Software Quality Introduction

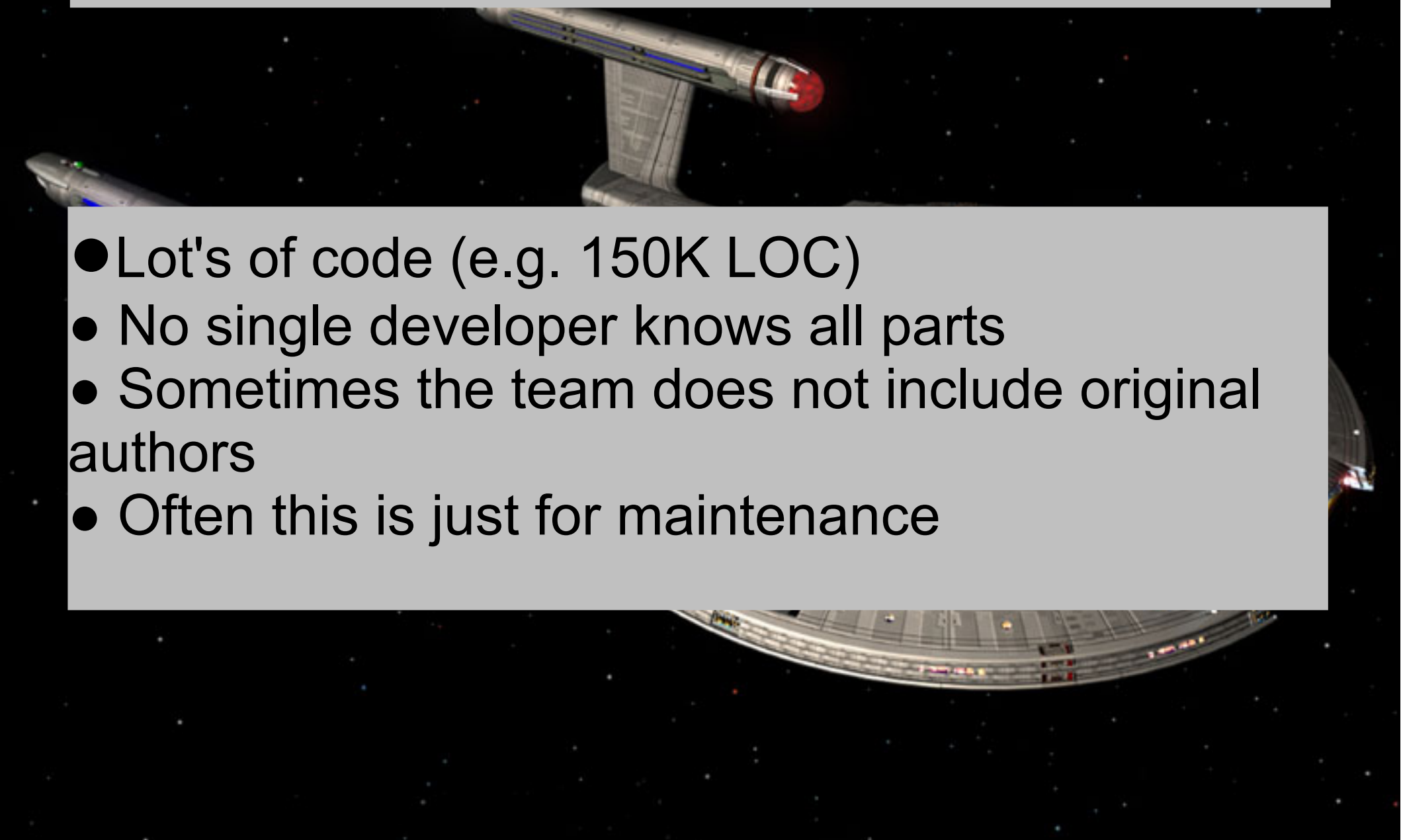
Kapelonis Kostis
(kkapelon@gmail.com)

JHUG February 2012

Quality in Greek IT ?

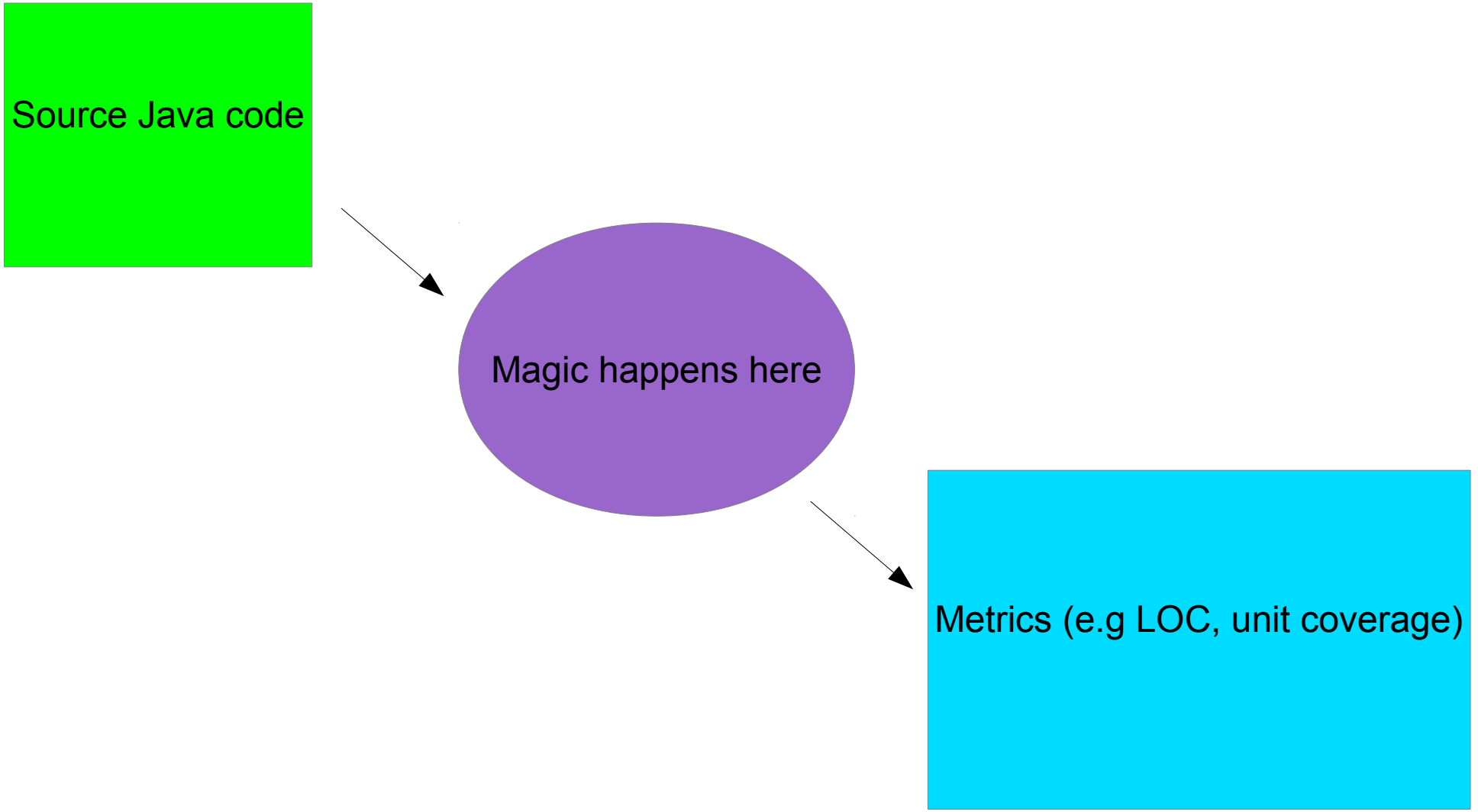
Enterprise Projects

Enterprise Projects



- Lot's of code (e.g. 150K LOC)
- No single developer knows all parts
- Sometimes the team does not include original authors
- Often this is just for maintenance

Software Quality Denifition (mine)



Common Quality Pitfalls

- Quality is an ongoing process (like security)
- Your code must be correct
- Metrics show trends and indications
- Historical data is a must

Software Quality 1/3

High level (OOP)
Package quality
General architecture
Component interconnections

Software Quality 2/3

- High level (OOP)
Class quality
Focus on a file
RFC, DIT, CC, LCOM, e.t.c.

Software Quality 3/3

- Low level (Java)

Code quality
Java issues
Findbugs, PMD, e.t.c

Quality ala JDepend

Jdepend (bread and butter for package quality)

Metrics used are covered in “OO Design Quality Metrics” by Robert Martin in 1994

http://www.objectmentor.com/resources/articles/oodmetrc.pdf

Metric Results

[summary] [packages] [cycles] [explanations]

The following document contains the results of a JDepend metric analysis. The various metrics are defined at the bottom of this document.

Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
org.displaytag	1	1	0	9	3	0.0%	25.0%	75.0%	1
org.displaytag.decorator	13	8	5	5	13	38.0%	72.0%	11.0%	1
org.displaytag.exception	14	12	2	7	6	14.0%	46.0%	40.0%	1
org.displaytag.export	16	11	5	1	17	31.0%	94.0%	26.0%	1
org.displaytag.filter	6	5	1	0	10	17.0%	100.0%	17.0%	1
org.displaytag.localization	6	4	2	2	18	33.0%	90.0%	23.0%	1
org.displaytag.model	9	9	0	5	13	0.0%	72.0%	28.0%	1
org.displaytag.pagination	5	4	1	2	8	20.0%	80.0%	0.0%	1
org.displaytag.properties	5	5	0	7	17	0.0%	71.0%	29.0%	1
org.displaytag.render	6	3	3	2	16	50.0%	89.0%	39.0%	1
org.displaytag.tags	12	10	2	2	24	17.0%	92.0%	9.0%	1
org.displaytag.tags.el	8	8	0	0	8	0.0%	100.0%	0.0%	1
org.displaytag.util	18	14	4	7	15	22.0%	68.0%	10.0%	1

How to run it

- mvn jdepend:generate
- Eclipse JDepend plugin
- Eclipse CAP plugin
- Metrics are OOP (language independent)
- Theoretically also applies to .NET or C++

Package Cycles

Cycles

[summary] [packages] [cycles] [explanations]

There are no cyclic dependencies.

Good (YES!)

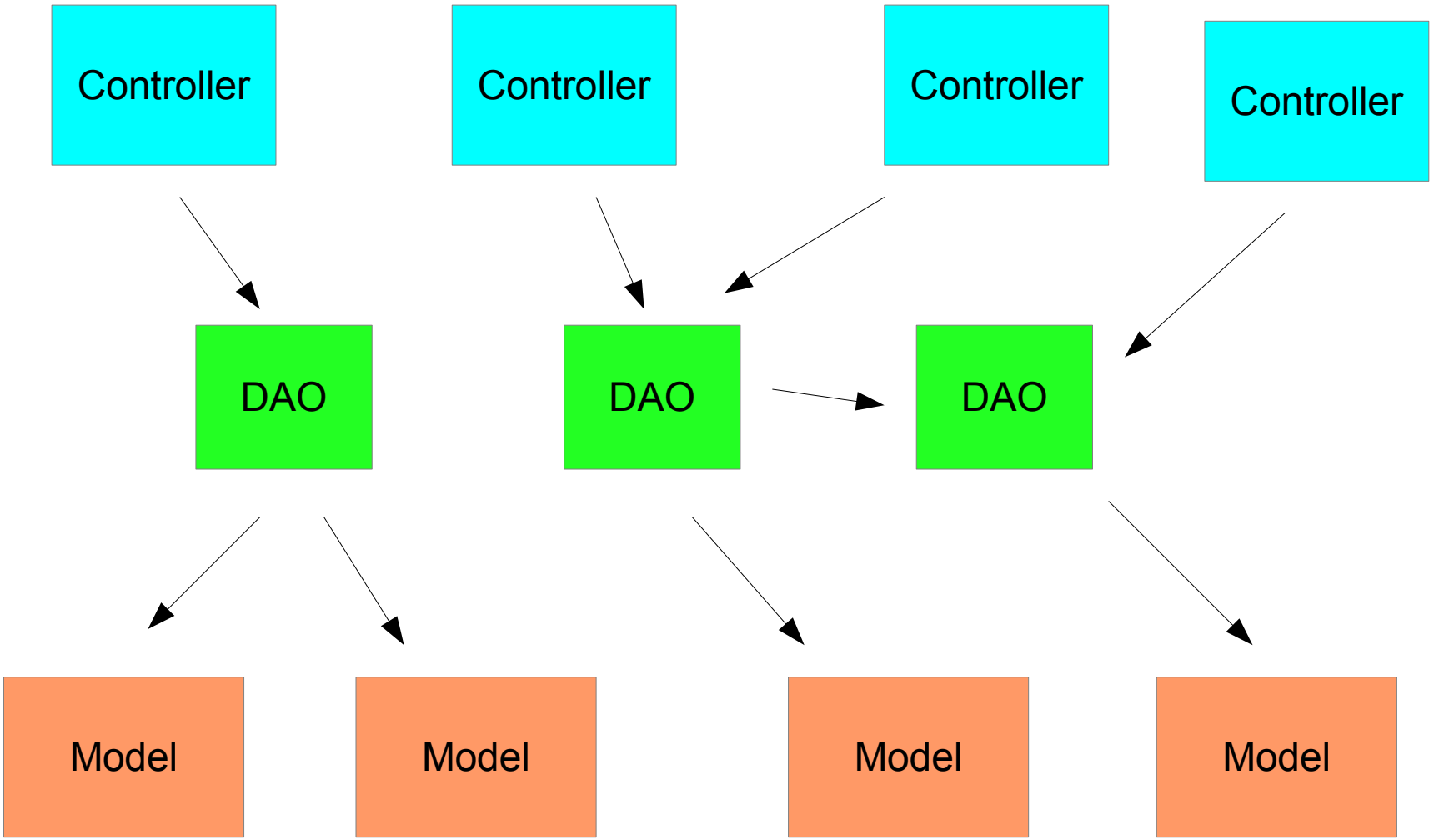
Package cycles

[summary] [packages] [cycles] [explanations]

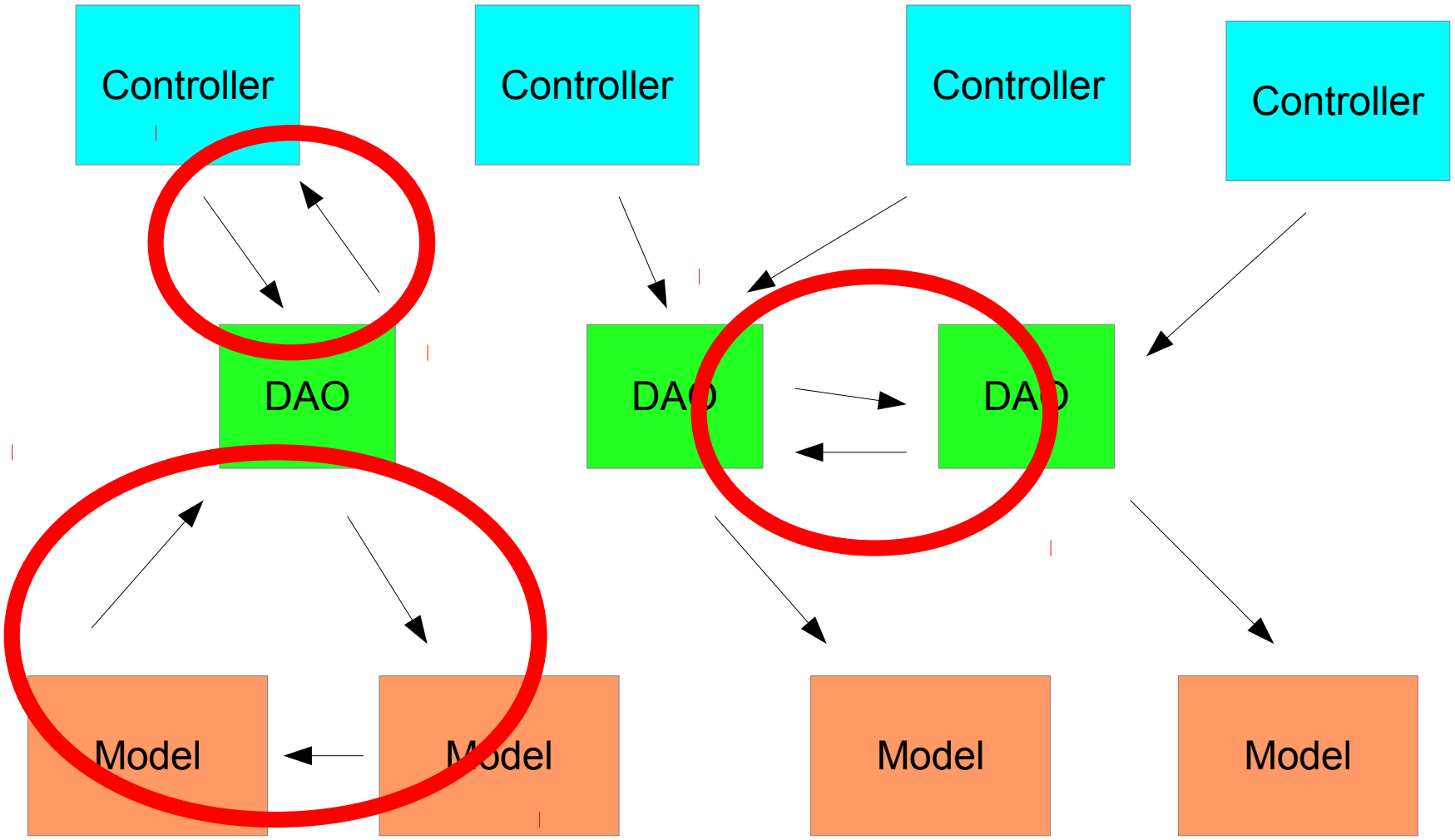
Package	Package Dependencies
org.apache.ws.commons.schema	org.apache.ws.commons.schema.utils org.apache.ws.commons.schema
org.apache.ws.commons.schema.extensions	org.apache.ws.commons.schema org.apache.ws.commons.schema.utils org.apache.ws.commons.schema
org.apache.ws.commons.schema.tools	org.apache.ws.commons.schema org.apache.ws.commons.schema.utils org.apache.ws.commons.schema
org.apache.ws.commons.schema.utils	org.apache.ws.commons.schema org.apache.ws.commons.schema.utils

Bad (Time to refactor!)

Good Architecture



Bad Architecture



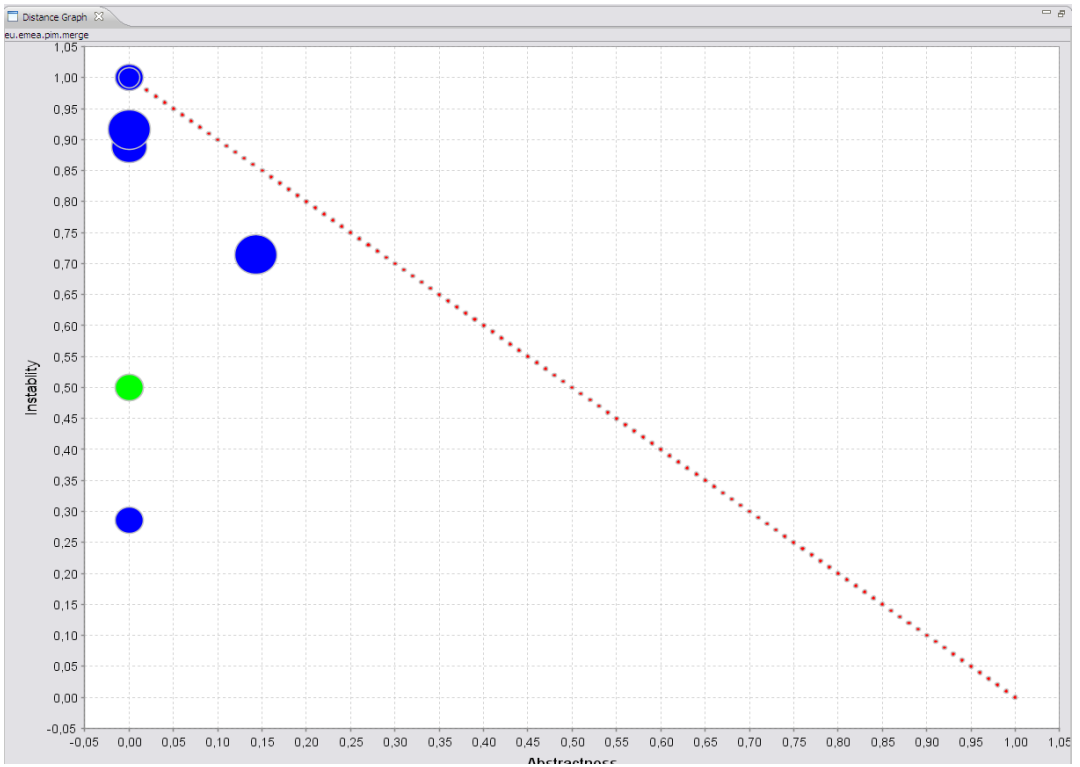
Why cycles are bad

- Logic errors (Dao depends on Controller)
- Packages with cycles are difficult to refactor
- You must change all packages at once
- Solve cycles by moving classes to new packages
- Solve cycles by merging packages

Architecture distance

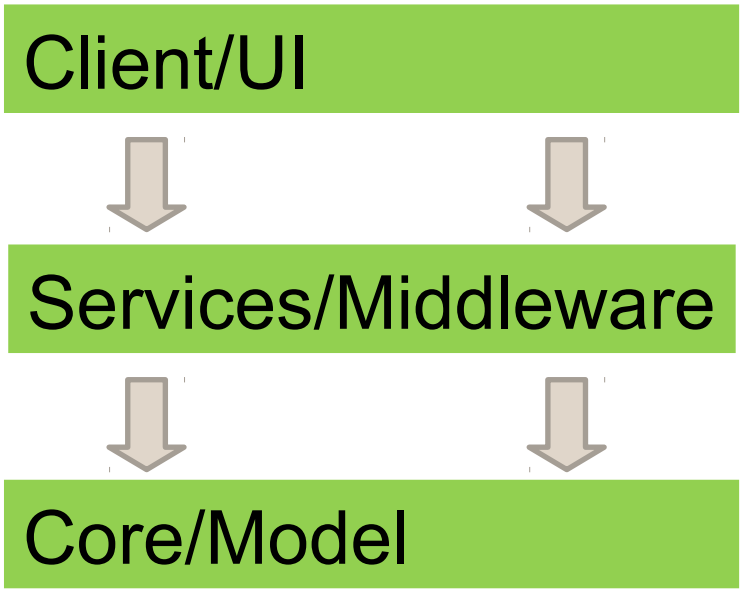
Distance is a number from 0% to 100%

- Distance is 0% means perfect system, 100% means ugly system
- We need to define what is the *perfect system according to JDepend*
- We also define instability and abstractness (also by JDepend)



Typical Enterprise system

- There are classes *used* by everybody
- There are classes that *use* everybody else
- Each layer depends on the one below (ideally)
- Core classes do not depend on anything
- Clients are not used by anything

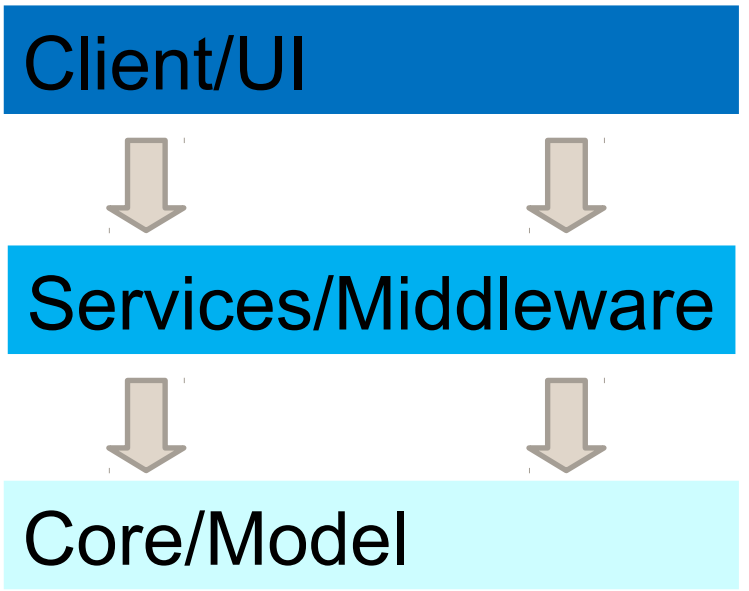


Perfect Enterprise System

- JDepend suggests that:
- Classes that *use* everybody else should be concrete
- Classes *used* by everybody should be abstract
- Distance is how far you are from this perfect system

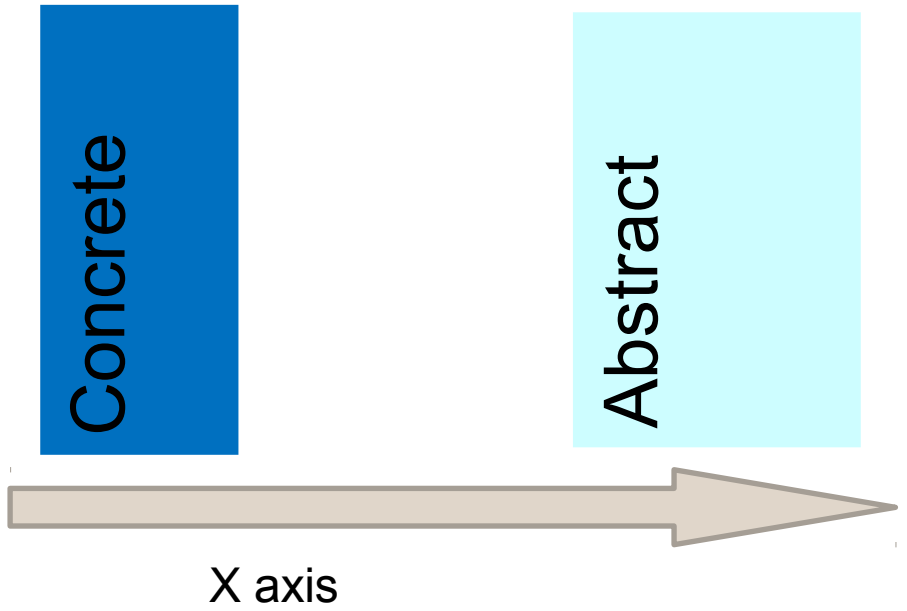
Concrete Classes

Abstract/Interfaces



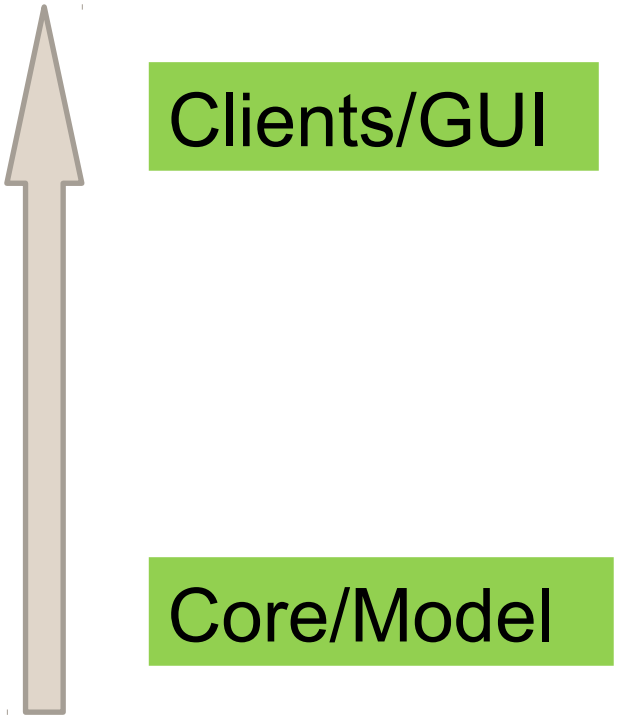
Abstractness

- Percent of classes in a package that are abstract/interfaces
- 0 = a package with concrete classes only
- 1 = a package with only abstract classes
- The x – direction shows “abstractness” of a package



(In)Stability

- (Inverse) Ratio of packages that are depended upon this package
- 0 = a package that everybody uses
- 1 = a package nobody uses
- The y – direction shows “instability” of a package



Distance example (1/5)

- Perfect package for gui/clients
- Used by nobody (instability = 1)
- All classes are concrete (abstractness = 0)

Good!

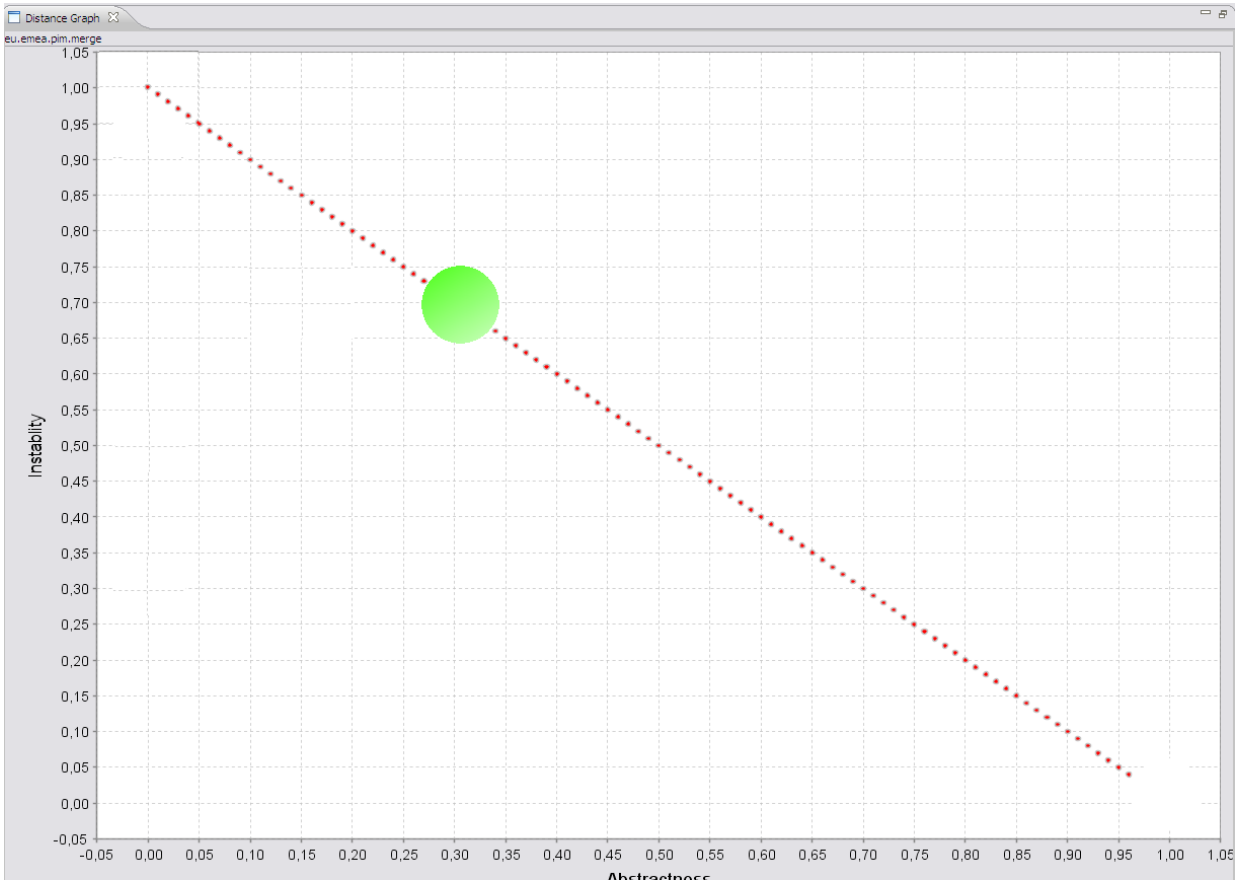
Distance Example (2/5)

- Perfect package for core/model
- Used by everybody(instability = 0)
- No concrete class (abstractness = 1)

Good!

Distance Example (3/5)

- Middleware
- Used by some and uses others (instability = 0 -1)
- Both concrete class and abstract classes(abstractness = 0 -1)



Good!

Distance example (4/5)

- Badly designed core (most common case!)
- Used by everybody (instability = 0)
- Concrete implementation – hard to change (abstractness = 0)

Bad

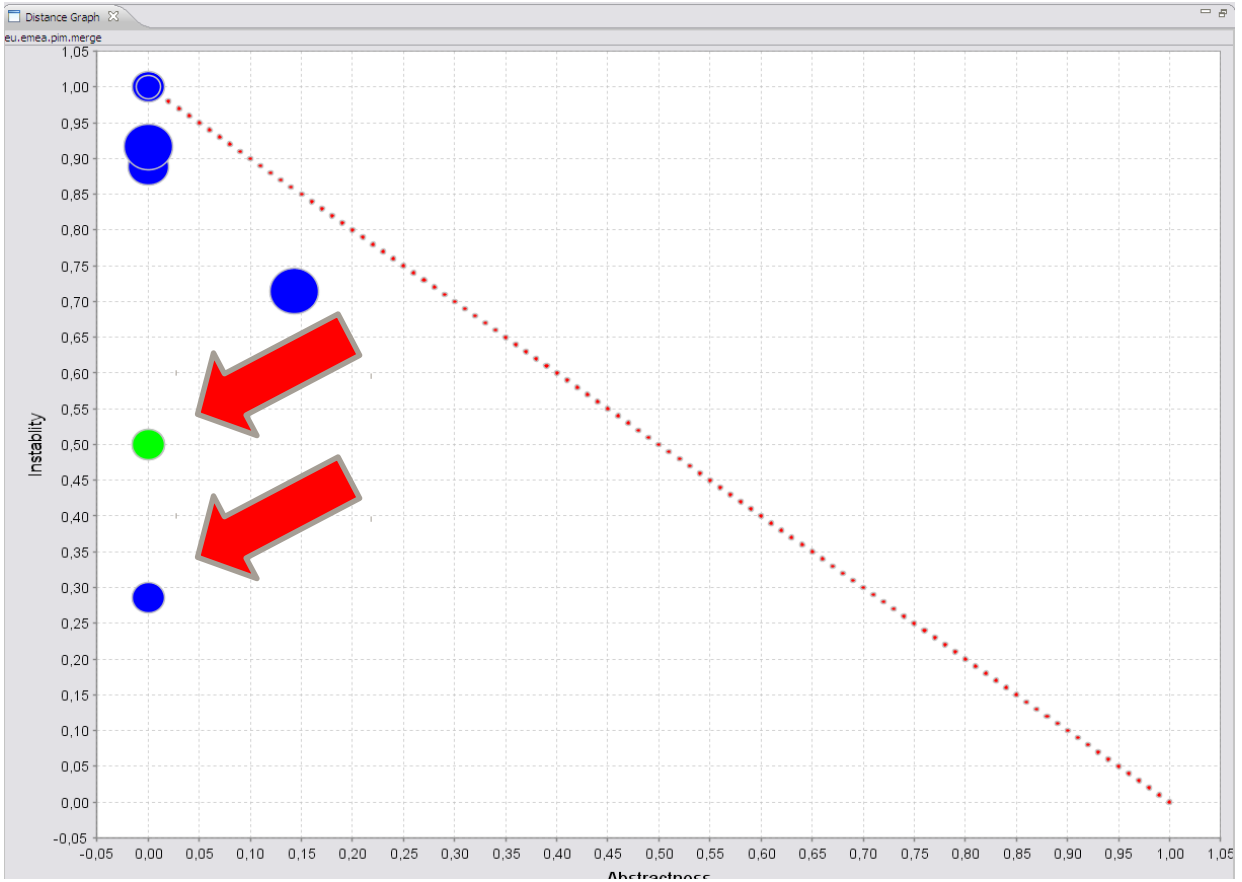
Distance Example (5/5)

- Un-needed abstract classes (uncommon case)
- Used by nobody(instability = 1)
- Abstract implementation (abstractness = 1)

Bad

Full Example

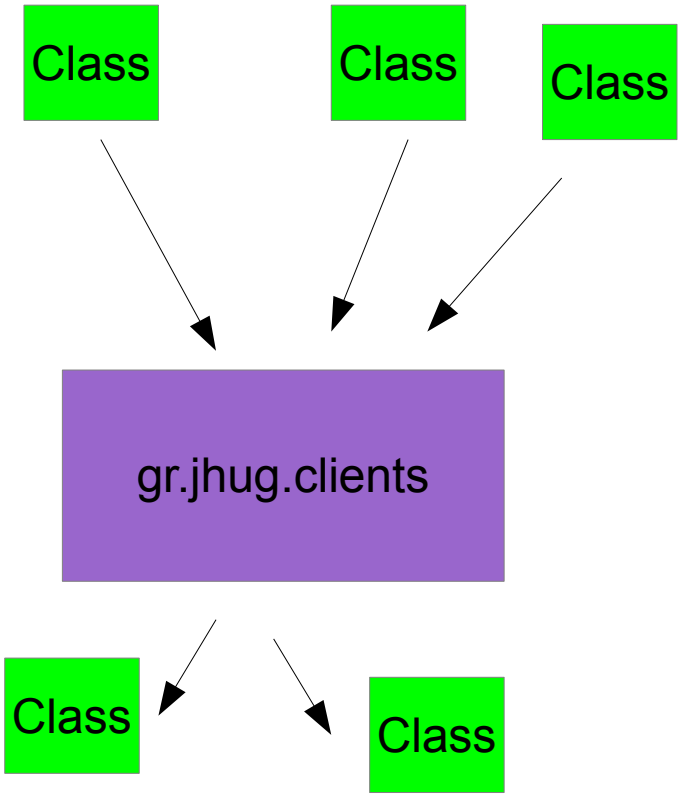
- With one look you can see the general architecture
- You can select a package on the graph to see details
- Real life example (two packages should be more abstract)



Questions/Answers

JDepend metrics (1/3)

- Afferent Couplings (AC) Classes that use this package
- Efferent Couplings (EC) Classes used by this package



AC = 3, EC =2

Jdepend Metrics (2/3)

- Abstractness = Percent of abstract classes interfaces
- 1 Interface, 3 classes = 0.25
- 1 Interface, 1 Abstract, 4 concrete classes = 0.3

Jdepend metrics (3/3)

- Instability CE/ (CE +CA)
- I = 0 means stable package (CE = 0)
- I = 1 means unstable package (CA = 0)
- CA = 4, CE = 2 means I = 0.3
- D = difference for A + I = 1