Option 1: empirical network analysis – Analysing Software Quality with a Graph

Task: find data, analyze data (and visualize it), then interpret.

Author: Peter Huber,

Coursera-Profile https://www.coursera.org/user/i/bcbe876fe37a9cd5e3df7b10d971364b

Linkedin: http://de.linkedin.com/in/huberpeter/

DON'T PANIC – This file is only this big because I have included my lengthy input file in the Appendix!

Abstract: One importand measure for Softwarequality is often found in the area of Coupling of the different Building-Blocks of the Software, such Building Blocks are in Java so called Packages which contain Classes. So Java software is actually a network of so called Classes from different packages working together in some way. So you can say some Java classes that use different other Java classes depend on these Classes. Such usage/dependence relations can span multiple Java Packages.

If I refer to Classes or Packages in the following text this actually means the Java constructs.

If you look on such networks then you can find certain defects, one kind of defect is Dependency Cycles, that is when the Dependency Graph goes like: Class A uses B uses … finally come back to Class A and the Cycle spans multiple packages.

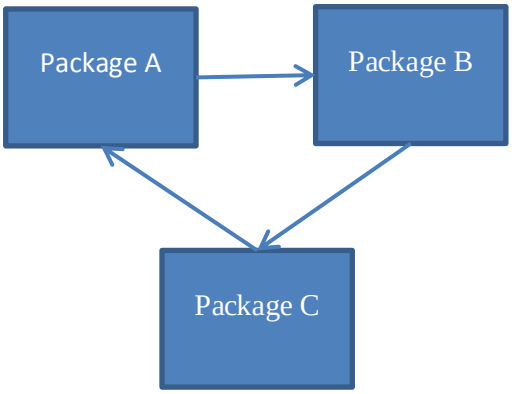


Figure 1: Trivial Cycle of 3 Packages

Why is this bad?

1.) It's a rather tight coupling of all the packages on the cycle, which is an indicator for bad software architecture – because SW-Engineers actually use packages to group Classes of a certain abstraction level. The "uses" / "depends" relation is most often also a dependency of "higher level of abstraction uses lower level of abstraction". In a cycle you must come to the point where this is turned upside down.

2.) If you start to refactor (i.e. change) Class A it might lead to a change in Class C which might lead to a change in B […] and finally it may require a 2nd change in Class A which leads to another change in class C…I guess you got it. BTW - The change-cycle is reverse order, because if you look at Figure 1: Trivial Cycle of 3 Packages then C depends on A)
3.) Cycles often hide: In medium size to large software projects you may end up with a multitude of cycles which are not easily found because they are more complex in structure than the one found depicted in Figure 1: Trivial Cycle of 3 Packages

And though there are a lot of software tools that are actually able to find such cycles and visualize them, it's mostly not done with capable "network" tools like gephi is, such that you can try to find more network properties like Clustering, Betweenness, etc.

And so this is my Project: Visualize and investigate the Dependency Cycles in the popular open source software "JUnit"

Obtaining data

1. Get the Code: I obtained the data, which is actually the source code of JUnit from https://github.com/junit-team/junit

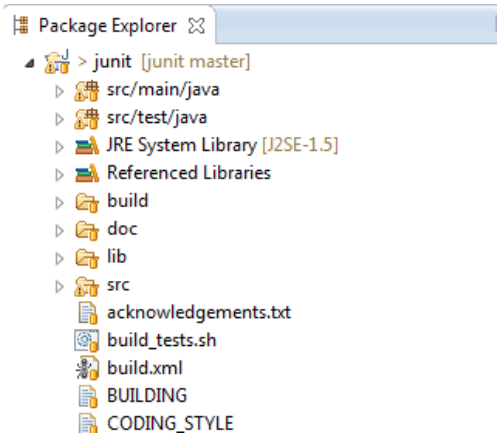


Figure 2 Junit Source Code opened in eclipse IDE

2. Then I used a rather old, but nevertheless very good Open source Tool to find the cycles in the Source Code with the name JDepend - http://www.clarkware.com/software/JDepend.html
With this tool I performed a static source code analysis on the JUnit Source Code and ended up with a XML file that contains information about all java packages, inter package dependencies and cycles.

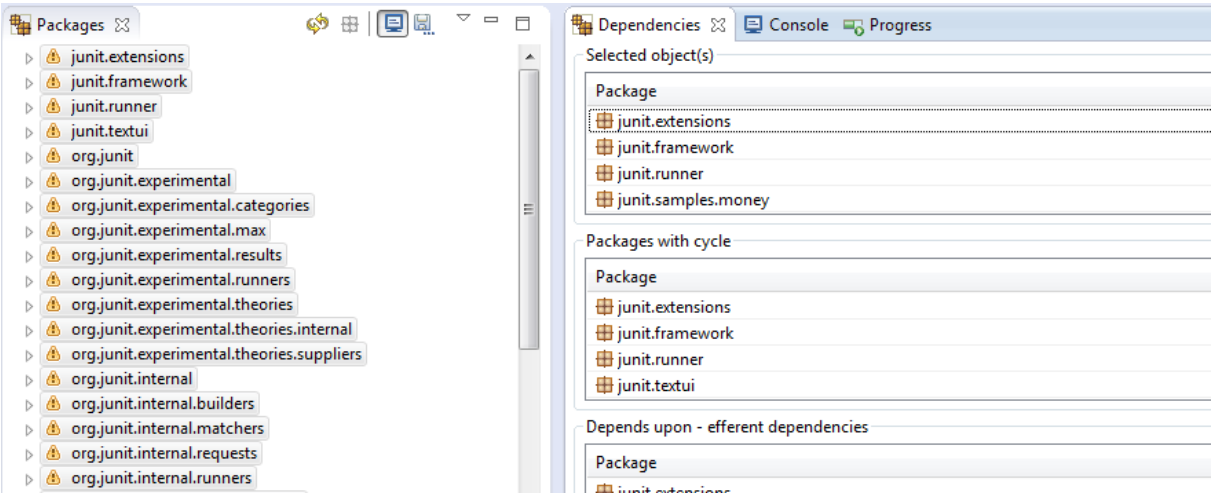


Figure 3 JDepend Analysis result: No Visualization

3. Then I run a Python script in Gephi (s. "Python-File for Gephi Python Scripting") to build nodes and edges. I have included a copy of the script in Chapter "Python-File for Gephi Python Scripting": What the Script does:
    a. For each Java Package found in the JDepend XML it creates a node. Nodes are sized by their degree.
    b. For each DependsUpon definition in JDepend XML it creates a directed edge from the current Package to the Package it uses.
    c. For each Cycle
        i. Mark Nodes within a cycle with a Cycle-Tag – find it in Gephis Data laboratory and Attribute "***javaCycles***". Each Package-Cycle is represented by its number as ordered in the JDepend XML-File and a Cycle Tag looks like "<1>" for the 1st cycle. If you locate node "org.junit" you find it having this value in ***javaCycles***. *"<1><3><5><7><9>[…]<107>*".
        Also Attribute ***javaCyclesNumber*** is used to record the number of Cycles and for easy access. Also edge weight is adjusted: For each cycle an edge participates in the edge weight is increased by 0.1.

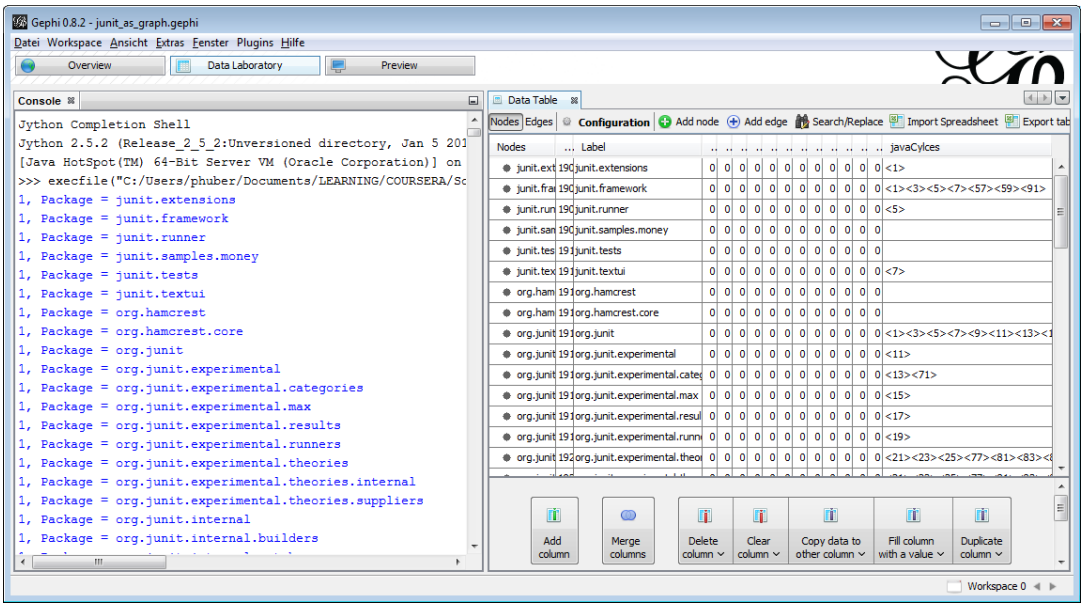
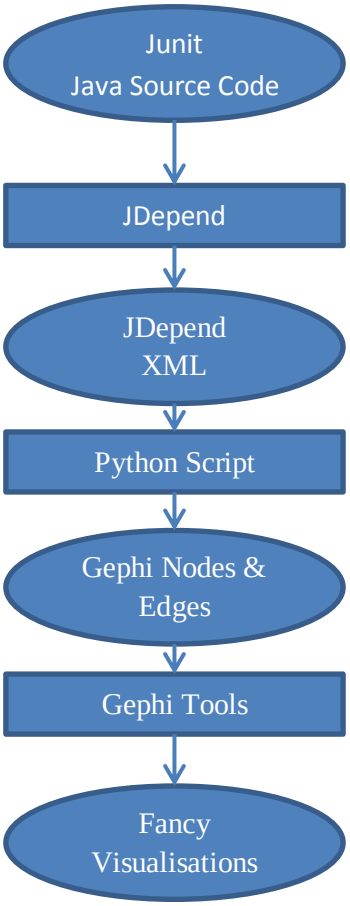
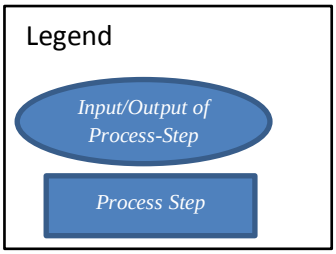


Figure 4 Data laboratory with resulting Nodes

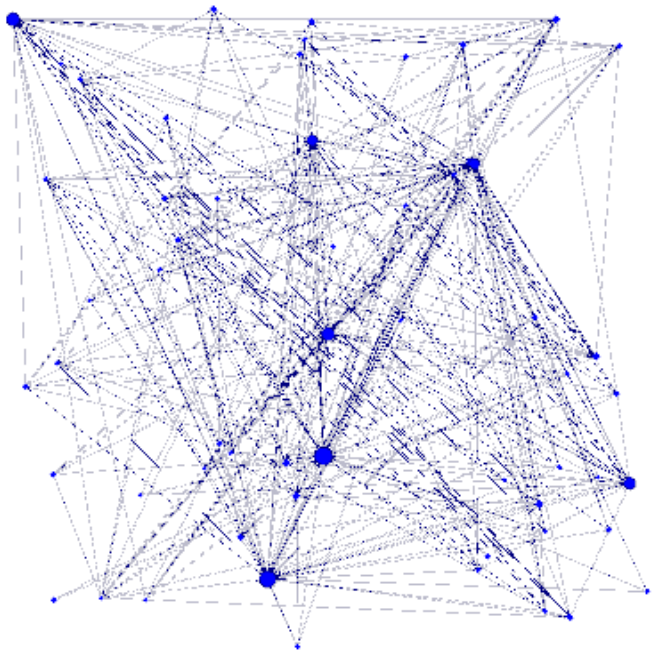
So the process summarized as Process-Workflow with Inputs and Outputs





Data analysis

After the Preprocessing step the Graph looks like this – not to satisfying, but already we can see bigger nodes which hints on their "importance" (remember size was set due to indegree)



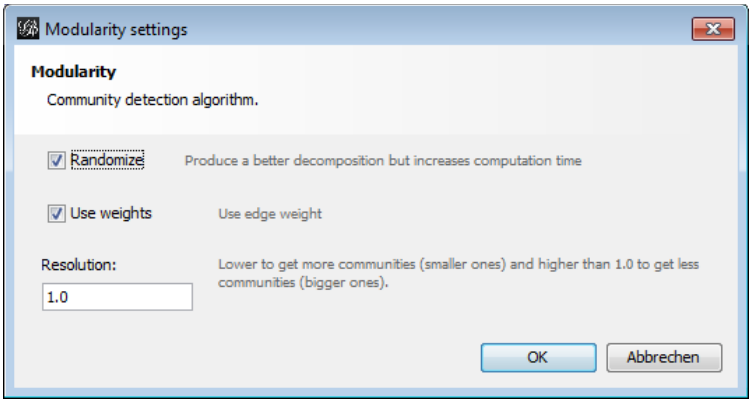
My goal was now to find important nodes or sort of not to obvious structures.

Modularity

Question: Is it possible to find a Graph-based modularity which in some sense reflects the modularity of the Software?

I computed edge weight with a +0.1 for each cycle an edge participates

1.) I went to compute Modularity with the following settings – Remember I set edge weights according to the number of cycles they participate in



2.) Also applying Force-Atlas 2 layout and…
3.) Partitioned the graph by "Modularity Class" which results in

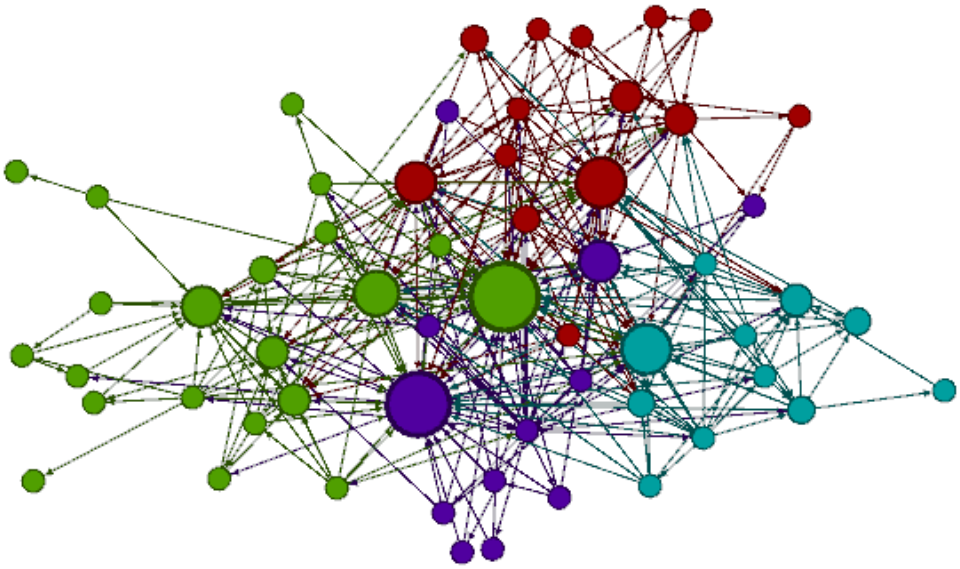


Figure 5 Partitions by Modularity Class

You can see there are 4 Modularity Classes – and mostly you have one big package representing this Module. Only exception is the “red” Module which has two almost same size “biggest” nodes.

Let’s look what the big nodes are. I show labels and highlight the big node with its dependencies

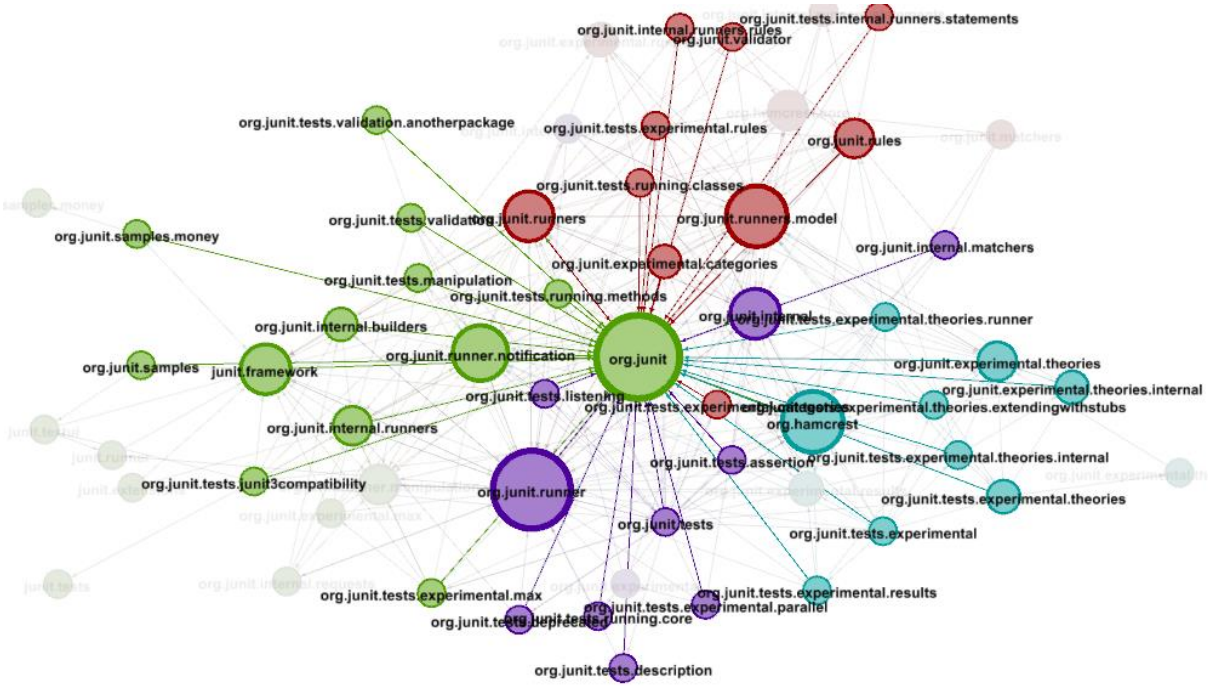


Figure 6 Partitions by Modularity Class – Highlight “org.junit”

It’s not surprising that the main Package “org.junit” is the biggest node as it is actually the central package of JUnit. And you can see that all other “Modules” and even the big nodes of the modules are connected

Let’s compare this to a highlight view of “org.junit.runner”

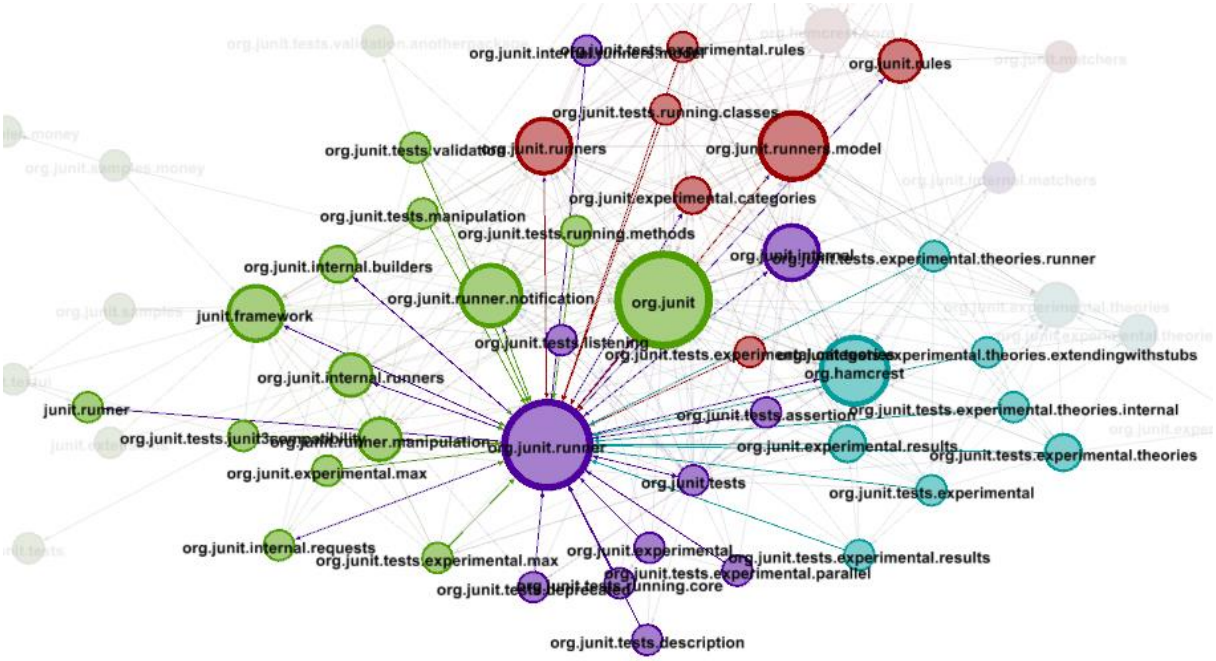


Figure 7 Partitions by Modularity Class – Highlight "org.junit.runner"

It's almost the same number of nodes and almost the same nodes either. Also not to surprising, because the JUnit is a Test-Framework where org.Junit let's say is responsible for "test definition" and "org.juinit.runner" is responsible for running those tests.

The red and cyan big-nodes have far fewer connections

Cyan "org.hamcrest"

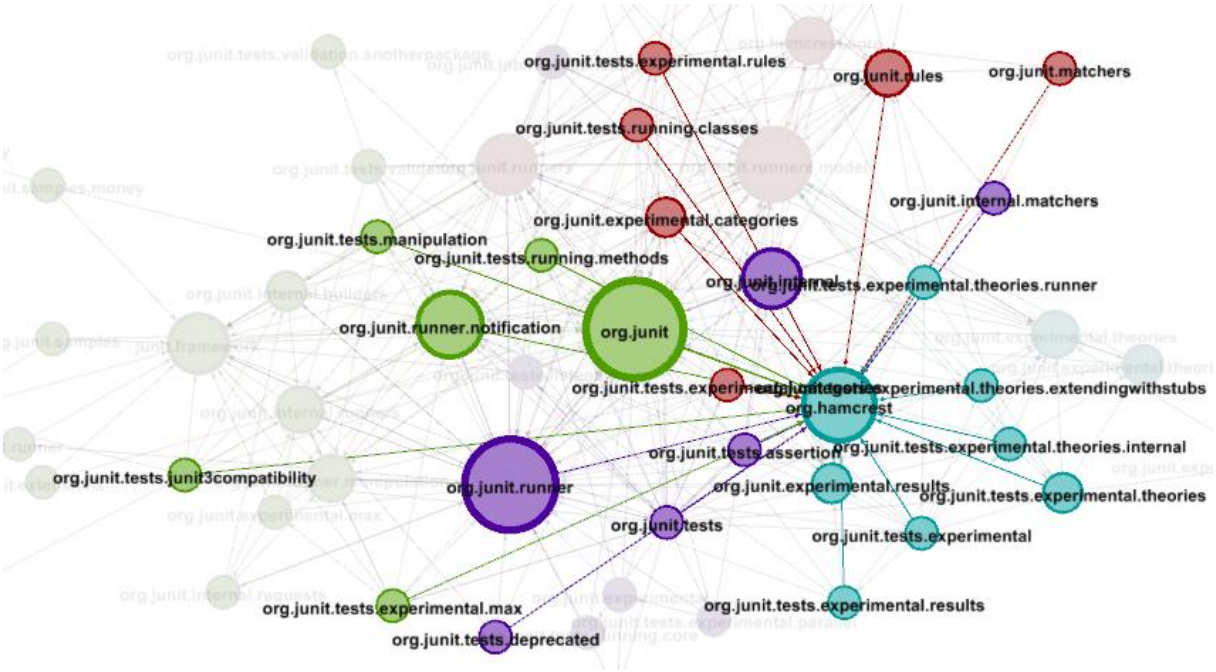


Figure 8 Partition by Modularity Class – Highlight "org.hamcrest"

Red "org.junit.runner"

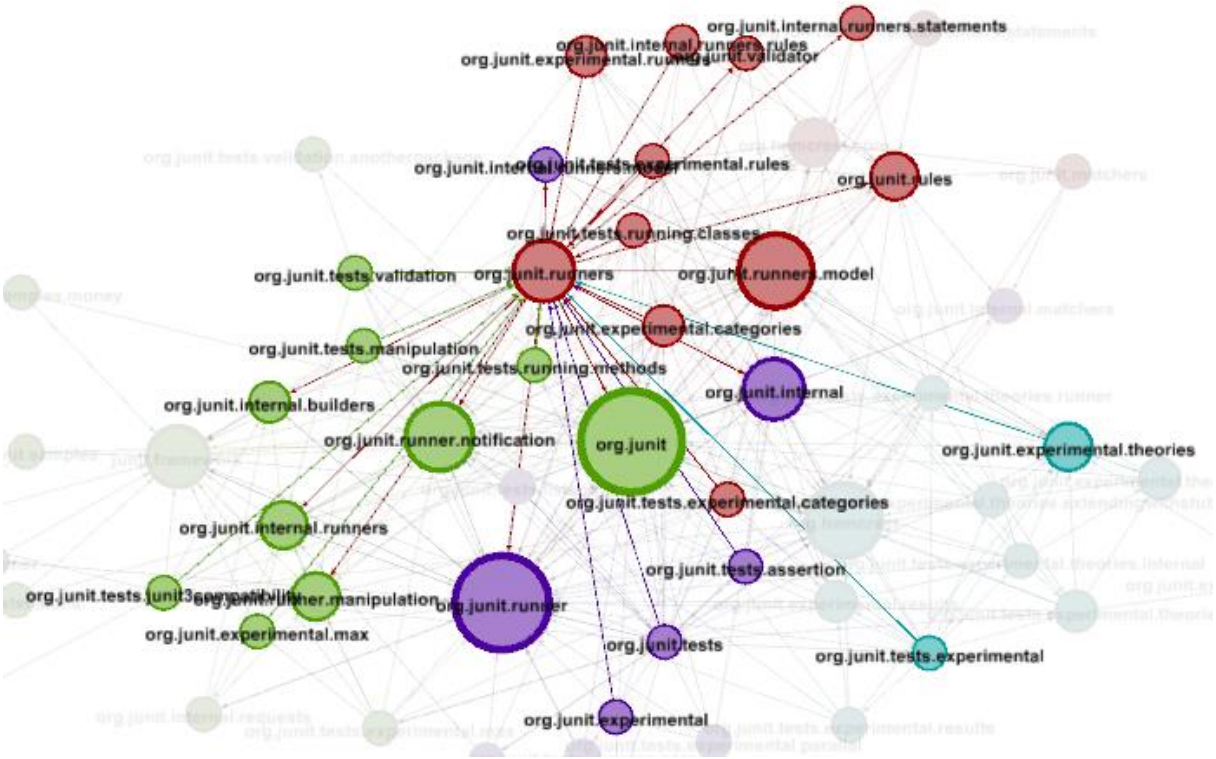


Figure 9 Partition by Modularity Class – Highlight "org.junit.runners"

Findings:

1.) A bit striking is that "org.junit.runner" (purple) and "org.junit.runners" (red) form different Graph-Modules. One might expect that those Java-packages stick close together because they seem to share similar purpose, i.e. running JUnit-tests.
2.) Overall the structure is awesome complicated and it seems that almost every package is somehow connected to each other package.

Modularity with different edge weights

I computed edge weight with a **+0.25** for each cycle an edge participates

Interesting to see that we now have 5 Modularity Classes

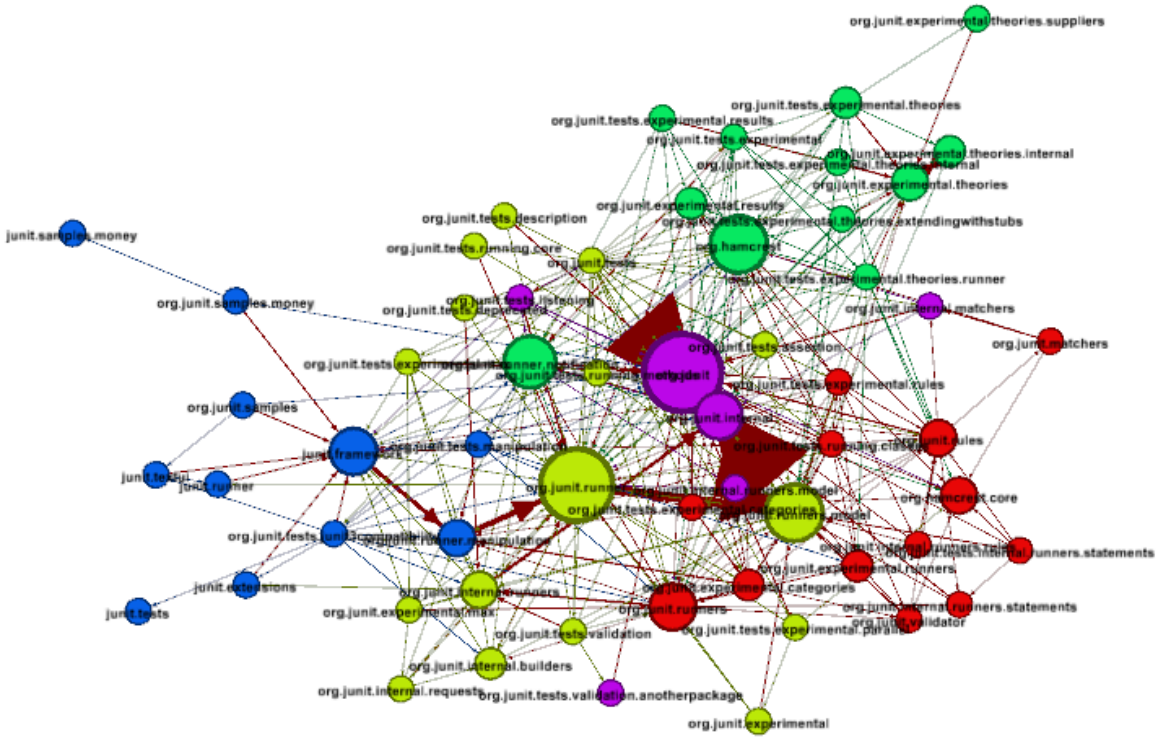


Figure 10 Partitions by Modularity Class with MORE Weights

We see that “org.junit” with only some satellites forms now a Modularity Class of its own.

The big “red” triangles come from the high edge weights, pointing us to those packages which actually participate in a lot of package cycles.

Looking at betweenness

1.) Next I computed Betweeness (Run “Network Diameter” in Gephi).
2.) Afterwards I’ve applied ranking based on “Betweenness Centrality”

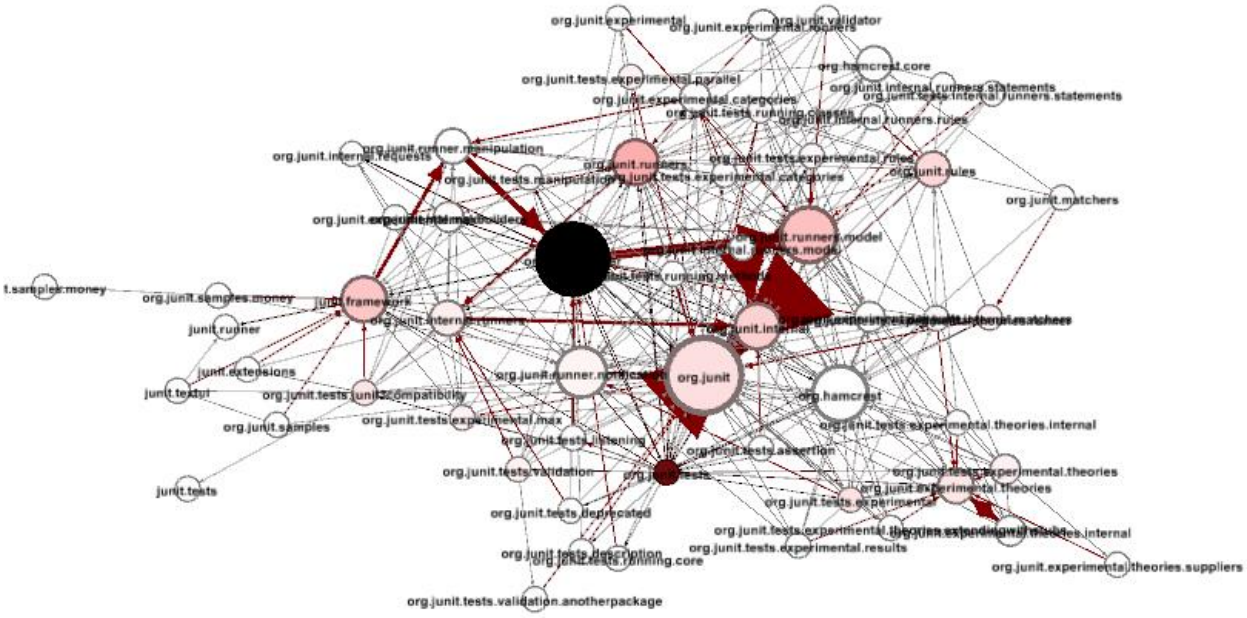


Figure 11 Betweenness Ranking

Finding:

1.) Though one might have expected that the highest betweenenss should be found in the central JUnit Package “org.junit” it is actually found in the package “org.junit.runner”
2.) I as a software architect would say this is sort of a problem that needs a closer look.
3.) You can also see that “betweenness” value somehow corresponds to the package cycles which are shown in red/bigger edges. Nodes with are special in their betweenness are connected to the biggest red edges which shows that they participate in many package cycles (see thick edge going in and coming out from “junit.org.runner”, the black node)

Connected Components

Next was to find out whether there are strongly connected components

Finding: No, there was just one! From what I have seen before this is again not surprising, as there are many package cycles and a big number of package dependencies overall. This means that actually all of joint is what software architects may call a “big ball of mud”.

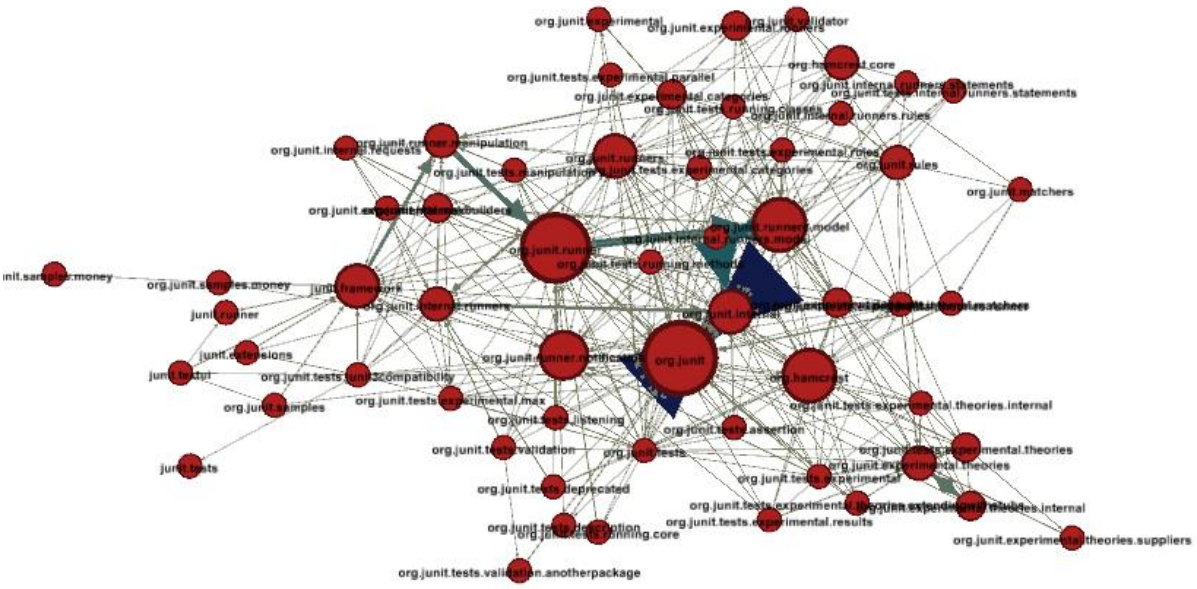


Figure 12 strongly connected Components

How close is the JUnit Graph to a random graph?

This chapter compares the graph metric values as computed by gephi .

Random Graph has the following Values

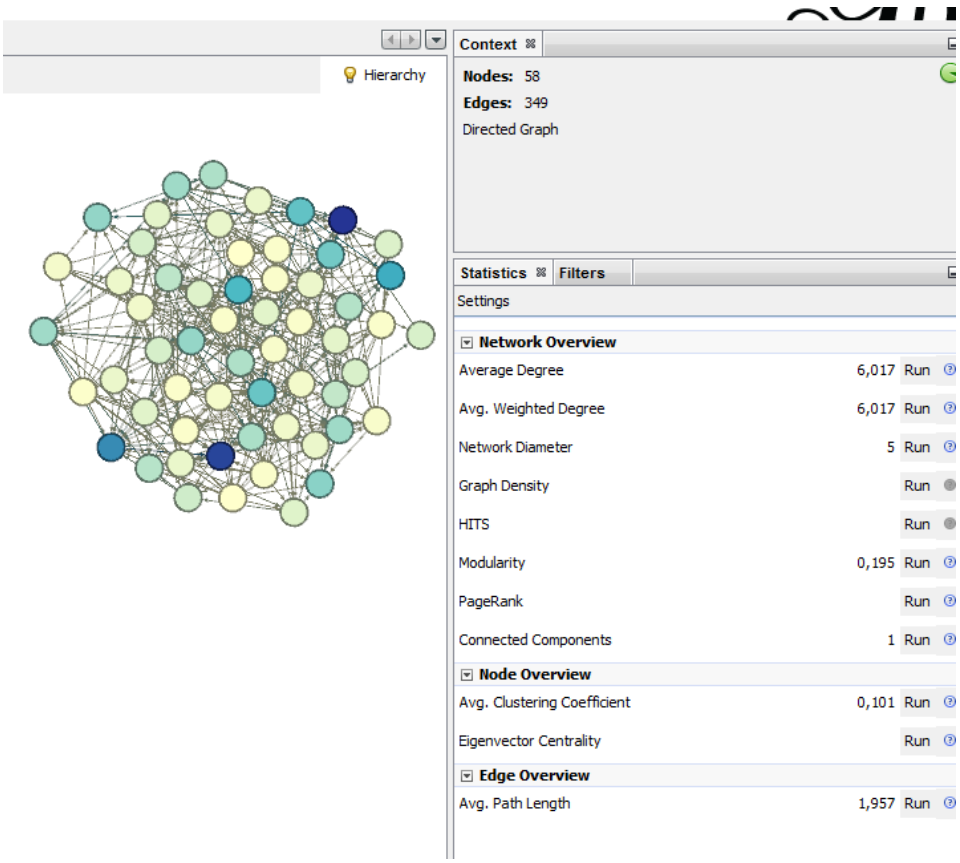


Figure 13 Metric Values of random graph

Now let's compare to the values of JUnit Package-Graph

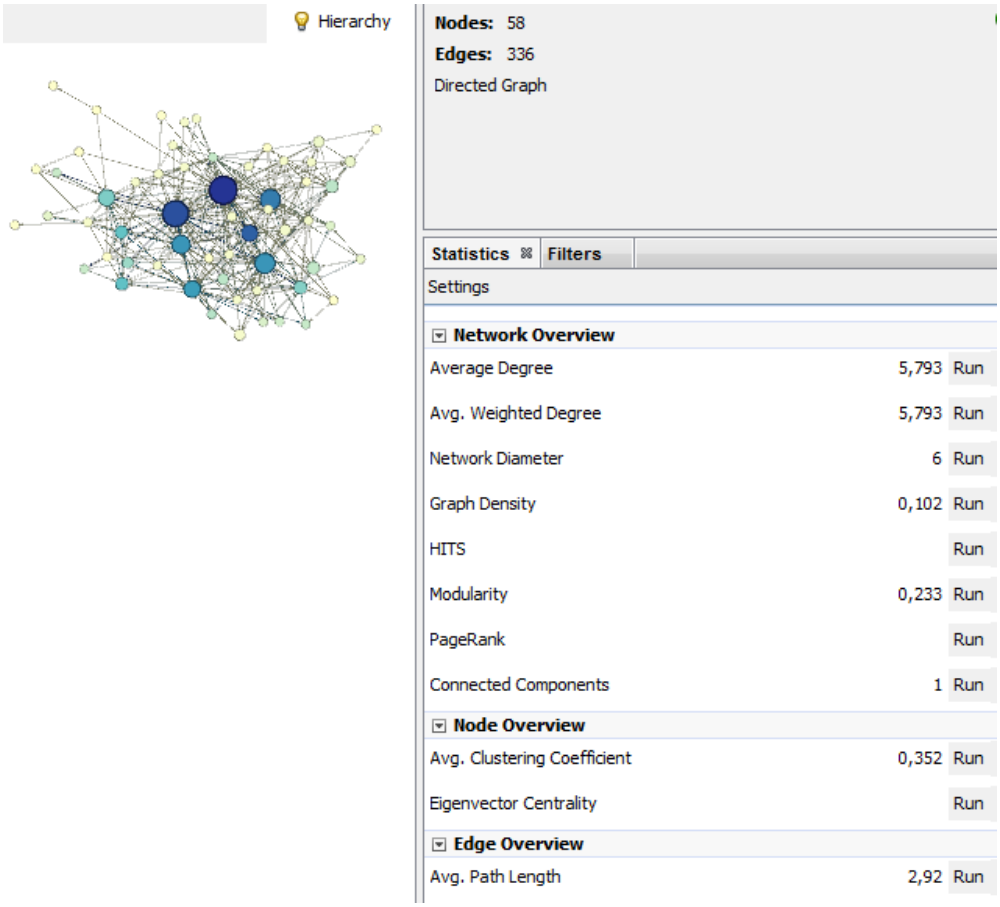


Figure 14 Metric Values of JUnit-Graph

Finding: We see slightly differing values. But if you say "Software architecture is a controlled process to plan, execute and enforce a sound software structure" I would have expected the values to be more different – for instance I would have expected a higher modularity (random = 0,195, joint = 0,233).

Interpretation

Most of the Interpretations you can find in the "Findings" sections of chapter "Data analysis".

These interpretations are only based on looking at just the Open source Project "JUnit". To be able to truly assess software by this graph metrics based approach, it would be necessary to build a data base of different software projects and have them assessed this way. Then you might

a.) Really find relevant metrics
b.) Be able to fine tune the parameters, like for instance my +0.1 for edged weight for each cycle it participates. This was selected purely by random ;-)
c.) Be able to find ranges for relevant metrics to classify software as bad, average or good in terms of architectural structure.

Appendix

Python-File for Gephi Python Scripting

This file turns a JDepend-File into Gephi Nodes and Edges.

For each **Packages/Package**-Element *A* in the JDepend-File it creates a Gephi Node.

For each **DependsUpon**-Element in the JDepend-File *B* it creates a directed Edge, which represents "The Java Package under inspection *A* depends upon the package *B*"

Here's a short excerpt of the JDepend-File

```
<?xml version="1.0"?>
<JDepend>
    <Packages>

        <Package name="junit.extensions">


            <DependsUpon>
                <Package>junit.framework</Package>
            </DependsUpon>
```

This example will result in a new node created "junit.extensions" and an edge from it to the other node which represents Java-package "junit.framework". These "junit."-Names are names of the Java-Packages used in JUnit.

```
#############################################################################################
# Python file to be used in gephi. It reads in JDepend XML Files and creates Nodes and Edges from it
# (c) 2013 - Peter Huber, MUC
#
# You can use this script as is or modify it and redistribute. Redistributing original or
# changed version is only allowed with giveing reference to the original author
#############################################################################################
# START WITH: execfile("[yourpath]/jdepend-xml-to-nodes_and_edges.py")
#
# Read about Python, german only: http://openbook.galileocomputing.de/python/python_kapitel_08_003.htm#mjfb4d02fccab9edcdc5ad084f35eaeaa6
#
import xml.dom.minidom as dom

 #
 # PASS 4: Handle Cycles
 # let's see how we can get the cycles information into it
 #
def checkOrAddCycle(gephiNode, cycleNumber):
          packageTag = ("<%d>" % (cycleNumber))
          currentCycles = gephiNode.javaCylces
          if currentCycles.find(packageTag) == -1:
                    gephiNode.javaCylces=currentCycles+packageTag
                    gephiNode.javaCyclesNumber = gephiNode.javaCyclesNumber+1
                    print gephiNode.javaCylces

def fctPass4HandleCycles(cyclesXMLElem, gephiNodesByPackageLabel):
#first prepare all "cycles"
          for node in gephiNodesByPackageLabel.values():
                    node.javaCylces = ""
                    node.javaCyclesNumber = 0


          i = 0;
          for xmlElem in cyclesXMLElem.childNodes:
                    if xmlElem.nodeType==dom.Node.ELEMENT_NODE and xmlElem.nodeName=="Package":
                              currentGephiNodeName = xmlElem.getAttribute("Name")
                              currentPackageGephiNode = gephiNodesByPackageLabel[currentGephiNodeName]
                              checkOrAddCycle(currentPackageGephiNode,i)
                              cycleMembersXMLElems = xmlElem.getElementsByTagName("Package")
                              if(len(cycleMembersXMLElems) != 0):
                                        for memberXMLElem in cycleMembersXMLElems:
                                                  memberPackageName = memberXMLElem.childNodes.item(0).data
                                                  memberPackageGephiNode = gephiNodesByPackageLabel[memberPackageName]
                                                  print "Cycle %d, member: %s" % (i, memberPackageName)
                                                  checkOrAddCycle(memberPackageGephiNode,i)
                                                  #now treat the corresponding edge
                                                  edges = currentPackageGephiNode -> memberPackageGephiNode
                                                  #be aware that the special "->" edge selector returns a set!
                                                  #in our case with just one member
                                                  edge = edges.pop()
```

```
                                                edge.color=red
                                                edge.weight = edge.weight+0.05
                                                #don't forget we are in a lop here, the next edge starts
                                                #from out current end
                                                currentPackageGephiNode = memberPackageGephiNode
                        i = i+1
 #
 # PASS 3: Go over all Package-Gephi-Nodes and resizes them according to their
 # indegree
 #
def fctPass3ResizeNodesByIndegree(gephiNodesByPackageLabel):
          for node in gephiNodesByPackageLabel.values():
                    node.size = 5.0 + (node.indegree / 4)


 #
 # PASS 2: Go over all Package-Elements and read their "DependsUpon"
 # We need this for later wiring dependencies thru edges
 #
def fctPass2ReadDependsOnCreateEdges(packagesXMLElem, gephiNodesByPackageLabel):
          for xmlElem in packagesXMLElem.childNodes:
                    if xmlElem.nodeType==dom.Node.ELEMENT_NODE and xmlElem.nodeName=="Package":
                              thisGephiNodeName = xmlElem.getAttribute("name")
                              thisPackageGephiNode = gephiNodesByPackageLabel[thisGephiNodeName]
                              ###getElementsByTagName works here because there's ONLY one single DependsUpon
                              ###per Package elem, whereas there are Package Elems wrapped in Packaeg Elems...;-(
                              theSingleDependsUponXMLElem = xmlElem.getElementsByTagName("DependsUpon")
                              #could be that packagees don't have DependsUpon-Section
                              if(len(theSingleDependsUponXMLElem) != 0):
                                        theDepPackages= theSingleDependsUponXMLElem.item(0).getElementsByTagName("Package")
                                        for depPackageXMLElem in theDepPackages:
                                                  #oh, wow, this DOM-API is really some kind of dep nested ;-)
                                                  otherGephiNodeName = depPackageXMLElem.childNodes.item(0).data
                                                  otherPackageGephiNode = gephiNodesByPackageLabel[otherGephiNodeName]
                                                  print "Edge: %s -> %s" % (thisPackageGephiNode, otherPackageGephiNode)
                                                  nuEdge = g.addDirectedEdge(thisPackageGephiNode, otherPackageGephiNode)
                                                  nuEdge.label="Source-DependsUpon-Target"

 #
 # PASS 1: Go over all Package-Elements and create a gephi Node for them
 # We need this for later wiring dependencies thru edges
 #
def fctPass1ReadPackageCreateNodes(packagesXMLElem):
          gephiNodesByLabel = {}
          ##print packagesElem
          ##print packagesElem.childNodes
          for xmlElem in packagesXMLElem.childNodes:
                    if xmlElem.nodeType==dom.Node.ELEMENT_NODE and xmlElem.nodeName=="Package":
                              gephiNodeName = xmlElem.getAttribute("name")
                              print "%d, %s = %s" % (xmlElem.nodeType, xmlElem.nodeName, gephiNodeName)
                              nuGephiNode = g.addNode(label=gephiNodeName,color=blue, size=5.0)
                              gephiNodesByLabel[gephiNodeName] = nuGephiNode
          return gephiNodesByLabel


def main():

          #now parse the xml
          jdependDOM = dom.parse("[yourpath]/jdepend-on-gephi-visualization_plugin.xml")

          # NODES AND EDGES
          #
          #Create Nodes which represent a Java-Package each
          #get access to the package element in the xml
          packageElement = jdependDOM.childNodes.item(0).childNodes.item(1)
          gephiNodesByPackageLabel = fctPass1ReadPackageCreateNodes(packageElement)

          #now wire the packages, i.e. we have to go over the xml again :-(
          fctPass2ReadDependsOnCreateEdges(packageElement, gephiNodesByPackageLabel)

          #resize by indegree
          fctPass3ResizeNodesByIndegree(gephiNodesByPackageLabel)

          # CYCLES
          #
          cyclesElement  = jdependDOM.childNodes.item(0).childNodes.item(3)
          fctPass4HandleCycles(cyclesElement, gephiNodesByPackageLabel)

main()
```

JDepend XML-Output for JUnit Source Code

This is a shortened version of the JDepend XML-Output on JUnit. It's purpose is to show you the structure of the file. Please note, that there is even more to analyse as each Java Packages also has some metrics like Ce, Ca, A, I, D which are currently not used.

```
<?xml version="1.0"?>
<JDepend>
    <Packages>
```

```
    <Package name="junit.extensions">
        <Stats>
            <TotalClasses>7</TotalClasses>
            <ConcreteClasses>6</ConcreteClasses>
            <AbstractClasses>1</AbstractClasses>
            <Ca>2</Ca>
            <Ce>1</Ce>
            <A>0.14</A>
            <I>0.33</I>
            <D>0.52</D>
            <V>1</V>
        </Stats>

        <AbstractClasses>
            <Class sourceFile="package-info.java">
                junit.extensions.package-info
            </Class>
        </AbstractClasses>

        <ConcreteClasses>
            <Class sourceFile="ActiveTestSuite.java">
                junit.extensions.ActiveTestSuite
            </Class>
            <Class sourceFile="ActiveTestSuite.java">
                junit.extensions.ActiveTestSuite$1
            </Class>
            […]
            <Class sourceFile="TestSetup.java">
                junit.extensions.TestSetup$1
            </Class>
        </ConcreteClasses>

        <DependsUpon>
            <Package>junit.framework</Package>
        </DependsUpon>

        <UsedBy>
            <Package>org.junit.internal.runners</Package>
            <Package>org.junit.tests.junit3compatibility</Package>
        </UsedBy>
    </Package>

    <Package name="junit.framework">
        <Stats>
            <TotalClasses>18</TotalClasses>
            <ConcreteClasses>13</ConcreteClasses>
            <AbstractClasses>5</AbstractClasses>
            <Ca>17</Ca>
            <Ce>5</Ce>
            <A>0.28</A>
            <I>0.23</I>
            <D>0.49</D>
            <V>1</V>
        </Stats>

        <AbstractClasses>
            <Class sourceFile="Protectable.java">
                junit.framework.Protectable
            </Class>
            […]
        </AbstractClasses>

        <ConcreteClasses>
            […]

        </ConcreteClasses>

        <DependsUpon>
            <Package>org.junit</Package>
            <Package>org.junit.internal</Package>
            <Package>org.junit.runner</Package>
            <Package>org.junit.runner.manipulation</Package>
            <Package>org.junit.runner.notification</Package>
        </DependsUpon>

        <UsedBy>
            <Package>junit.extensions</Package>
            <Package>junit.runner</Package>
            <Package>junit.textui</Package>
            <Package>org.junit.experimental.max</Package>
            <Package>org.junit.internal.builders</Package>
            <Package>org.junit.internal.runners</Package>
            <Package>org.junit.runner</Package>
            <Package>org.junit.samples</Package>
            <Package>org.junit.samples.money</Package>
            <Package>org.junit.tests</Package>
            <Package>org.junit.tests.experimental.max</Package>
            <Package>org.junit.tests.experimental.rules</Package>
            <Package>org.junit.tests.junit3compatibility</Package>
            <Package>org.junit.tests.listening</Package>
            <Package>org.junit.tests.manipulation</Package>
            <Package>org.junit.tests.running.classes</Package>
            <Package>org.junit.tests.running.methods</Package>
        </UsedBy>
    </Package>

    […many more package sections follow…]
</Packages>
```

```
    <Cycles>
        <Package Name="junit.extensions">
            <Package>junit.framework</Package>
            <Package>org.junit.runner.manipulation</Package>
            <Package>org.junit.runner</Package>
            <Package>org.junit.runners.model</Package>
            <Package>org.junit.internal.runners.model</Package>
            <Package>org.junit.internal</Package>
            <Package>org.junit</Package>
            <Package>org.junit.internal</Package>
        </Package>

        <Package Name="junit.framework">
            <Package>org.junit.runner.manipulation</Package>
            <Package>org.junit.runner</Package>
            <Package>org.junit.runners.model</Package>
            <Package>org.junit.internal.runners.model</Package>
            <Package>org.junit.internal</Package>
            <Package>org.junit</Package>
            <Package>org.junit.internal</Package>
        </Package>

        […many more cycles follow…]
    </Cycles>
</JDepend>
```