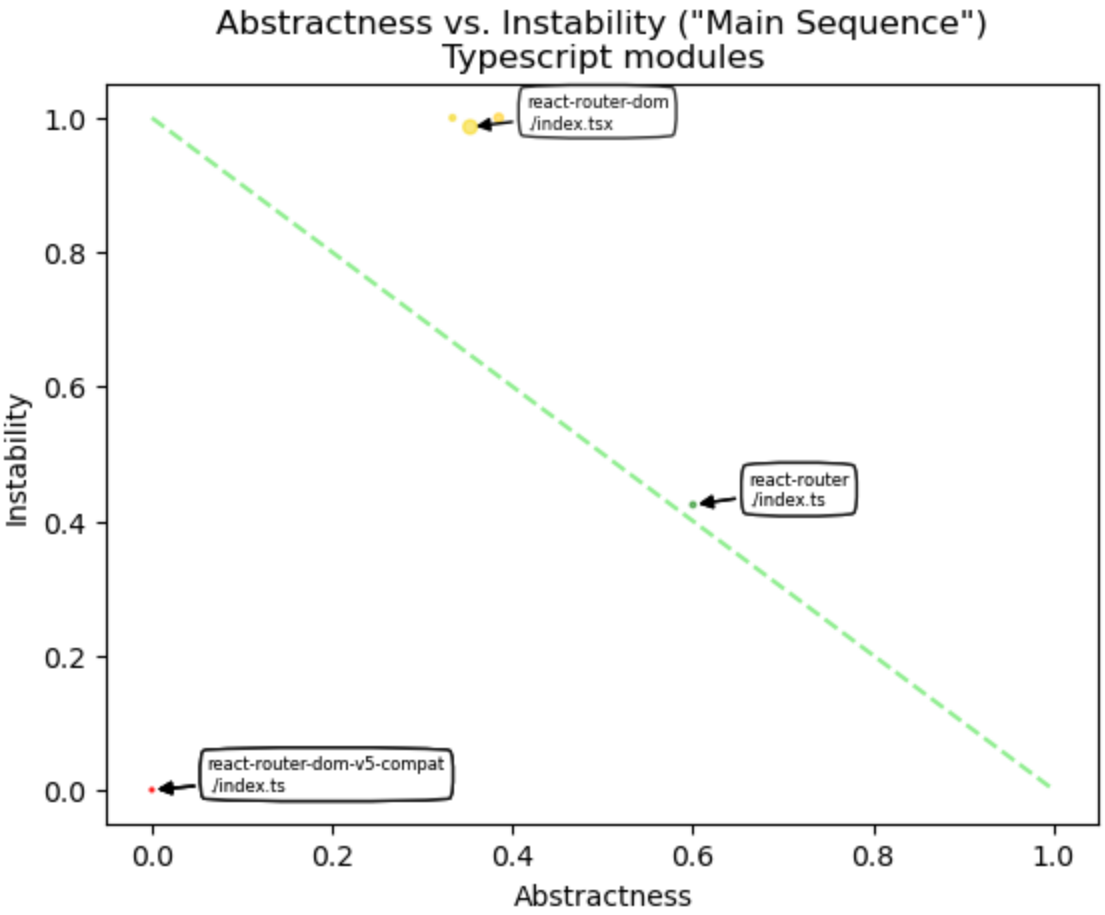
Abstractness vs. *Instability* Plot with "Main Sequence" line as reference

- Plot *Abstractness* vs. *Instability* of all packages
- Draw the "main sequence" as dashed green diagonal line
- Scale the packages by the number of types they contain
- Color the packages by their distance to the "main sequence" (blue=near, red=far)

Figure 1a - Packages without their sub-packages

Figure 1b - Packages including their sub-packages

Figure 1c - Typescript Modules


Abstractness vs. Instability ("Main Sequence")
Typescript modules
react-router-dom
/index.tsx
react-router
/index.ts
react-router-dom-v5-compat
/index.ts
Instability
Abstractness
0.0
0.2
0.4
0.6
0.8
1.0