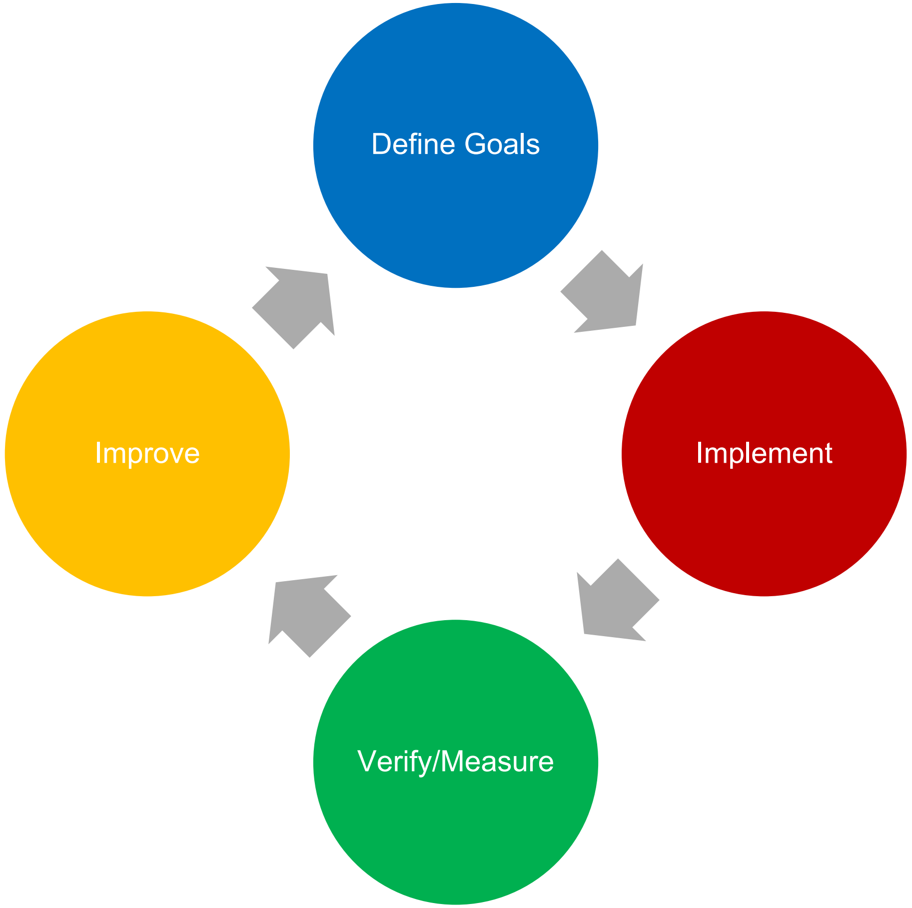
Software Metrics for Architects

Alexander v. Zitzewitz
a.zitzewitz@hello2morrow.com
blog.hello2morrow.com
@AZ_hello2morrow

Agenda

- The case for using metrics
- Fitness function
- Some useful code metrics

Continuous Improvement



Why you should use metrics

- They are the foundation of the crucial “verify/measure” node of the continuous improvement loop
- Free tools like Sonargraph-Explorer already provide a lot of metrics
- Automated measurement in CI builds allows you to discover harmful trends early enough
- You can enforce quality standards by using metrics in quality gates

Why metrics are underutilized

- Perceived lack of tools or knowledge about them
- Lack of knowledge about metrics and how to read them
- Often a single metric does not tell the whole story
- Who has time for this?
- Metrics are most useful when they are used to trigger actions
- Intimidating choices, which metrics should I use?

Agenda

- The case for using metrics
- Fitness function
- Some useful code metrics

Metrics quantify how well you met your goals

- Without measuring you are blind
- Trust is good – control is better (Lenin)
- But what are the goals?

- Maintainability?
- Scalability?
- Performance?
- Evolvability?
- Testability?
- Many more “ilities”…

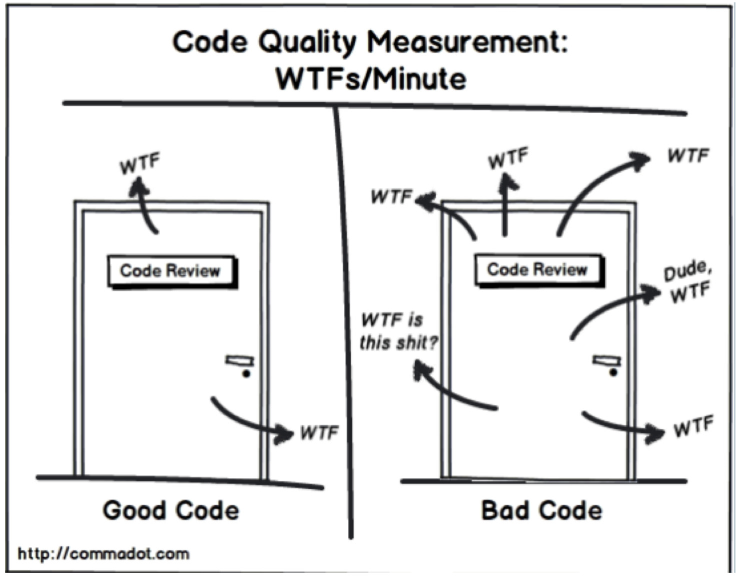
Prioritize goals

- Pick 3 to 4 “-ilities” as your top goals
- Maintainability should always be one of them (unless you write code that will never change)
- Define and quantify what it means to achieve a goal

Fitness functions

- Define how well you achieved your goal
- Can be based on
  - Code metrics derived from static analysis
  - Operational metrics
  - Production metrics
  - Manually collected metrics
- Automation is recommended, but not always possible
- Here we focus on code metrics

A Manual Fitness Function

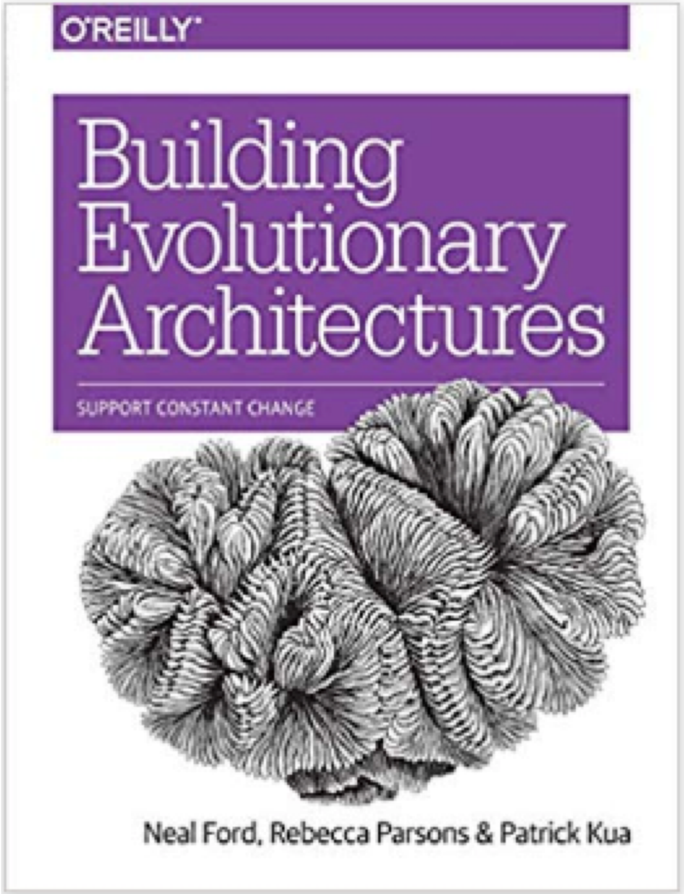


Example Fitness Function

Percentage of time used to develop new features

- Ratio of total development time spent on new features
- Measures agility/changeability
- Can be derived from extracting data from issue tracking systems like Jira
- Requires developers to properly enter times used on issues (operational maturity)
- Requires proper issue categorization (operational maturity)
- Can be automated
- Indirect measurement of technical debt

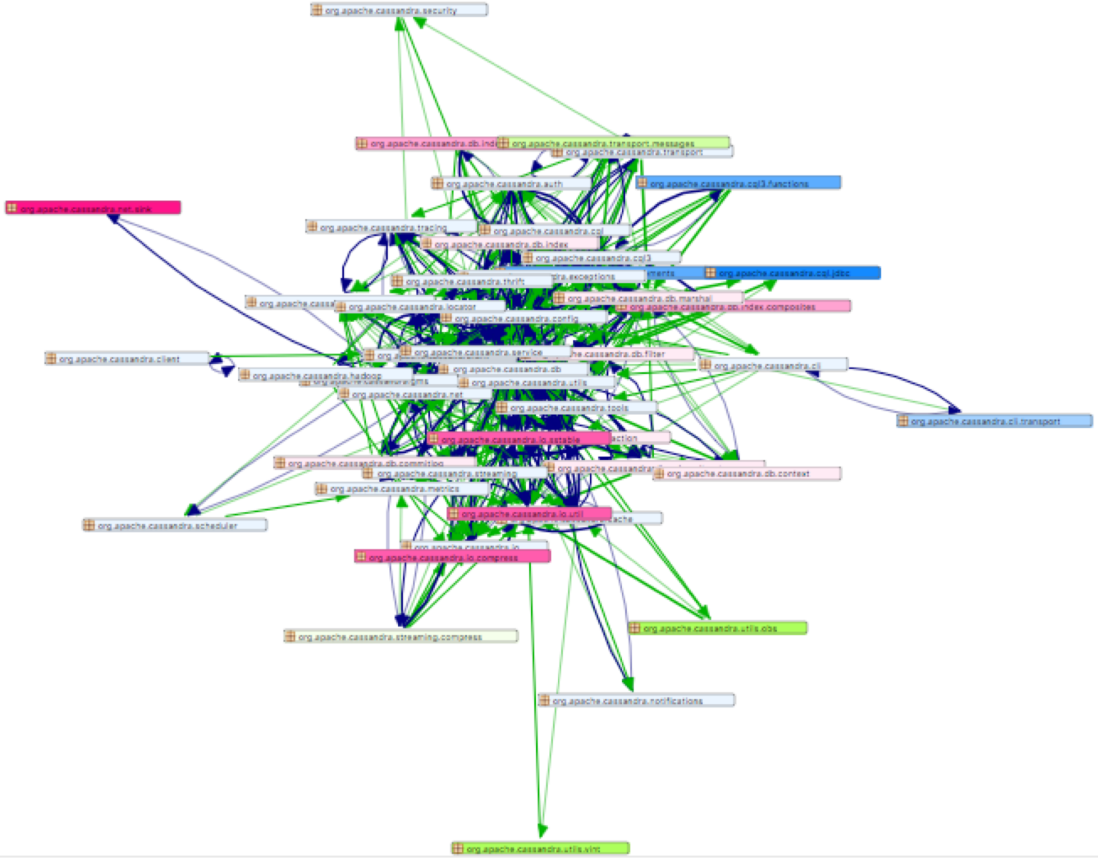
More about fitness functions



Agenda

- The case for using metrics
- Fitness function
- Some useful code metrics

This could have been avoided using metrics



Architecture of Apache-Cassandra (ML: 9%, PC: 62%)

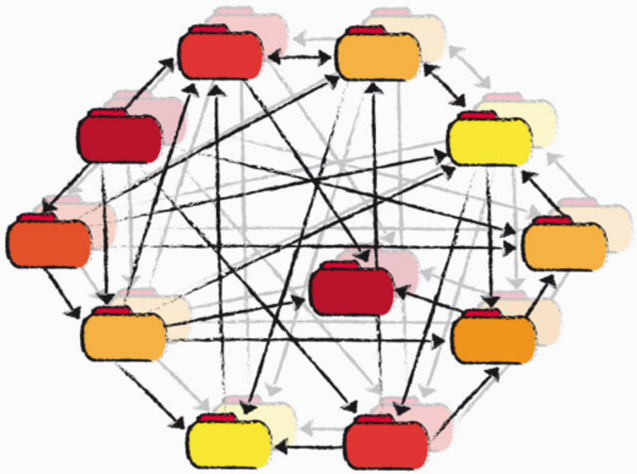
How can we measure Spaghettization?

Attributes of Spaghetti Code?

- High coupling
- Lots of cyclic dependencies
- No clear separation of responsibilities, e.g. features are spread all over the place
- Sounds familiar ?! 90% of systems suffer some some variety of this problem

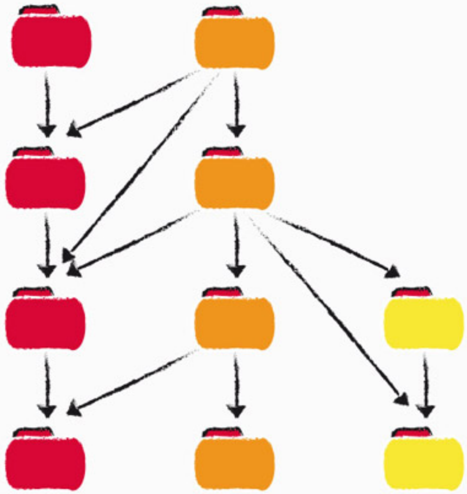
Spaghetti Code vs Clean Code

CHANCES ARE YOUR CODE LOOKS LIKE THIS:



- Much reduced team velocity
- Frequent regression bugs
- Hard to maintain, test and understand
- Modularization is impossible

ORGANIZED CODE LOOKS MORE LIKE THIS:



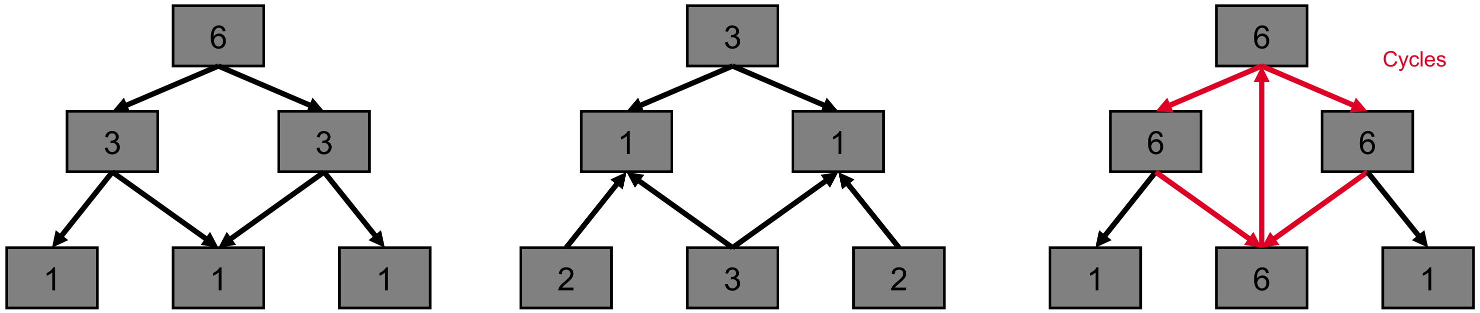
- Much lower cost of change
- Easier to maintain, test and understand
- Improved developer productivity
- Lower risk

Good metrics to identify Spaghetti Code

- ACD (Average Component Dependency): measures coupling
- Maintainability Level: measures coupling and cyclic dependencies
- Relative Cyclicity: focus on cyclic dependencies
- Structural Debt Index (SDI): focus on cyclicity

ACD – a metric to measure coupling

- ACD = Average Component Dependency
- Average number of direct and indirect dependencies
- rACD = ACD / number of elements
- NCCD: normalized cumulated component dependency

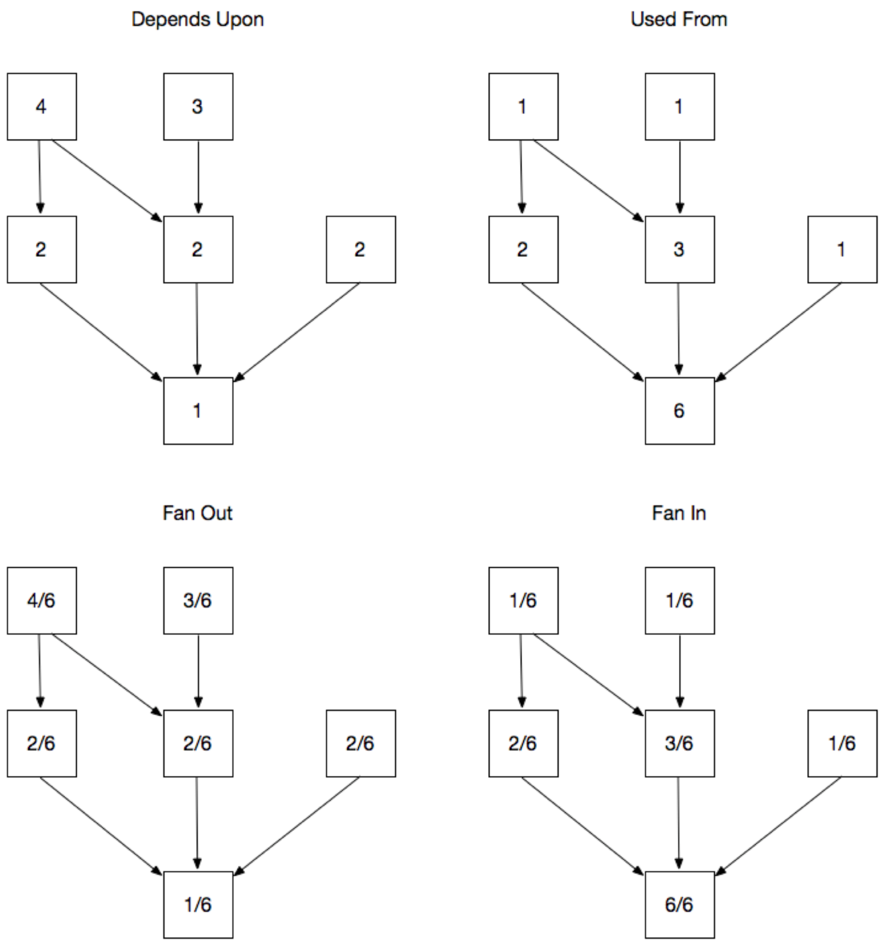


CCD = 15
ACD = 15/6 = 2,5

Dependency Inversion
ACD = 12/6 = 2

ACD = 26/6 = 4,33

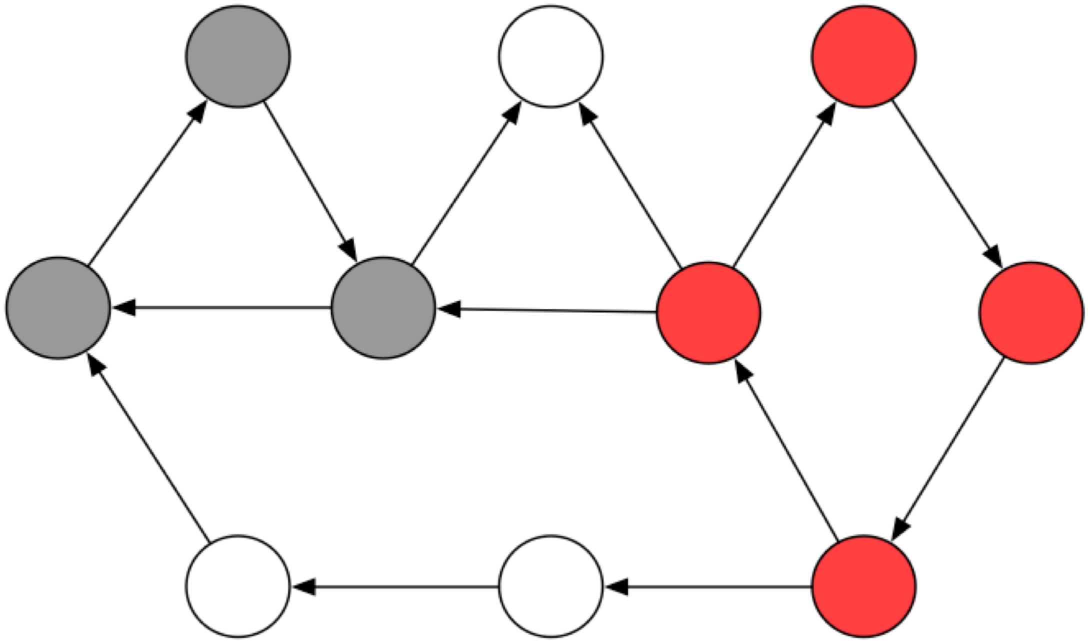
Low level metrics to measure coupling



Propagation Cost (MacCormack, Rusnak, Baldwin)

- Percentage value to indicate coupling
- Smallest value is 1/n*100, indicates no coupling
- Biggest value 100 means 100% coupling
- Calculated as average fan in (equals average fan out)
- Bad values are bad except for small systems
- Good values need to be verified by other metrics
- Usually shrinks with system size

What is a cycle group?



Cyclicity

- The cyclicity of a cycle group is the square of its size, e.g. a group with 3 elements has a cyclicity of 9.
- System / module cyclicity is the sum of all cycle group cyclicity values.
- Relative cyclicity is defined as:

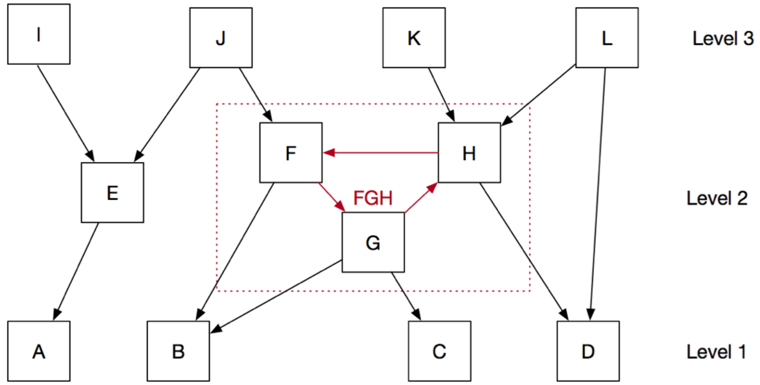
$$relativeCyclicity = 100 * \frac{\sqrt{sumOfCyclicity}}{n}$$

n is the total number of elements

Maintainability Level (ML)

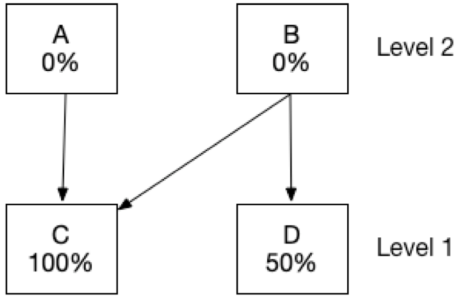
- Experimental metric in Sonargraph
- Implemented as percentage: 100% means no coupling
- Should be stable, when there are no major changes to architecture and design
- Measure decoupling and successful verticalization
- Reducing coupling and cyclicity will improve metric
- One of several indicators of design quality
- Recommended value: 75% or more

ML Implementation

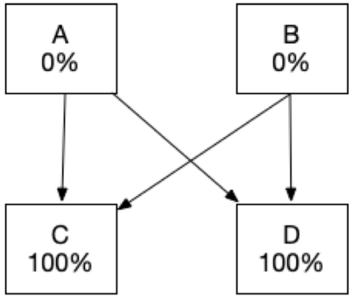


- Fan In (ML): percentage of higher-level components influenced by a given component
- E.g. A influences E, I and J, 3 of 8 higher level components. Its “Fan In (ML)” value therefore is 3/8 or 37.5%.
- Cycle groups are condensed into a single logical node. I the example F, G and H are condensed into a new node called FGH (weight of 3).
- Fan In (FGH) is ¾ = 75% (it influences J, K and L – 3 of 4 nodes in level 3.
- Fan-In (ML) of B, C and D is 6/8 = 75%. Cycle groups have a negative influence.
- Fan in of elements in highest level is always 0.

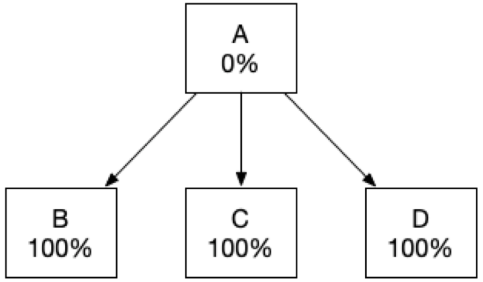
ML Example Calculations



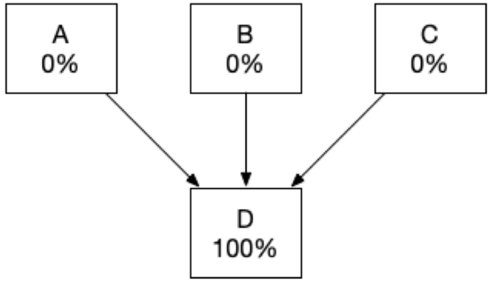
ML = (100% - 0%)/4
+ (100% - 0%)/4
+ (100% - 100%)/4
+(100% - 50%)/4
= 62.5%



ML = 50%



ML = 25%



ML = 75%

ML Observations

- The more components are in the topmost level, the better. Those components can be changed without influencing the rest of the system.
- Cycle groups have a negative influence, especially when the have more than 5 elements.
- Successful verticalization (minimizing dependencies between vertical silos) leads to better values.
- We added an alternative calculation measuring package cyclicity. The ML value of a module is the minimum of both values.
- Does not work very well for small number of nodes. Therefore we introduced a sliding minimum.

Finetuning ML

- Penalty for cycle groups with more than 5 elements
- Does not work too well for small modules with less than 100 components
  - Fixed by introducing a sliding minimum value for modules with less than 100 components
- Metric is blind regarding package/namespace structure
  - Fixed by adding an alternative calculation (1 – relativePackageCyclicity) and then using the minimum value between this value and ML
  - Also added a sliding minimum value for modules with less than 20 packages/namespaces
- System wide metric is calculated as the weighted average of the largest modules

ML for Nerds

- Details can be found on blog.hello2morrow.com

http://blog.hello2morrow.com/2018/12/a-promising-new-metric-to-track-maintainability

Structural Debt Index (Sonargraph)

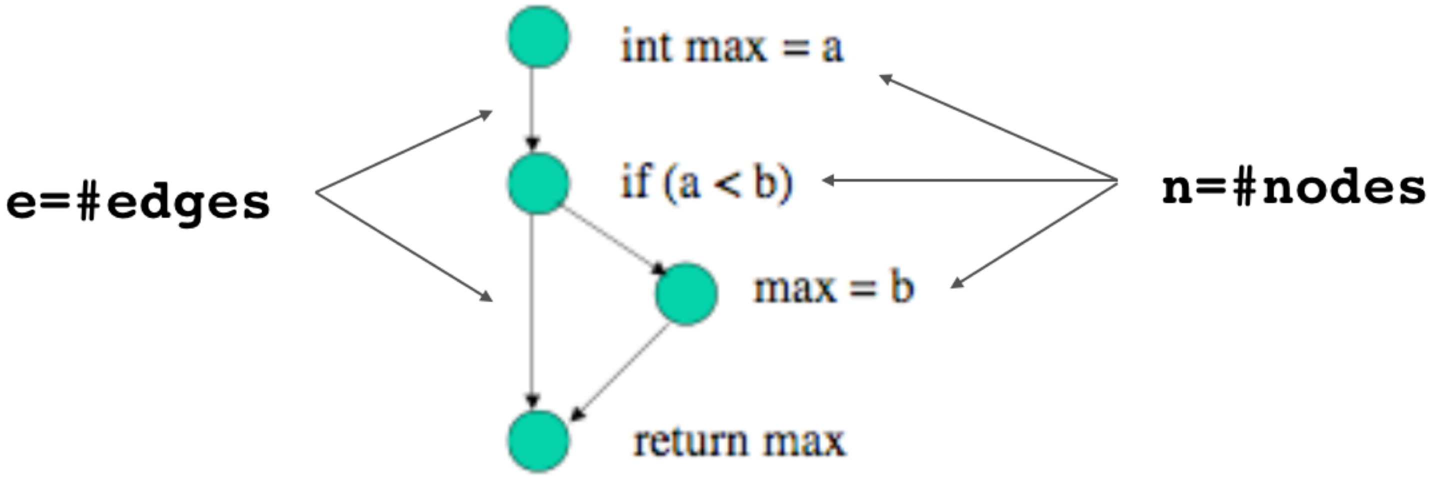
- This metric focusses on cyclic coupling and how difficult it would be to break the cycles
- Cyclic dependencies are a good indicator of structural erosion
- For each cycle group two values are computed:
  - How many links do I have to cut to break the cycle group
  - Total number of code lines affected by the links to break
- SDI = 10 * LinksToBreak + TotalAffectedLines
- SDI is then added up for modules and the whole system
- Can be computed on component level and on package/namespace level

Cyclomatic Complexity

Defined as CC = e – n + 2

e: edges

n: nodes



Cyclomatic Complexity Variants

- Modified cyclomatic complexity: only adds one per switch statement
- Extended cyclomatic complexity: adds one per logical and/or in conditions

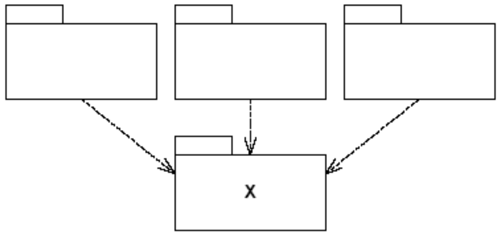
Average Cyclomatic Complexity

- Can be calculated on classes, packages/namespaces or modules
- Weighted average of cyclomatic complexity values of methods / classes.
- Use “number of statements” as weights

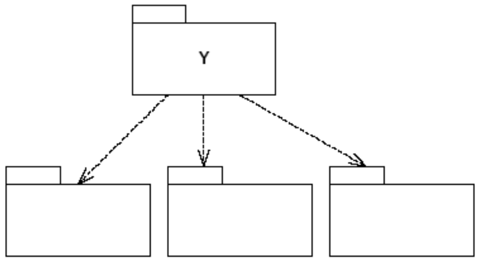
Max Indentation Depth

- Excellent complexity indicator
- Indentation >= 5 is problematic
- Average indentation = weighted average of max indentation depth

Architecture metrics of Robert C. Martin



X is „stable“



Y is „instable“

D_i = Number of incoming dependencies

D_o = Number of outgoing dependencies

Instability $I = D_o / (D_i+D_o)$

Build on abstractions, not on implementations

Abstractness (Robert C. Martin)

N_c = Total number of types in a type container

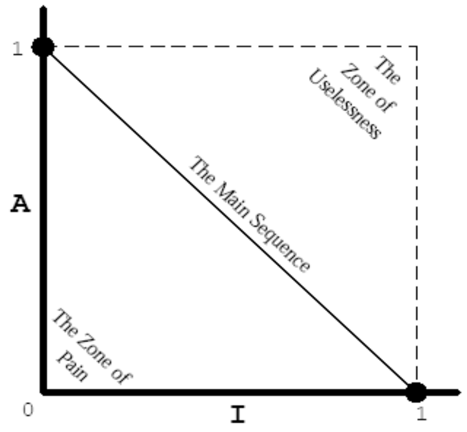
N_a = Number of abstract classes and interfaces in a type container

Abstractness $A = N_a/N_c$

Metric „distance“ (Robert C. Martin)

D = A + I – 1

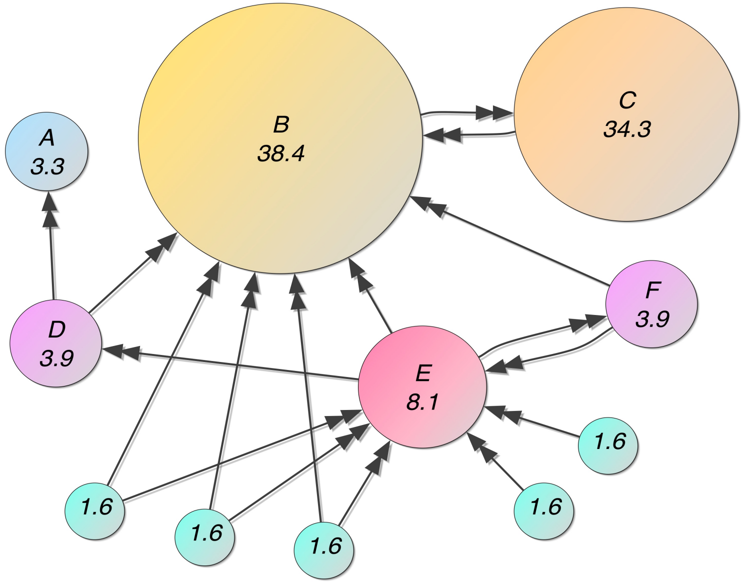
Value range [-1 .. +1]



- Negative values are in the „Zone of pain“
- Positive values belong to the „Zone of uselessness“
- Good values are close to zero (e.g. -0,25 to +0,25)
- „Distance“ is quite context sensitive

Component Rank

- Is based in Google's page rank metric
- Calculated iteratively until values stabilize
- Described in a Wikipedia article

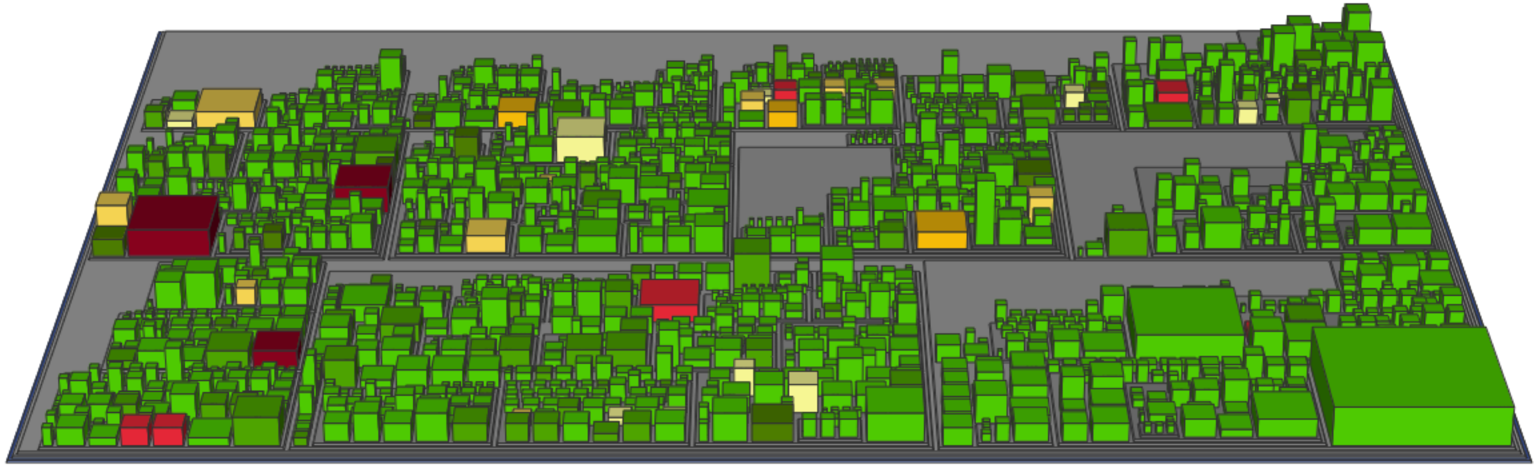


Source Code Management Metrics

- Very useful to identify useful refactorings
- Look for sources with high complexity and high change rate

- **File Changes (x days):** how often was a file committed in the last x days
- **Code Churn (x days):** how many lines have been added and removed from a file in the last x days
- **Code Churn Rate (x days):** percentage of code lines changed in the last x days based on total lines
- **Number of Authors (x days):** how many developers have worked on this file in the last x days

Hotspot Map



Software City rendered by Sonargraph

Q & A

a.zitzewitz@hello2morrow.com
blog.hello2morrow.com
@AZ_hello2morrow