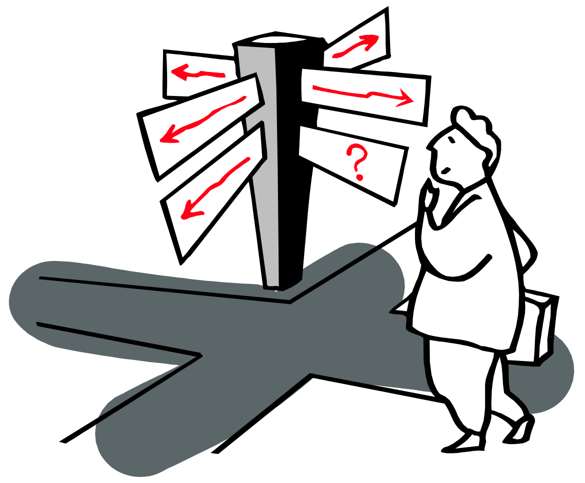
Software Reengineering P2: Code Smells and Evolution

Martin Pinzger
Delft University of Technology

Outline



Introduction

Problem detection in the source code

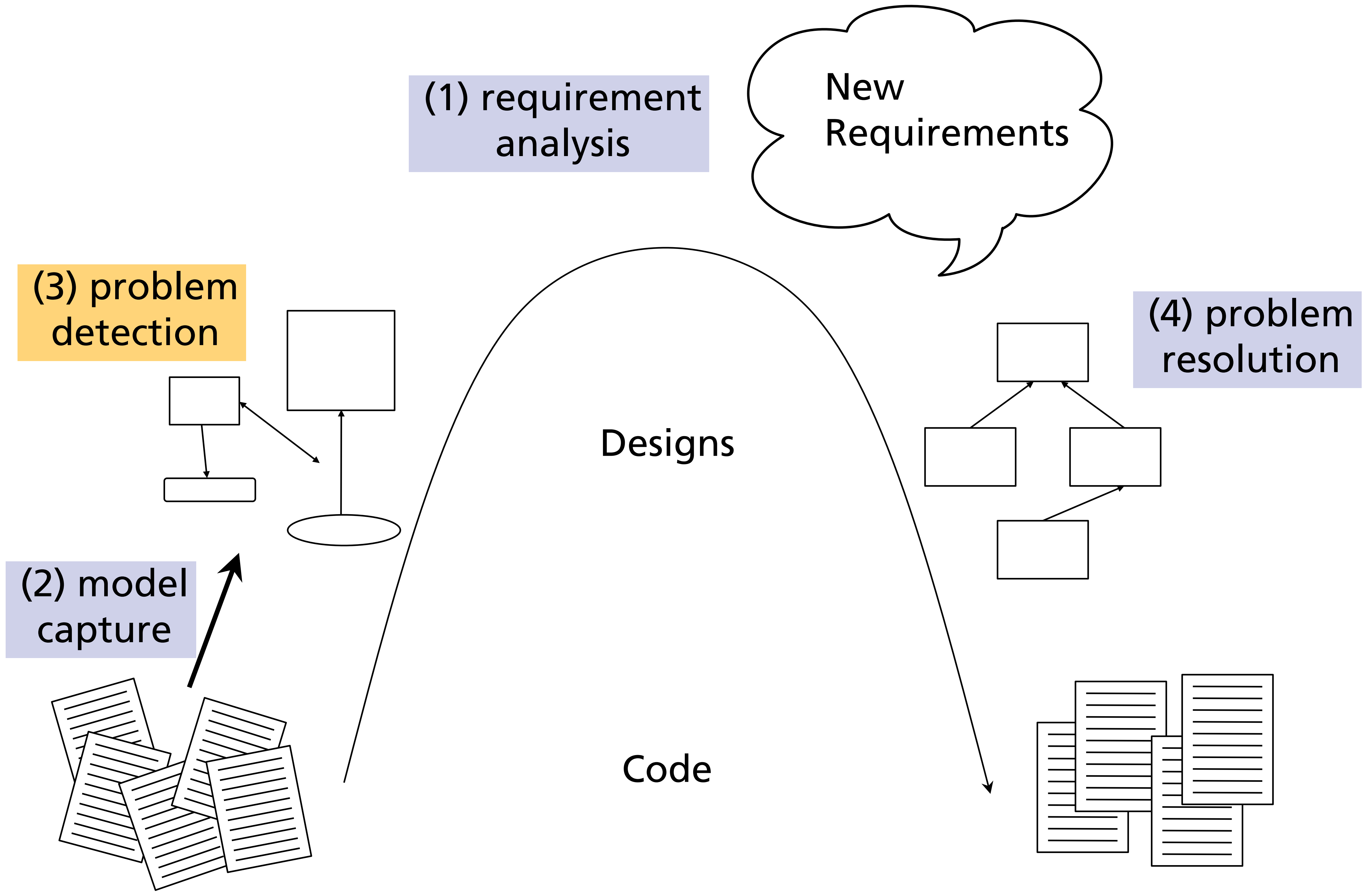
- Code Smells
- Polymetric Views

Problem detection in the evolution

- The Evolution Matrix
- Kiviat Graphs

Conclusions

The Reengineering Life-Cycle

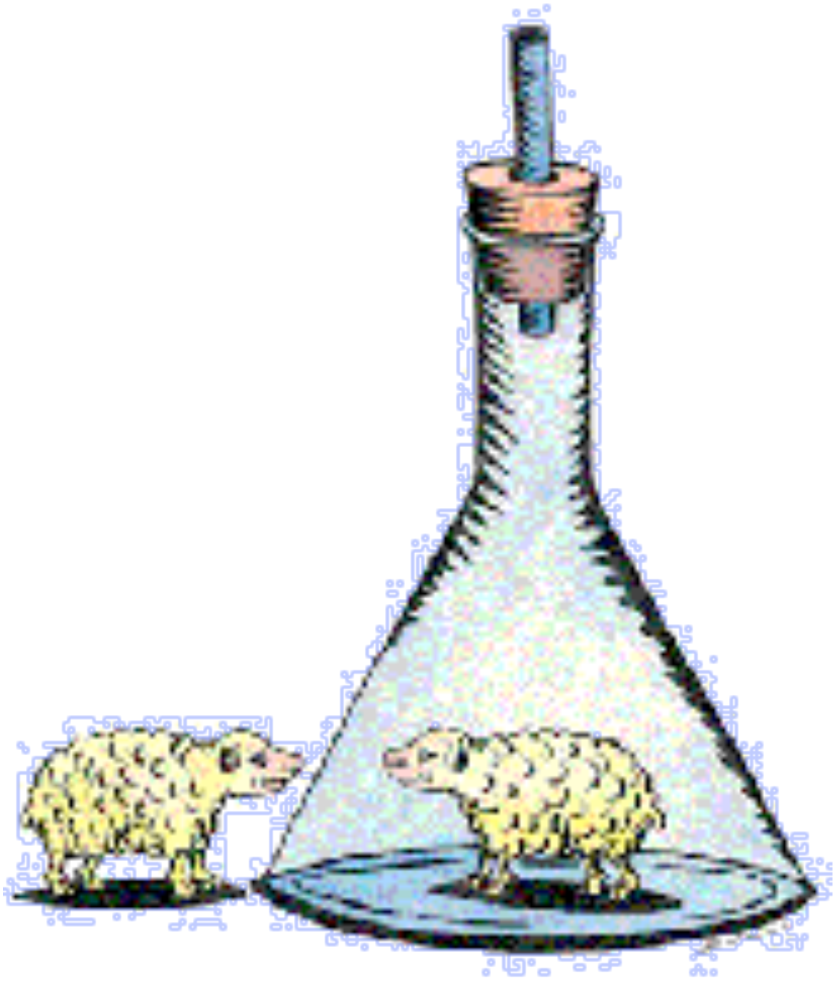


Design Problems

The most common design problems result from code that is

Unclear & complicated

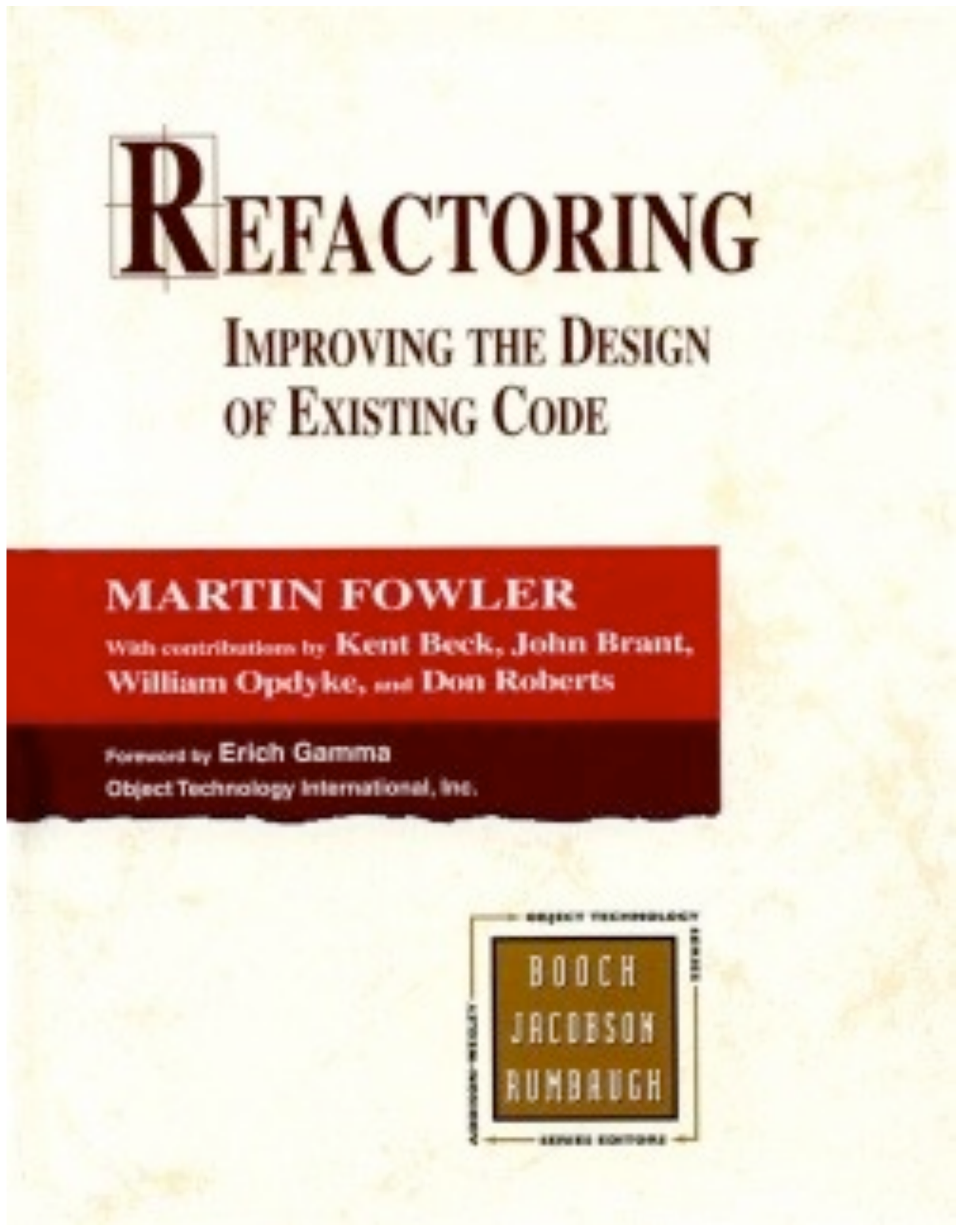
Duplicated (code clones)



Code Smells (if it stinks, change it)

A code smell is a hint that something has gone wrong somewhere in your code.

Duplicated Code
Long Method
Large Class
Long Parameter List
Divergent Change
Shotgun Surgery
Feature Envy
...


REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

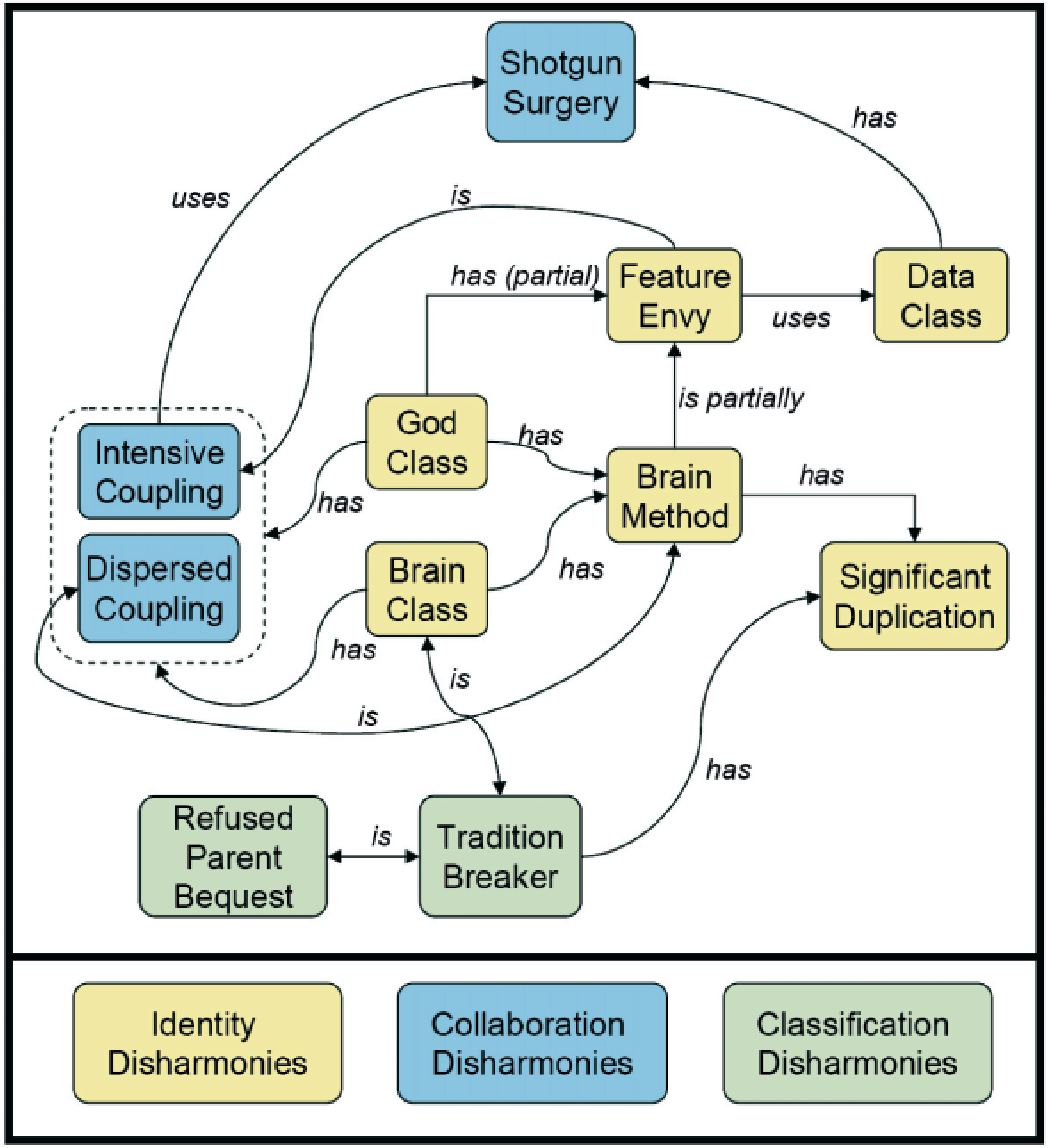
MARTIN FOWLER

With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International, Inc.

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
SERIES EDITORS

Design Disharmonies



Identity Disharmonies

Provide services and hide data

- A class should present itself to others only in terms of a set of provided services

Take responsibility

- Most non-abstract services of a class should be responsible for implementing a piece of the class’s functionality

Keep services cohesive

- Services provided by a class should be focused on one single responsibility

Be unique

- Each piece of concrete functionality is implemented once and only once

How To Detect?

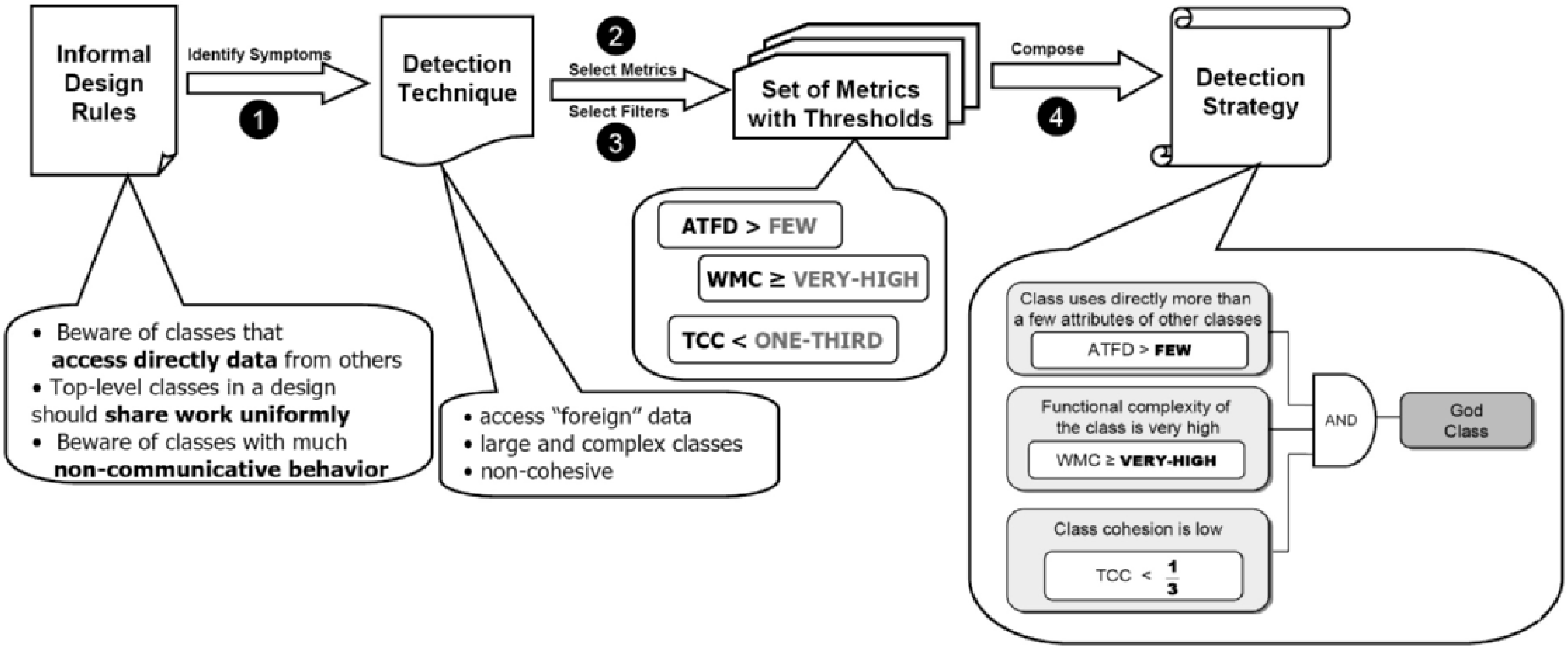
Measure and visualize quality aspects of the current implementation of a system

- Source code metrics and structures

Measure and visualize quality aspects of the evolution of a system

- Evolution metrics and structures

Detection Strategy - Overview

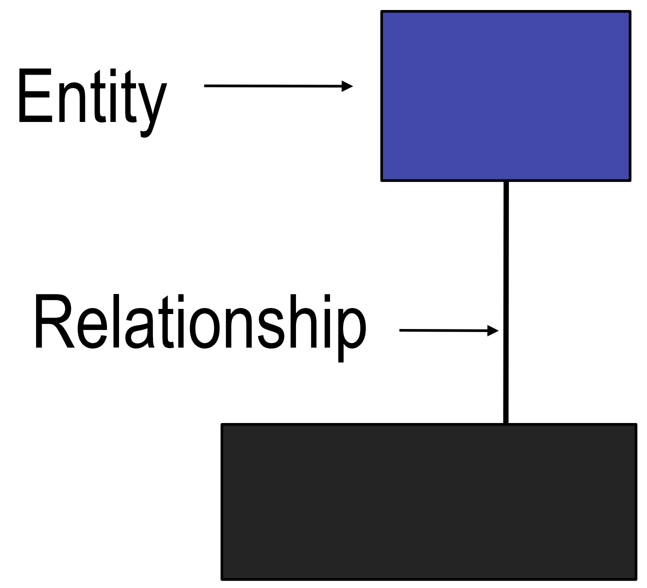

Informal Design Rules
Identify Symptoms
1
Detection Technique
2
Select Metrics
Select Filters
3
Set of Metrics with Thresholds
Compose
4
Detection Strategy
• Beware of classes that access directly data from others
• Top-level classes in a design should share work uniformly
• Beware of classes with much non-communicative behavior
• access "foreign" data
• large and complex classes
• non-cohesive
ATFD > FEW
WMC ≥ VERY-HIGH
TCC < ONE-THIRD
Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY-HIGH
Class cohesion is low
TCC < 1/3
AND
God Class

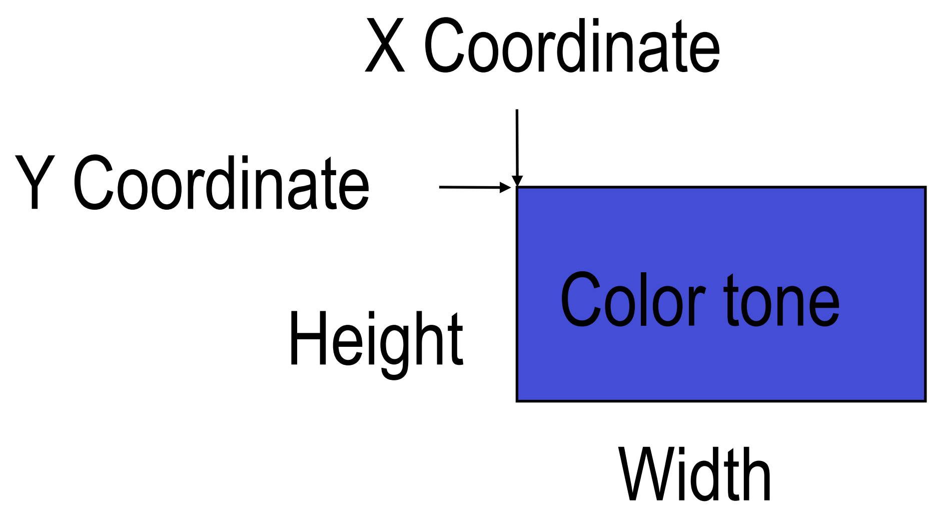
Simple Polymetric Views

Polymetric Views

A combination of metrics and software visualization

- Visualize software using colored rectangles for the entities and edges for the relationships
- Render up to five metrics on one node:
  - Size (1+2)
  - Color (3)
  - Position (4+5)





7

Smell 1: Long Method

The longer a method is, the more difficult it is to understand it.

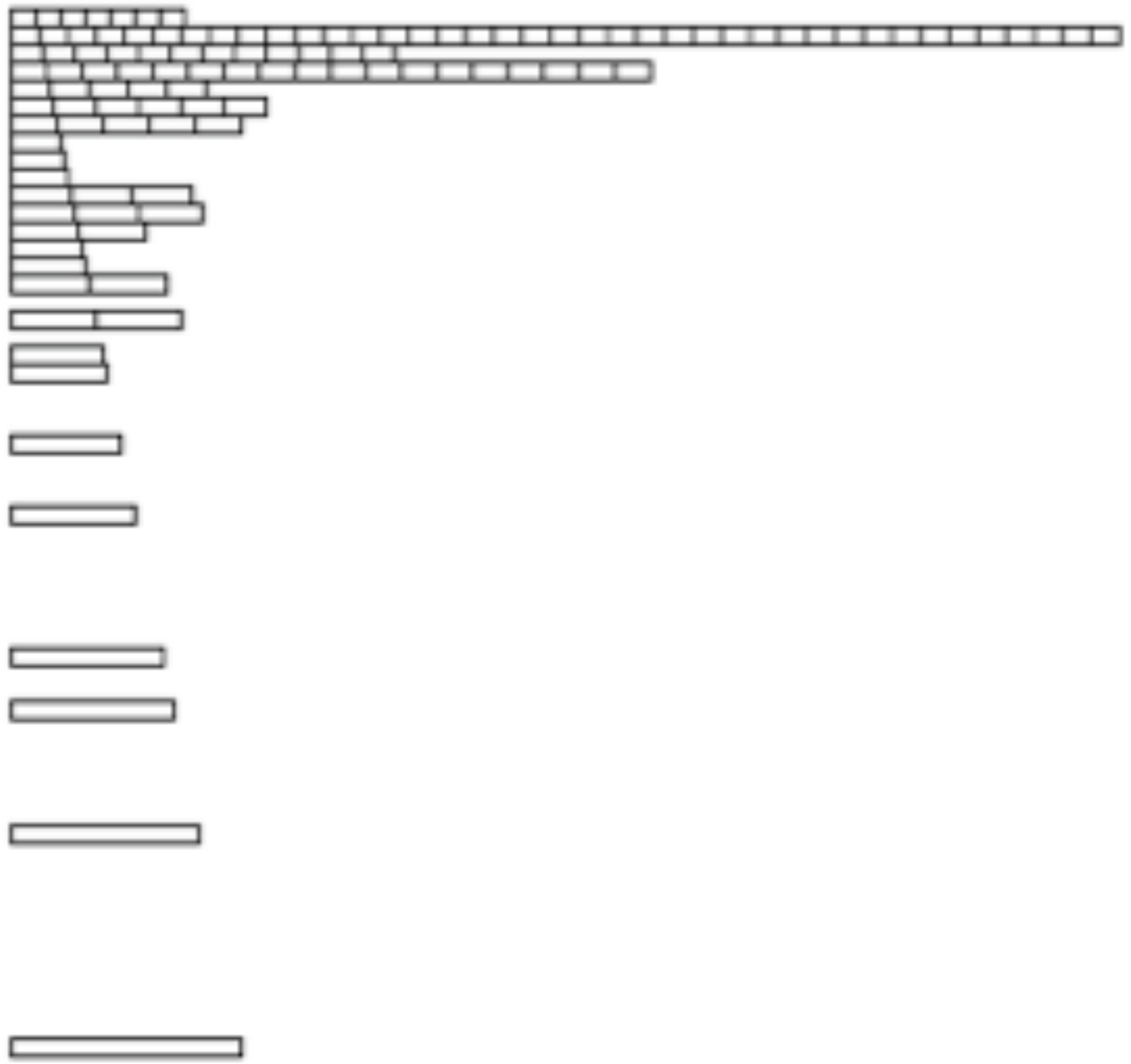
When is a method too long?

- Heuristic: > 10 LOCs (?)

How to detect?

- Visualize LOC metric values of methods
- “Method Length Distribution View”

Method Length Distribution



Metrics:
Boxes: Methods
Width: LOC
Position-Y: LOC
Sort: LOC

Smell 2: Switch Statement

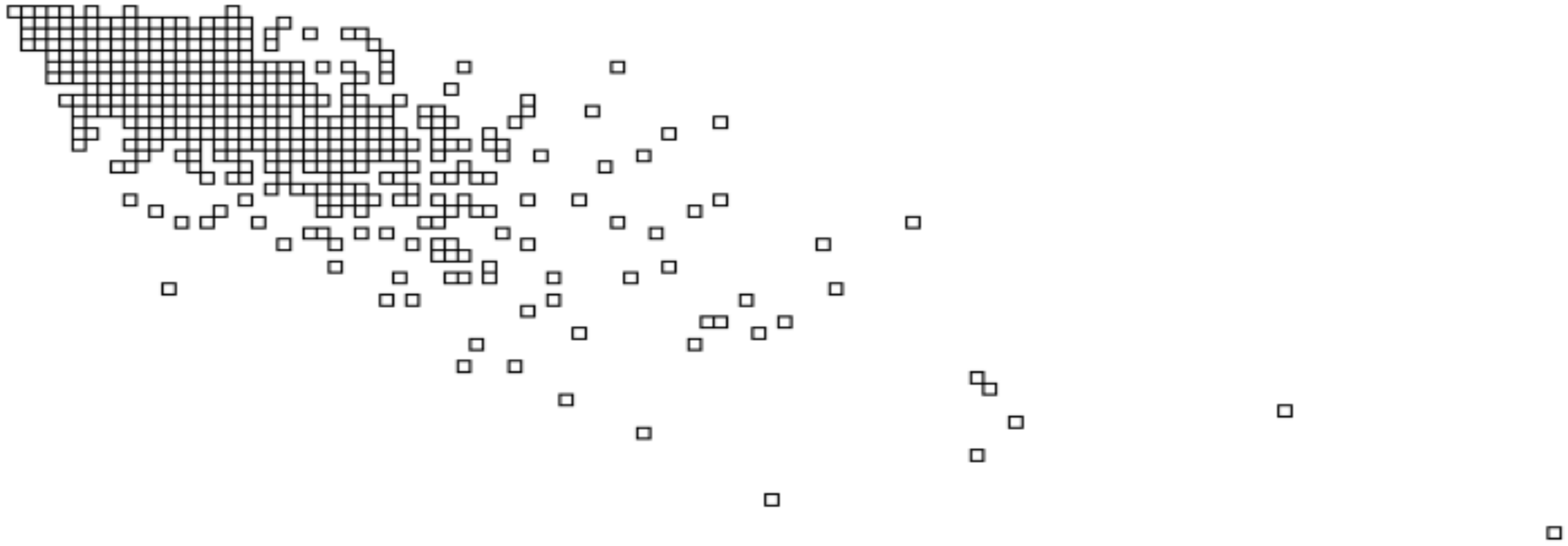
Problem is similar to code duplication

- Switch statement is scattered in different places

How to detect?

- Visualize McCabe Cyclomatic Complexity metric to detect complex methods
- “Method Complexity Distribution View”

Method Complexity



Metrics:

Boxes: Methods

Position-X: LOC

Position-Y: MCC

Sort: -

Smell 3: System Hotspots

Classes that contain too much responsibilities

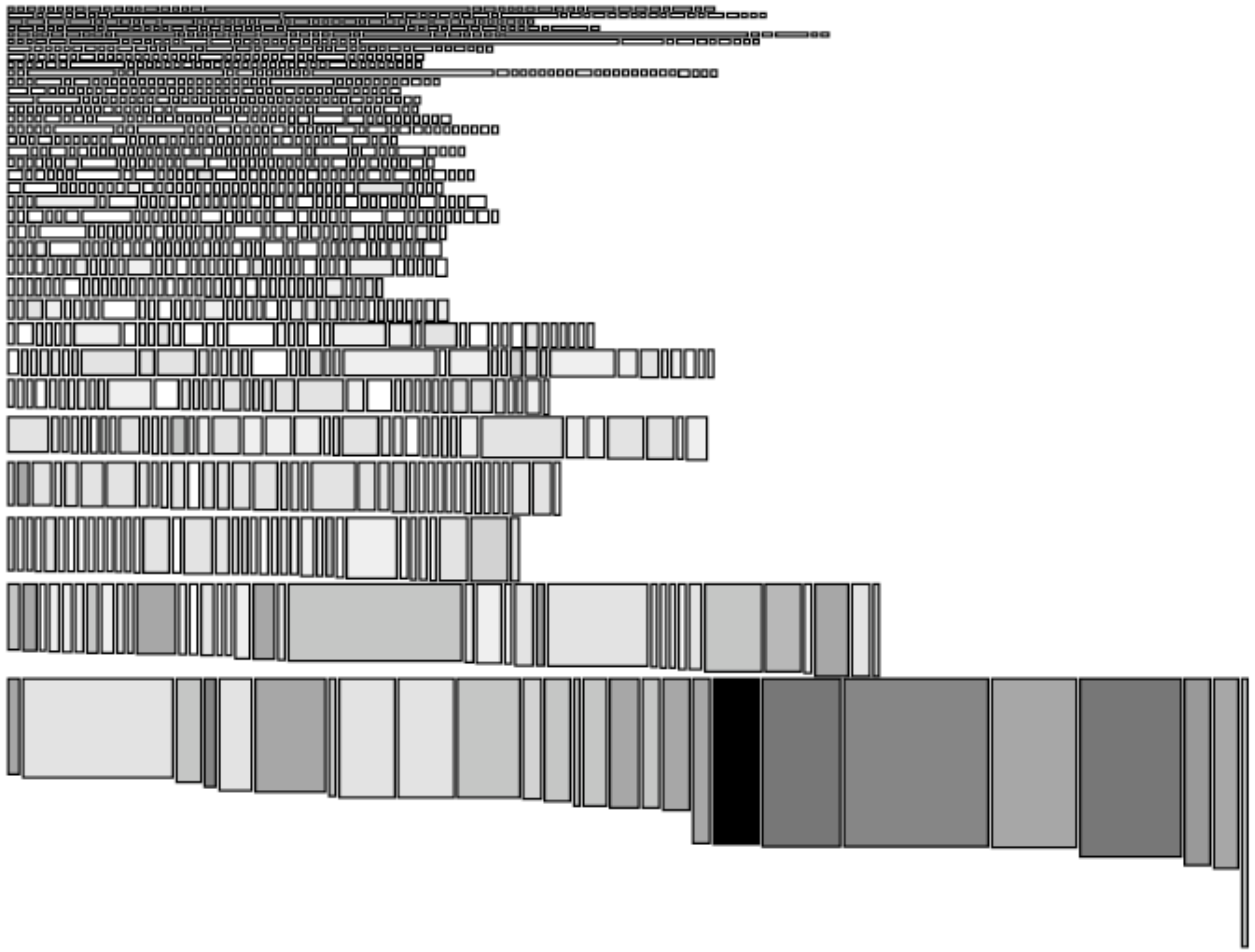
When is a class too large?

- Heuristic: > 20 NOM

How to detect?

- Visualize number of methods (NOM) and sum of lines of code of methods (WLOC)
- “System Hotspots View”

System Hotspots



Metrics:

Boxes: Classes

Width: NOA

Height: NOM

Color: LOC

Sort: NOM

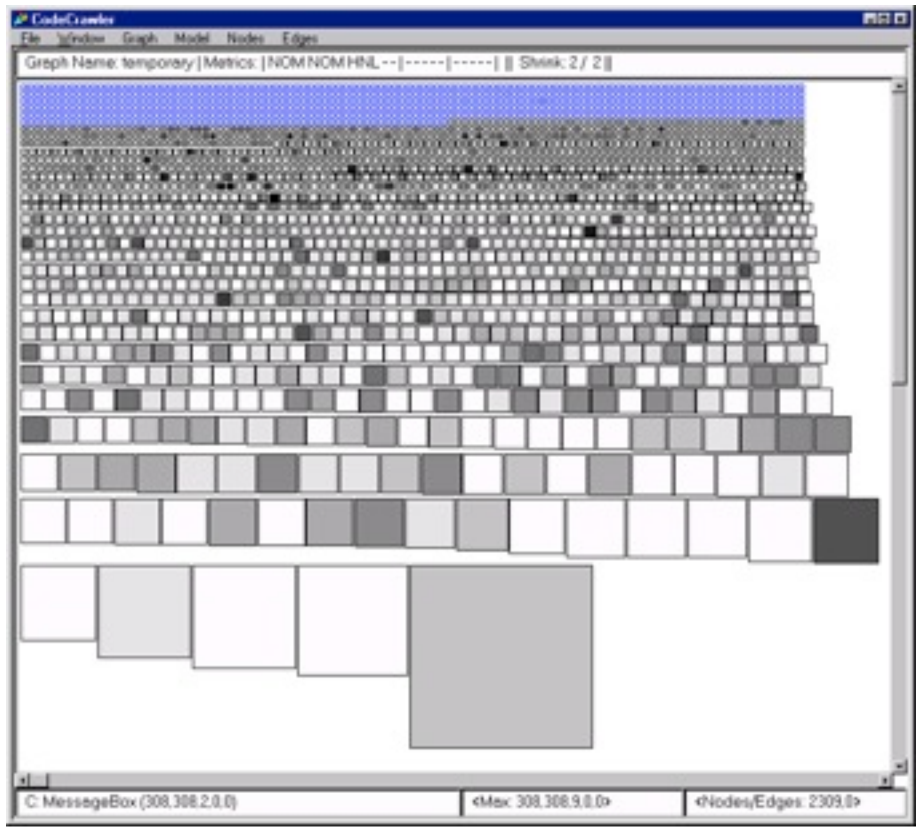
Evaluation: Polymetric Views

Pros

- Quick insights
- Scalable
- Metrics add semantics
- Interactivity makes the code “come nearer”
- Reproducible
- Industrial Validation is the acid test

Cons

- Level of granularity
- Code reading is needed



Graph Name: temporary |Metrics: |NOM NOM HNL --|-----|-----| || Shrink: 2 / 2||

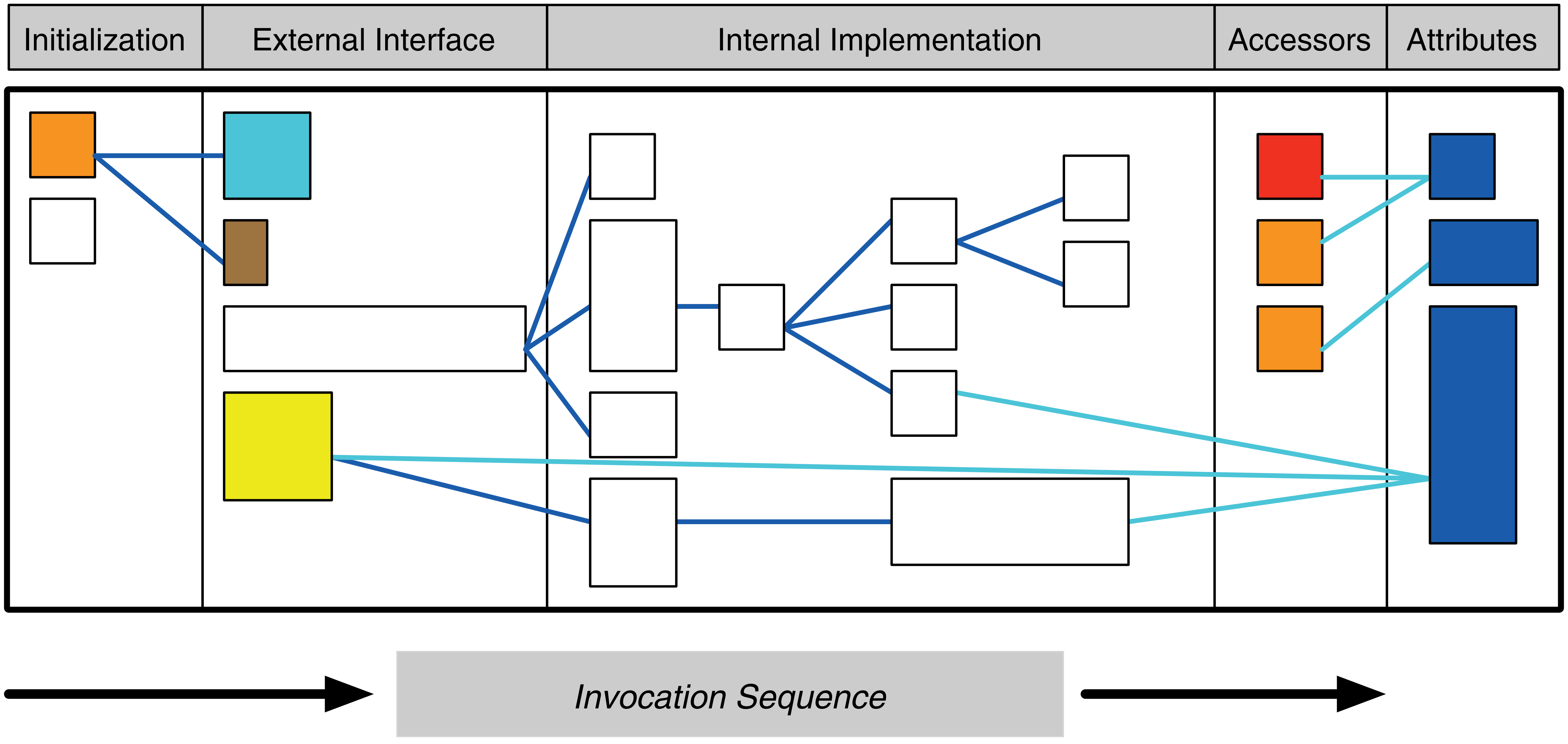
C: MessageBox (308,308,2,0,0)

<Max: 308,308,9,0,0>

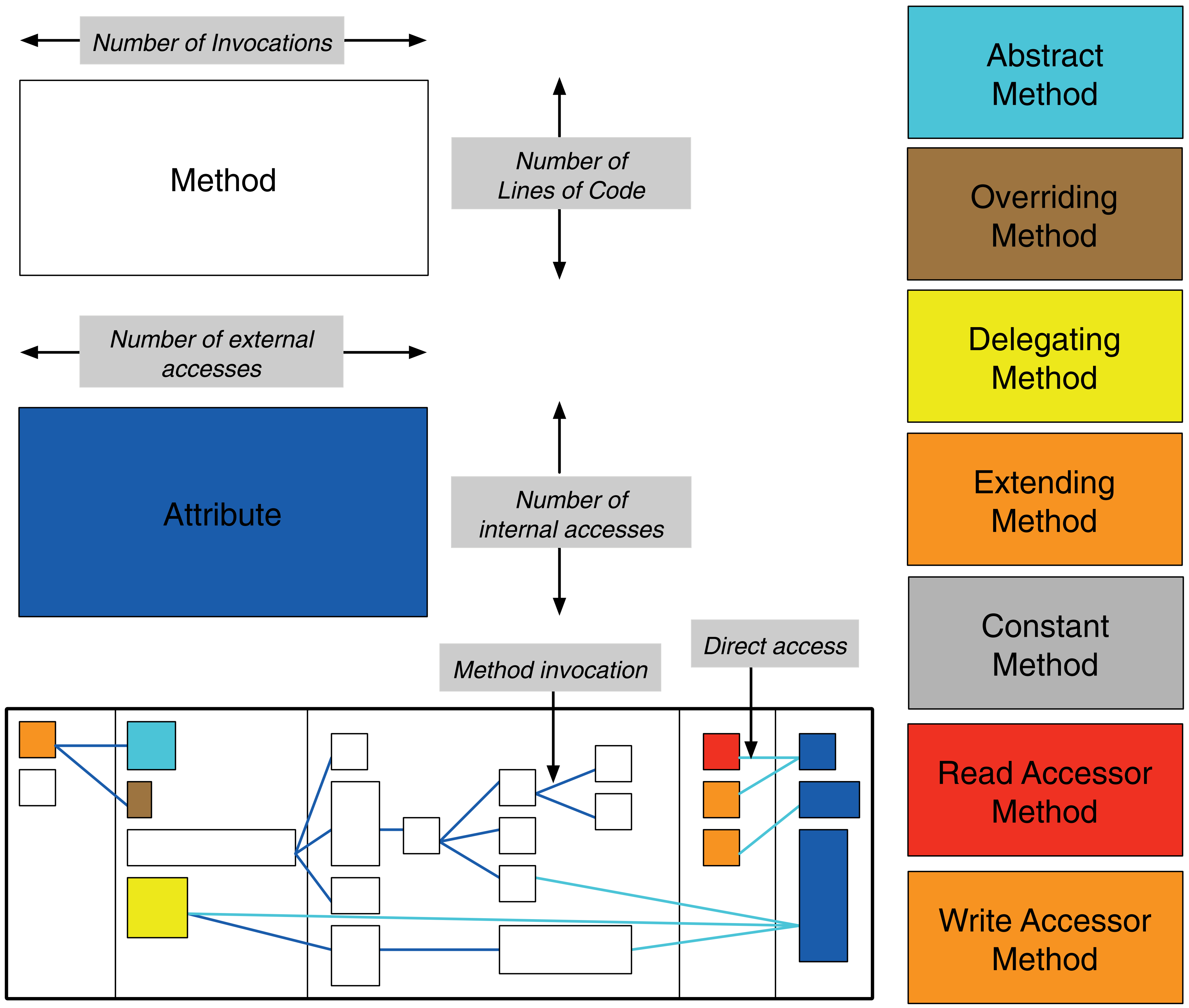
<Nodes/Edges: 2309,0>

Class Blueprint

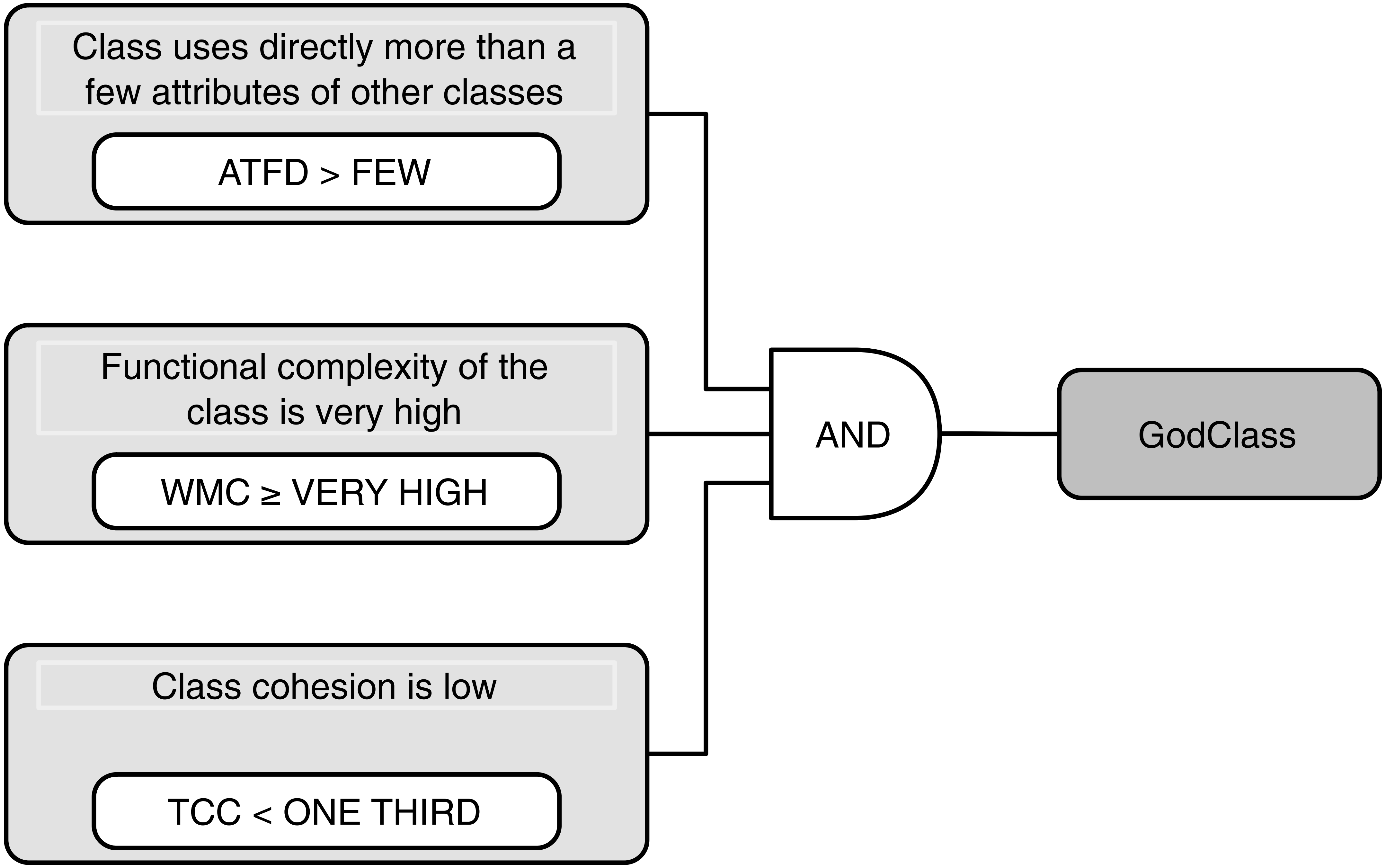
inCode - Class Blueprint



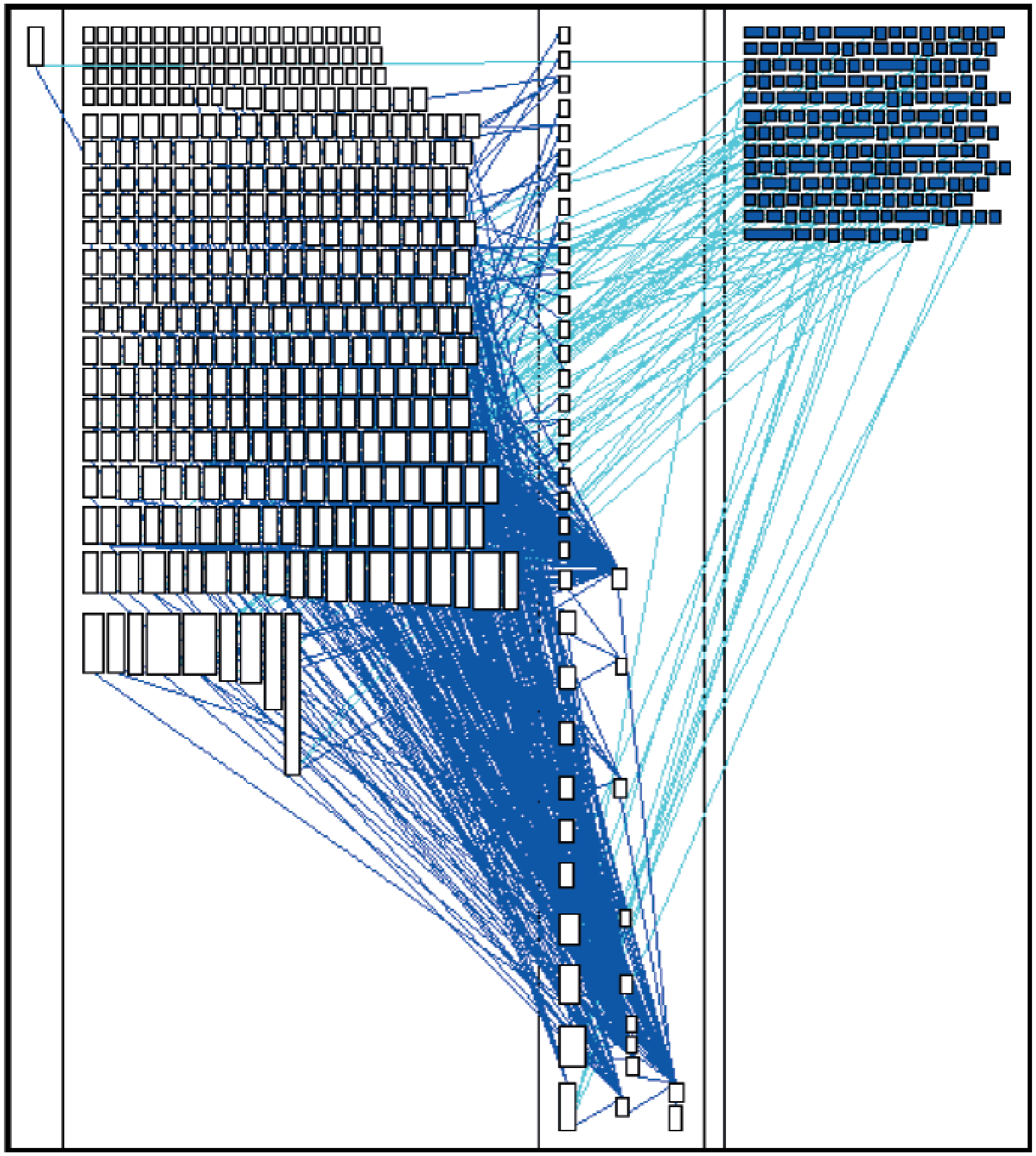
inCode - ClassBlueprint (cont.)



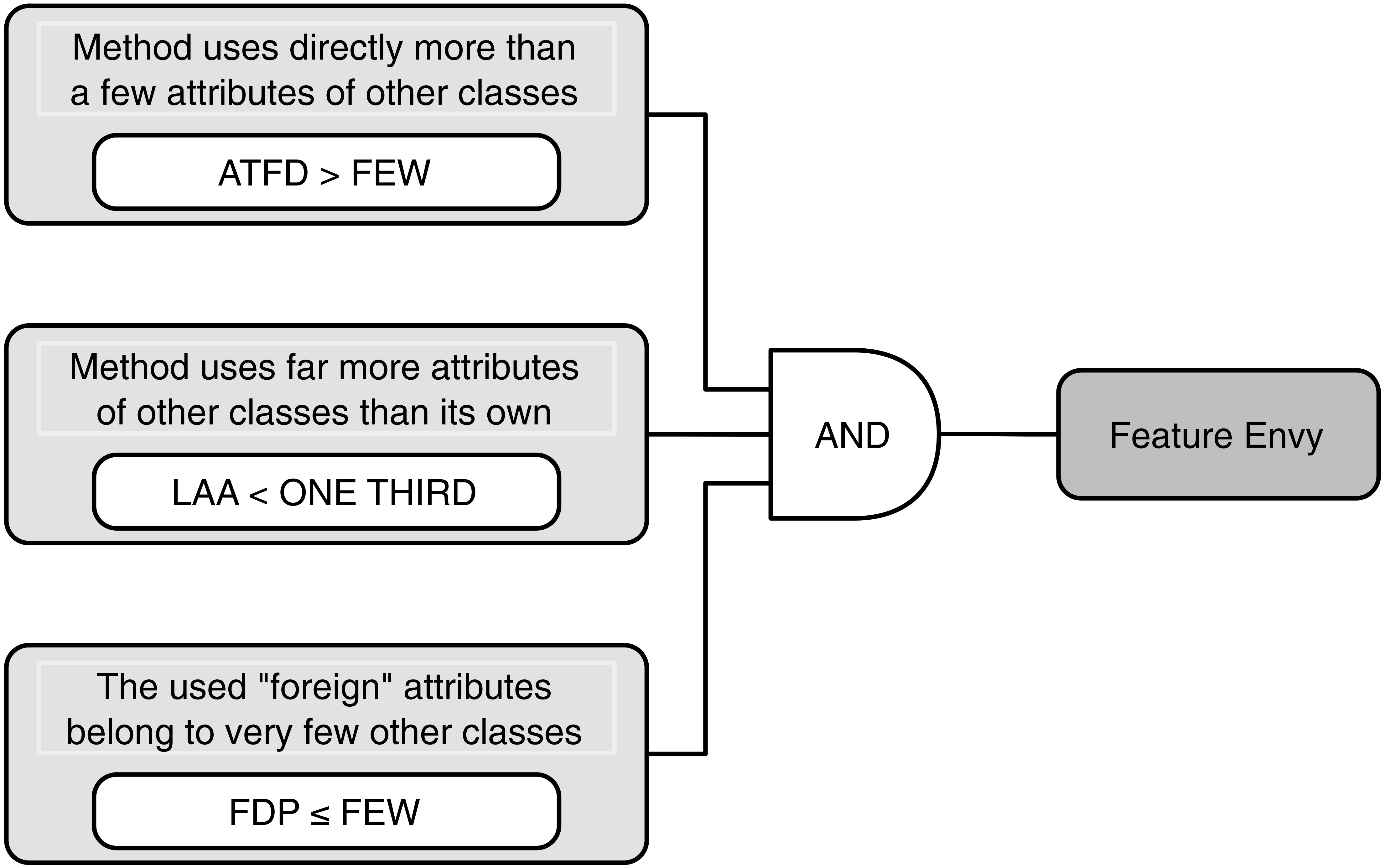
God Class


Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY HIGH
Class cohesion is low
TCC < ONE THIRD
AND
GodClass

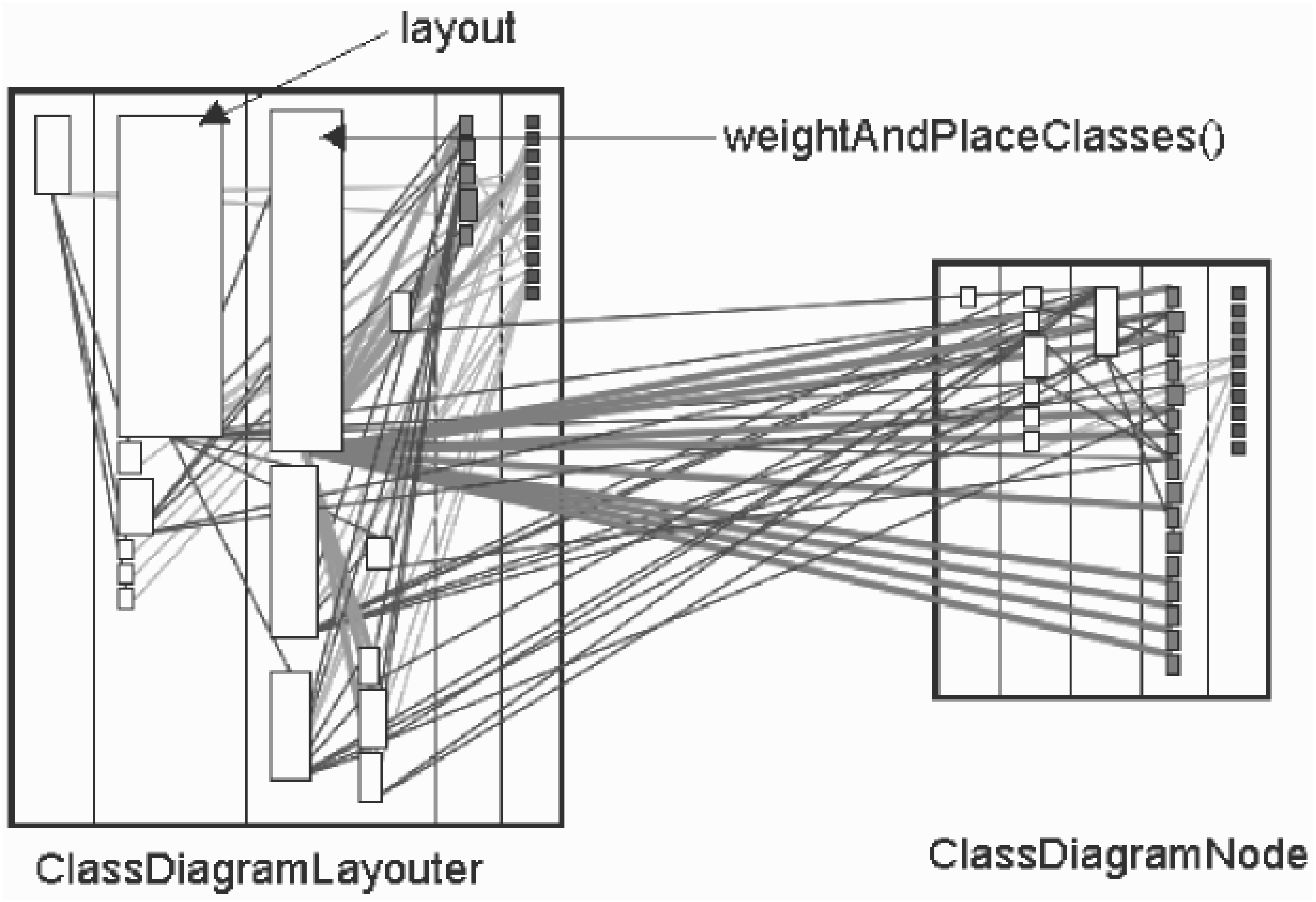
Example: God Class



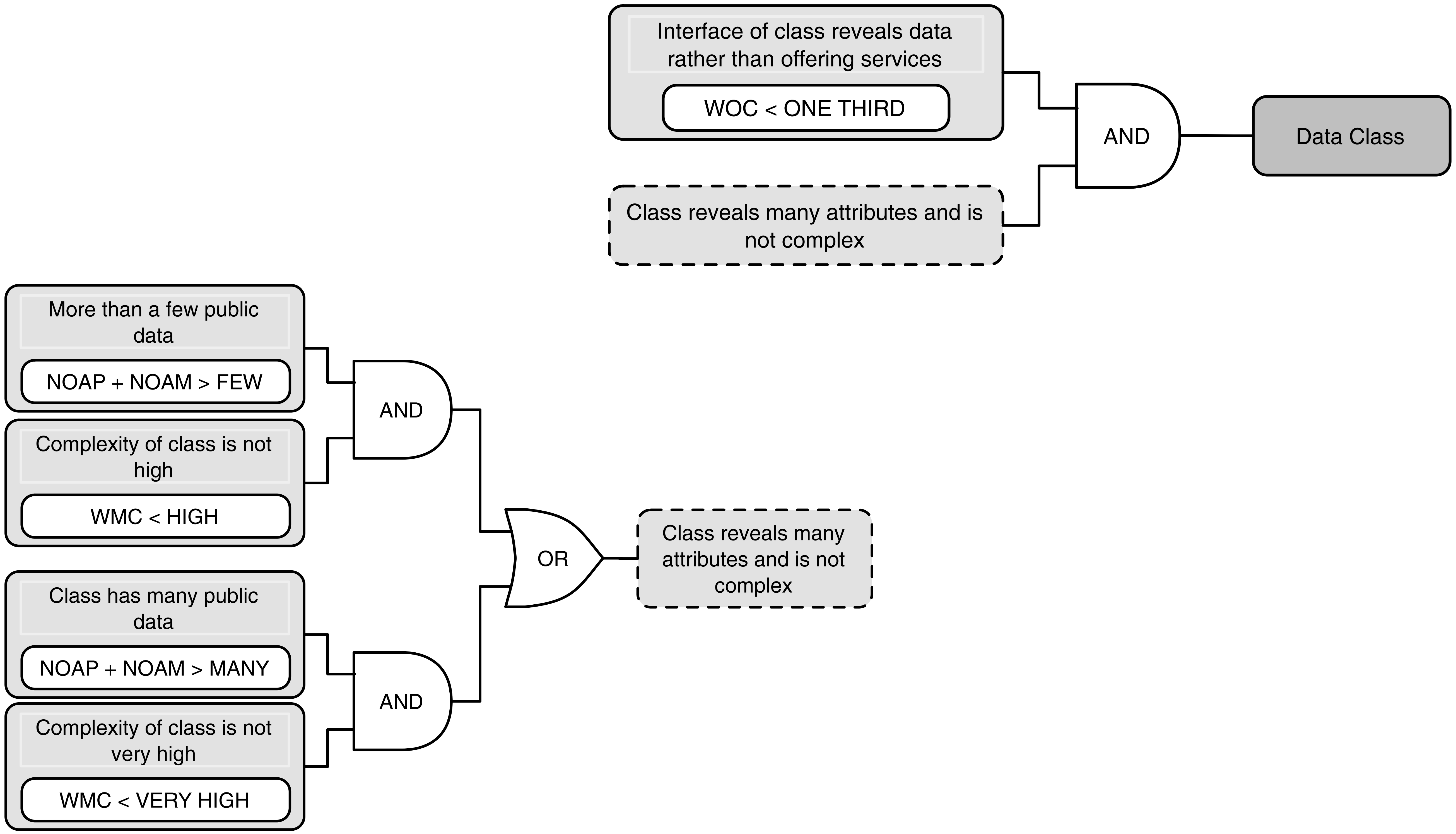
Feature Envy


Method uses directly more than a few attributes of other classes
ATFD > FEW
Method uses far more attributes of other classes than its own
LAA < ONE THIRD
The used "foreign" attributes belong to very few other classes
FDP ≤ FEW
AND
Feature Envy

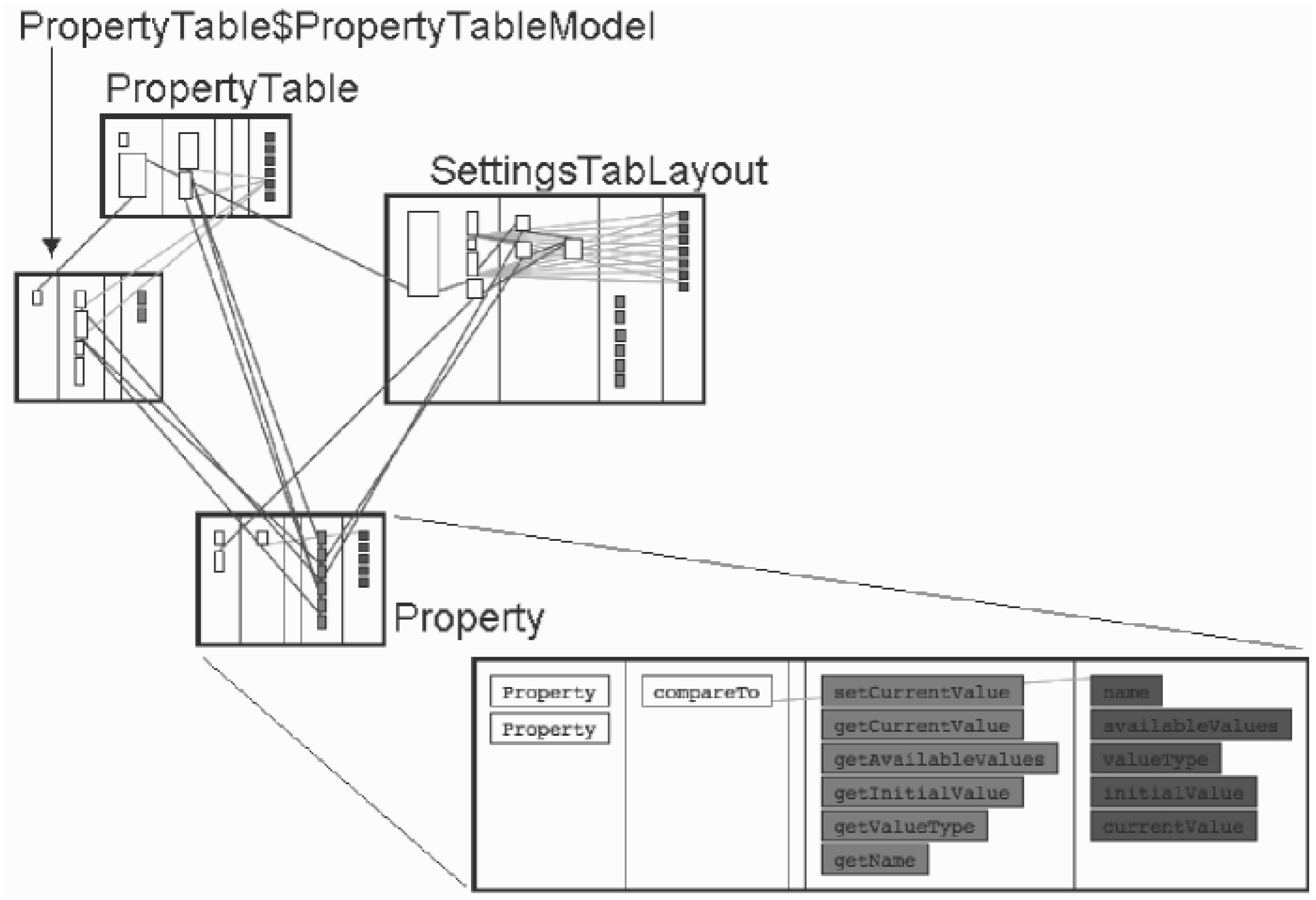
Example: Feature Envy



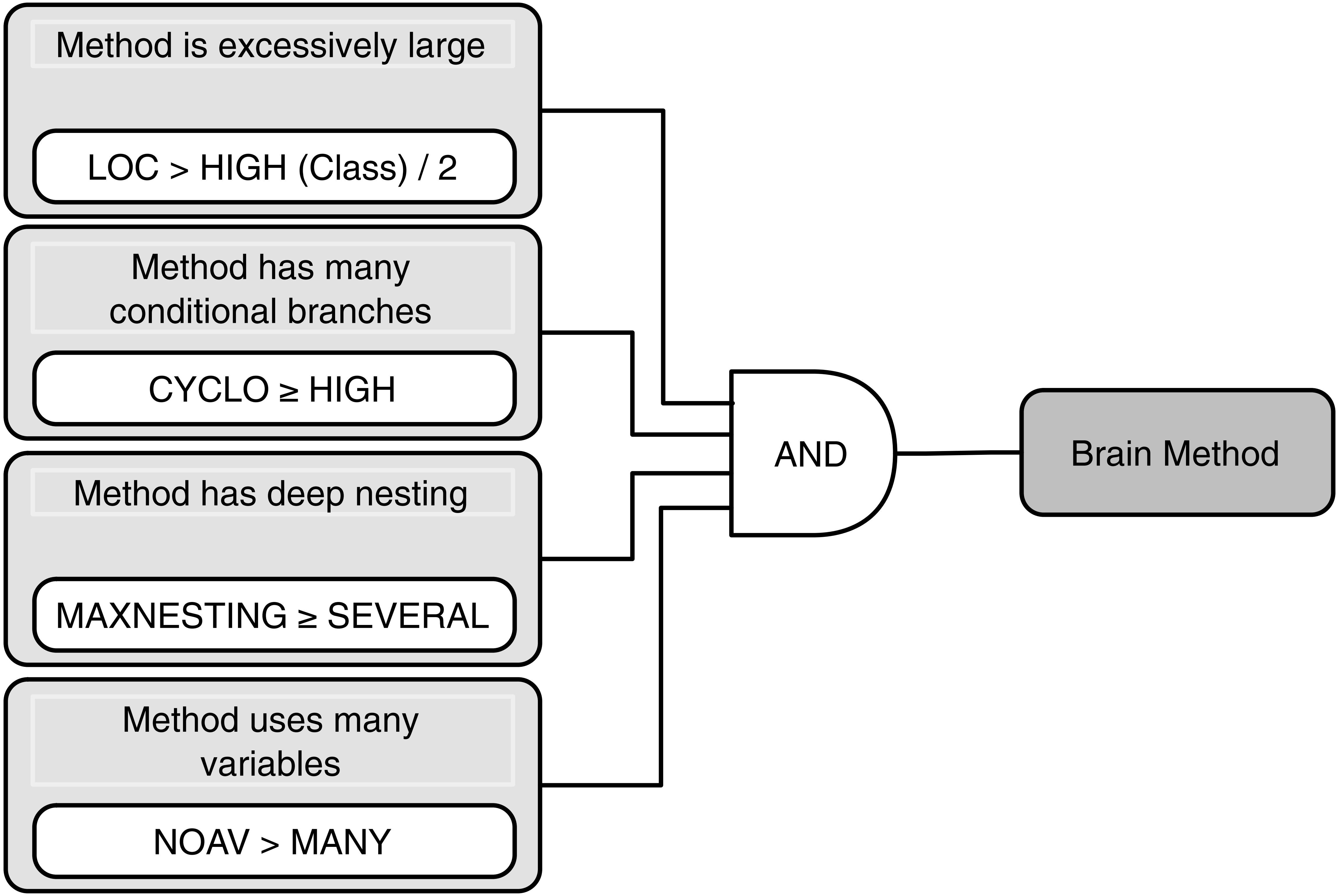
Data Class


Interface of class reveals data rather than offering services
WOC < ONE THIRD
AND
Data Class
Class reveals many attributes and is not complex
More than a few public data
NOAP + NOAM > FEW
AND
Complexity of class is not high
WMC < HIGH
OR
Class reveals many attributes and is not complex
Class has many public data
NOAP + NOAM > MANY
AND
Complexity of class is not very high
WMC < VERY HIGH

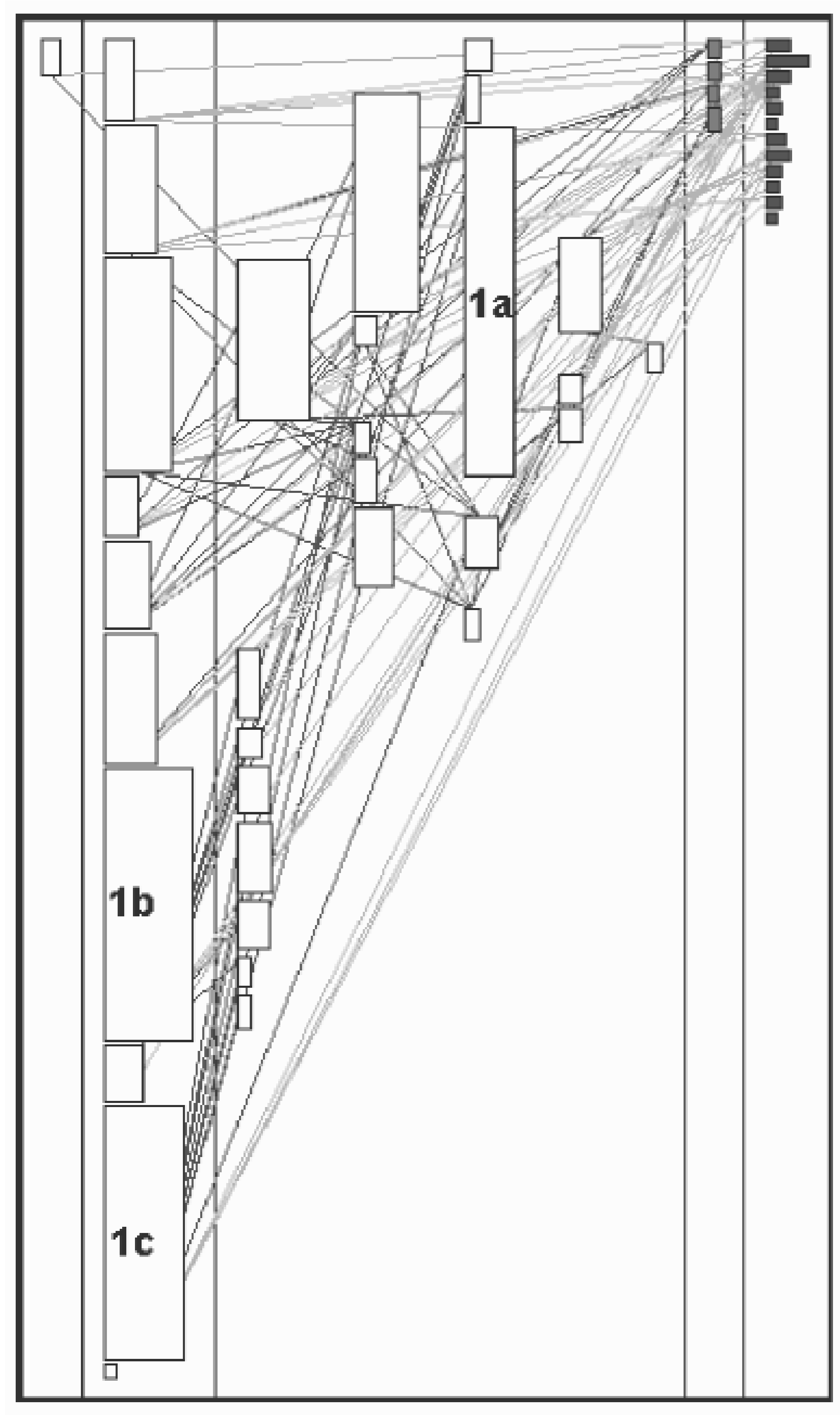
Example: Data Class



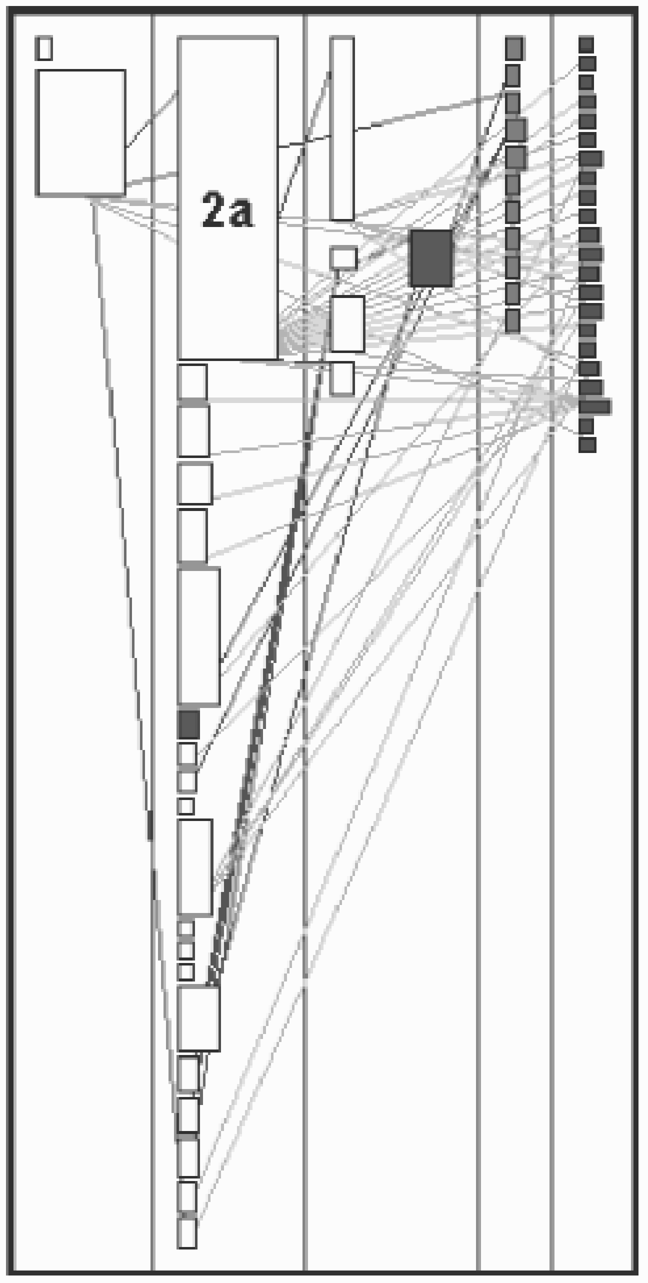
Brain Method



Example: Brain Method



Modeller



ProjectBrowser

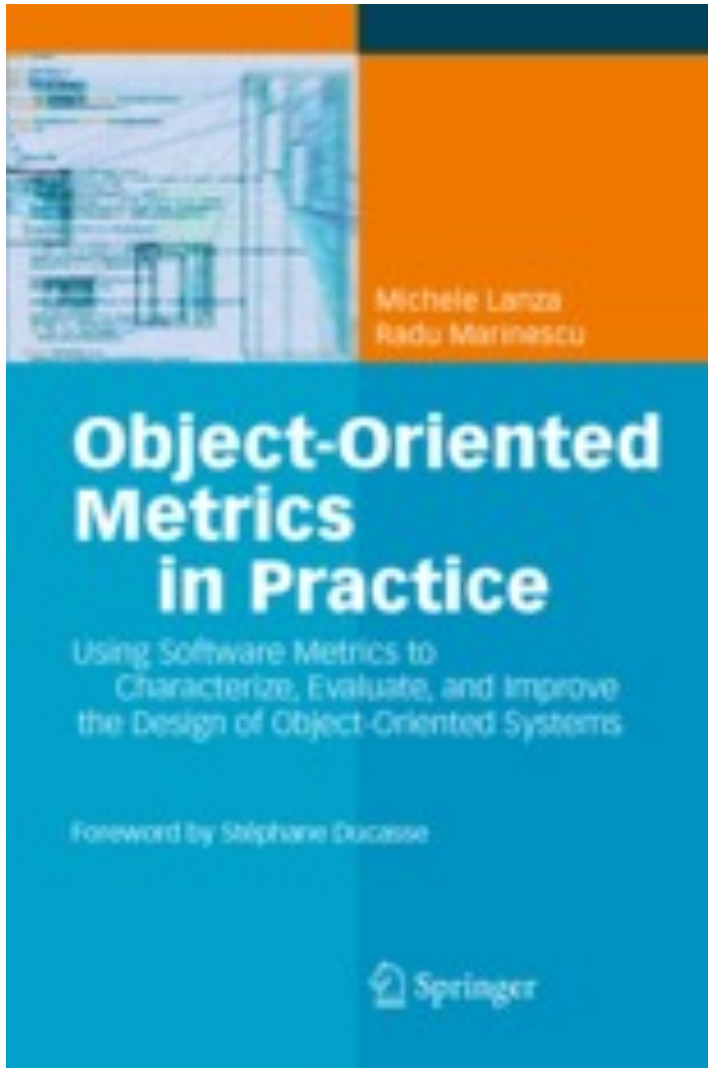
Tools

inCode

- http://loose.upt.ro/incode/pmwiki.php/

More info on Detection Strategies

Object-Oriented Metrics in Practice
Michele Lanza and Radu Marinescu, Springer 2006
http://www.springer.com/computer/swe/book/978-3-540-24429-5


Michele Lanza
Radu Marinescu
Object-Oriented Metrics in Practice
Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems
Foreword by Stéphane Ducasse
Springer

RoadMap

Introduction

Problem detection in the source code

- Code Smells
- Polymetric Views

Problem detection in the evolution

- The Evolution Matrix
- Kiviat Graphs

Conclusion

Understanding Evolution

Changes can point to design problems

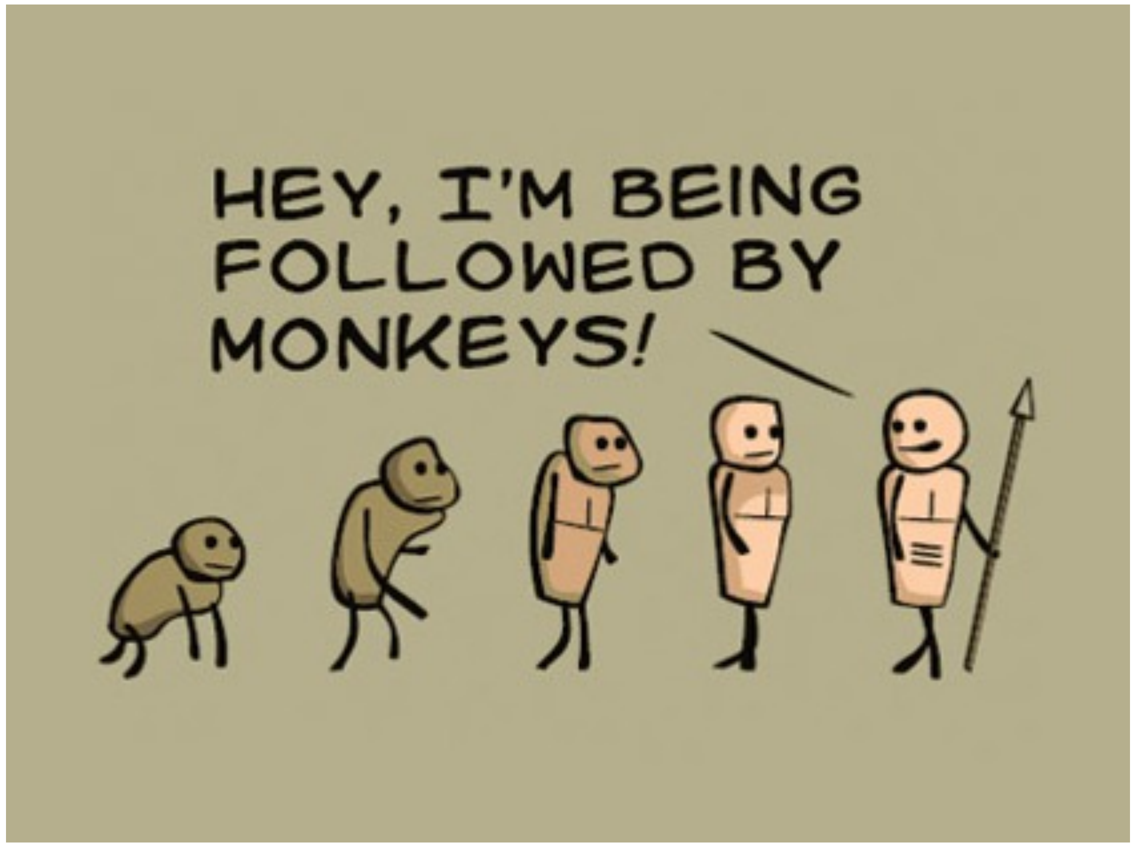
- “Evolutionary Smells”

But

- Overwhelming complexity
- How can we detect and understand changes?

Solutions

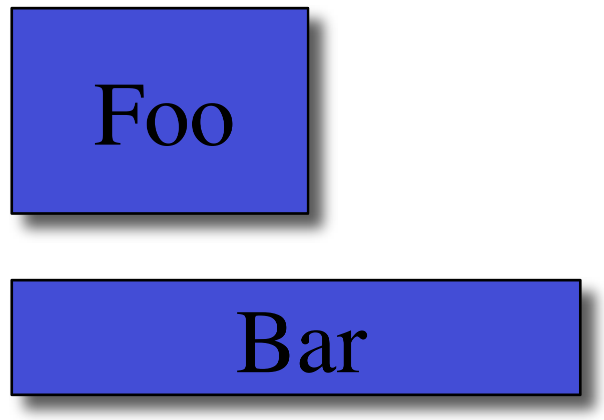
- The Evolution Matrix
- The Kiviat Graphs



Visualizing Class Evolution

Visualize classes as rectangles using for width and height the following metrics:

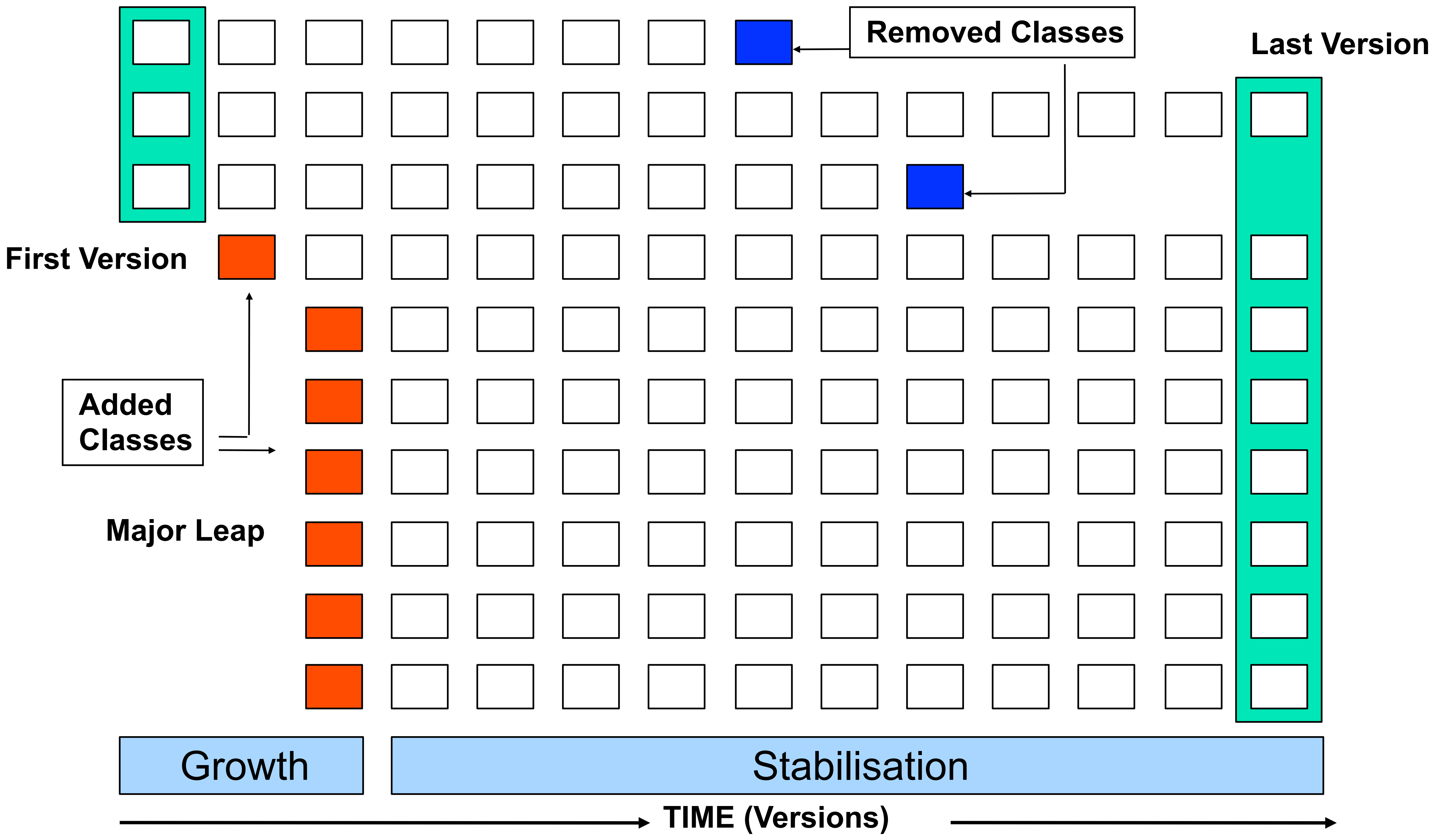
- NOM (number of methods)
- NOA (number of attributes)



The Classes can be categorized according to their "personal evolution" and to their "system evolution"

-> Evolution Patterns

The Evolution Matrix



Evolution Patterns & Smells

Day-fly (Dead Code)

Persistent

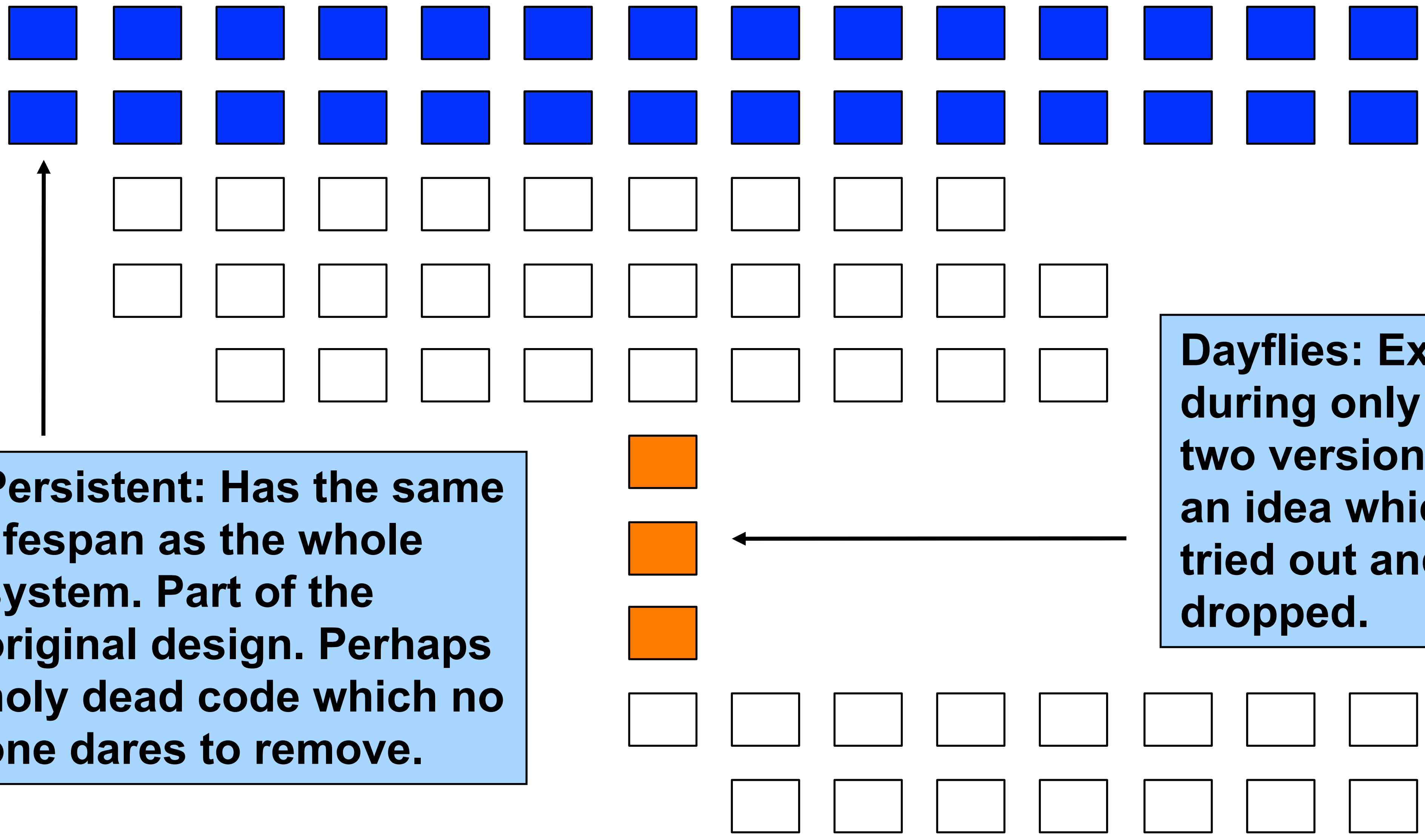
Pulsar (Change Prone Entity)

SupernovaWhite Dwarf (Dead Code)

Red Giant (Large/God Class)

Idle (Dead Code)

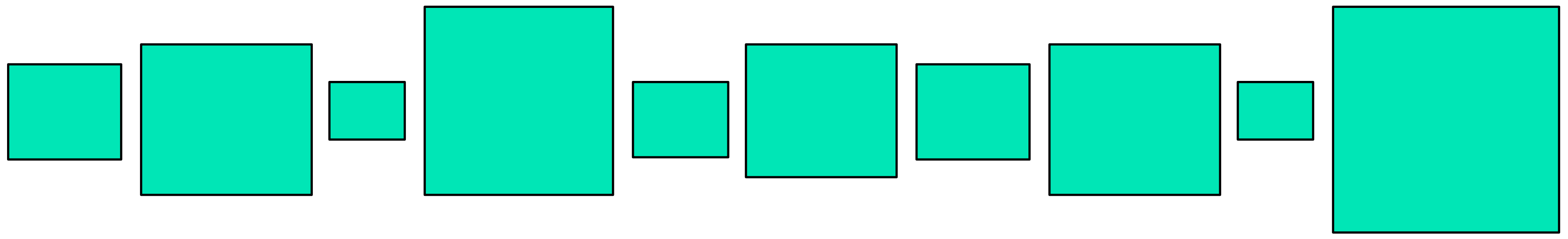
Persistent / Dayfly



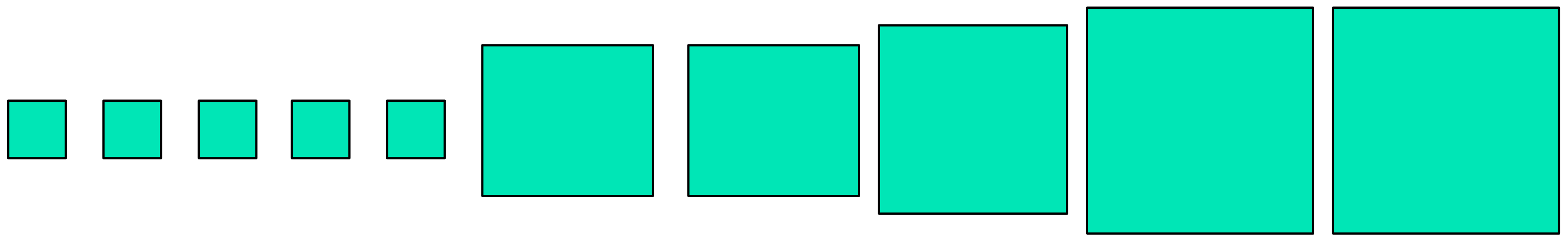
Persistent: Has the same lifespan as the whole system. Part of the original design. Perhaps holy dead code which no one dares to remove.

Dayflies: Exists during only one or two versions. Perhaps an idea which was tried out and then dropped.

Pulsar / Supernova

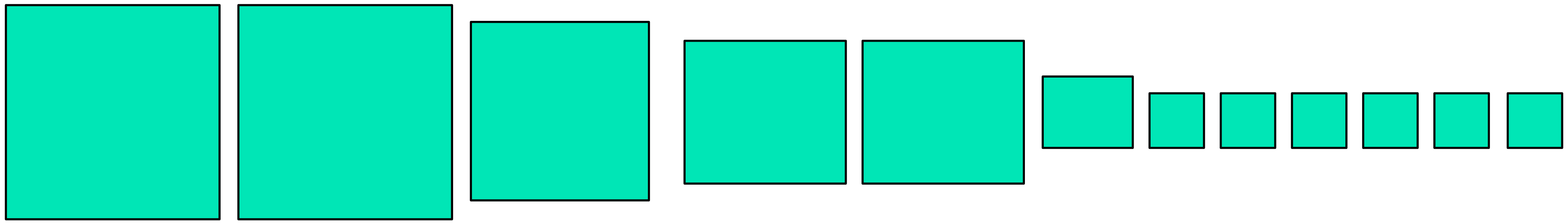


Pulsar: Repeated Modifications make it grow and shrink.
System Hotspot: Every System Version requires changes.

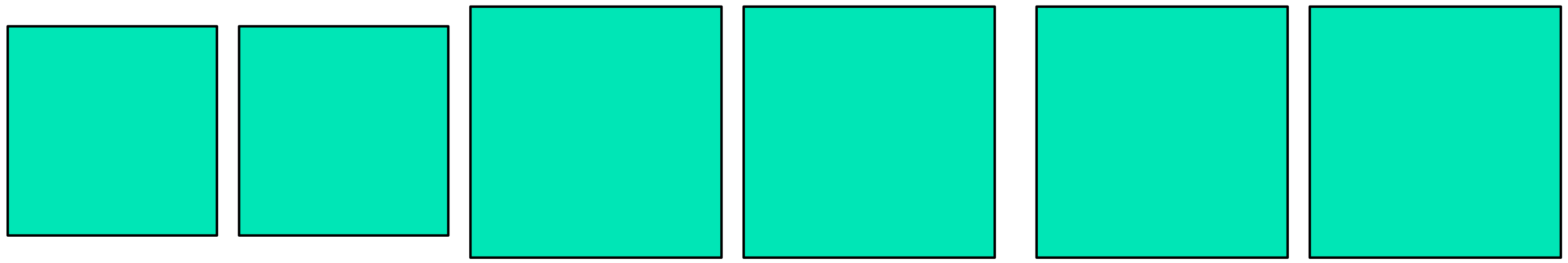


Supernova: Sudden increase in size. Possible Reasons:
- Massive shift of functionality towards a class.
- Data holder class for which it is easy to grow.
- *Sleeper*: Developers knew exactly what to fill in.

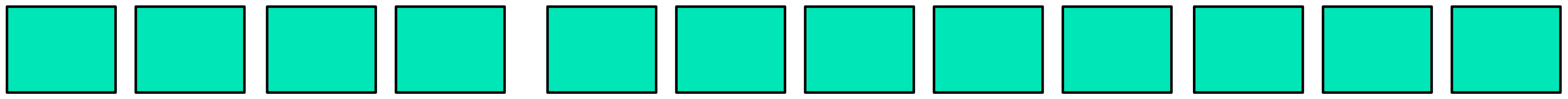
White Dwarf / Red Giant / Idle



White Dwarf: Lost the functionality it had and now trundles along without real meaning. Possibly dead code -> Lazy Class.

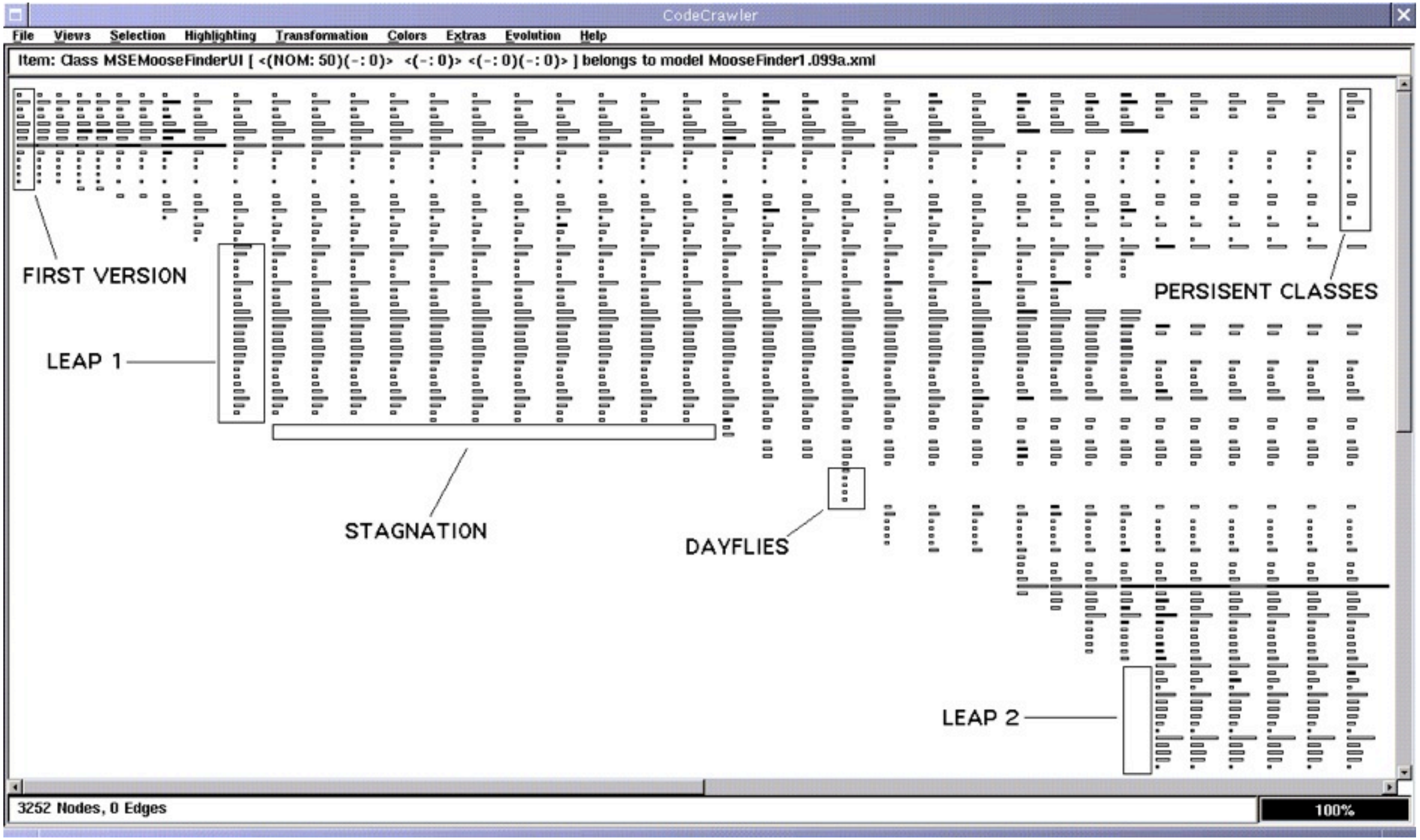


Red Giant: A permanent god (large) class which is always very large.



Idle: Keeps size over several versions. Possibly dead code, possibly good code.

Real Example: MooseFinder


CodeCrawler
File Views Selection Highlighting Transformation Colors Extras Evolution Help
FIRST VERSION
LEAP 1
STAGNATION
DAYFLIES
PERSISENT CLASSES
LEAP 2
100%

Item: Class MSEMooseFinderUI [<(NOM: 50)(-: 0)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model MooseFinder1.099a.xml

3252 Nodes, 0 Edges

Evaluation: Evolution Matrix

Pros

- Understand the evolution of a system in terms of size and growth rate
- Introduction of new classes
- Remove of classes
- Detection of Evolution Patterns & Smells
  - Dayflight, Persistent, White Dwarf, ...

Cons

- Scalability
- Limited to 3 metric values per glyph
- Fragile regarding the renaming of classes
  - What if the name of a class was changed?

Extended Polymetric Views

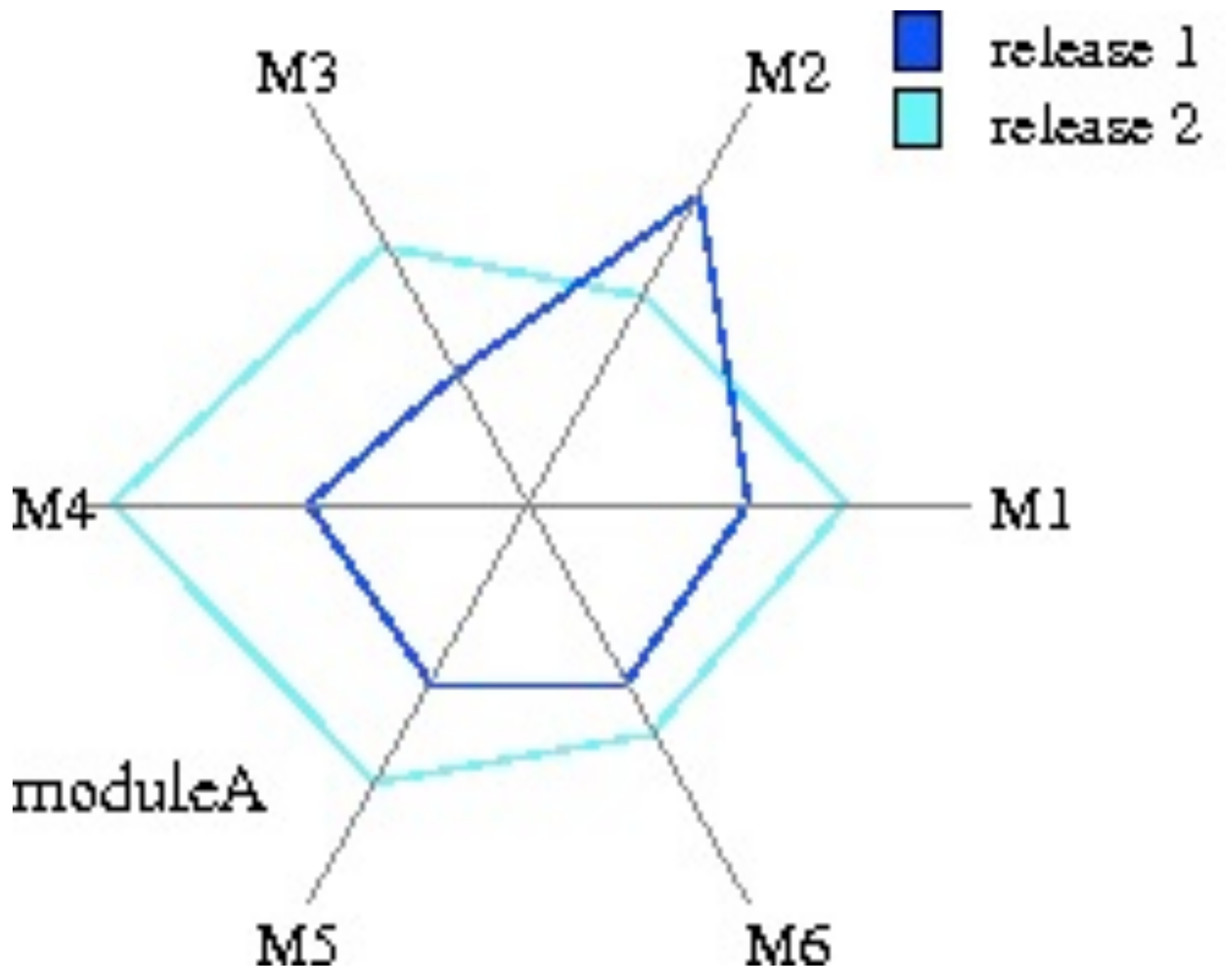
Goal:

- Visualize n metric values of m releases
  - More semantic in graphs
  - More flexibility to combine metric values

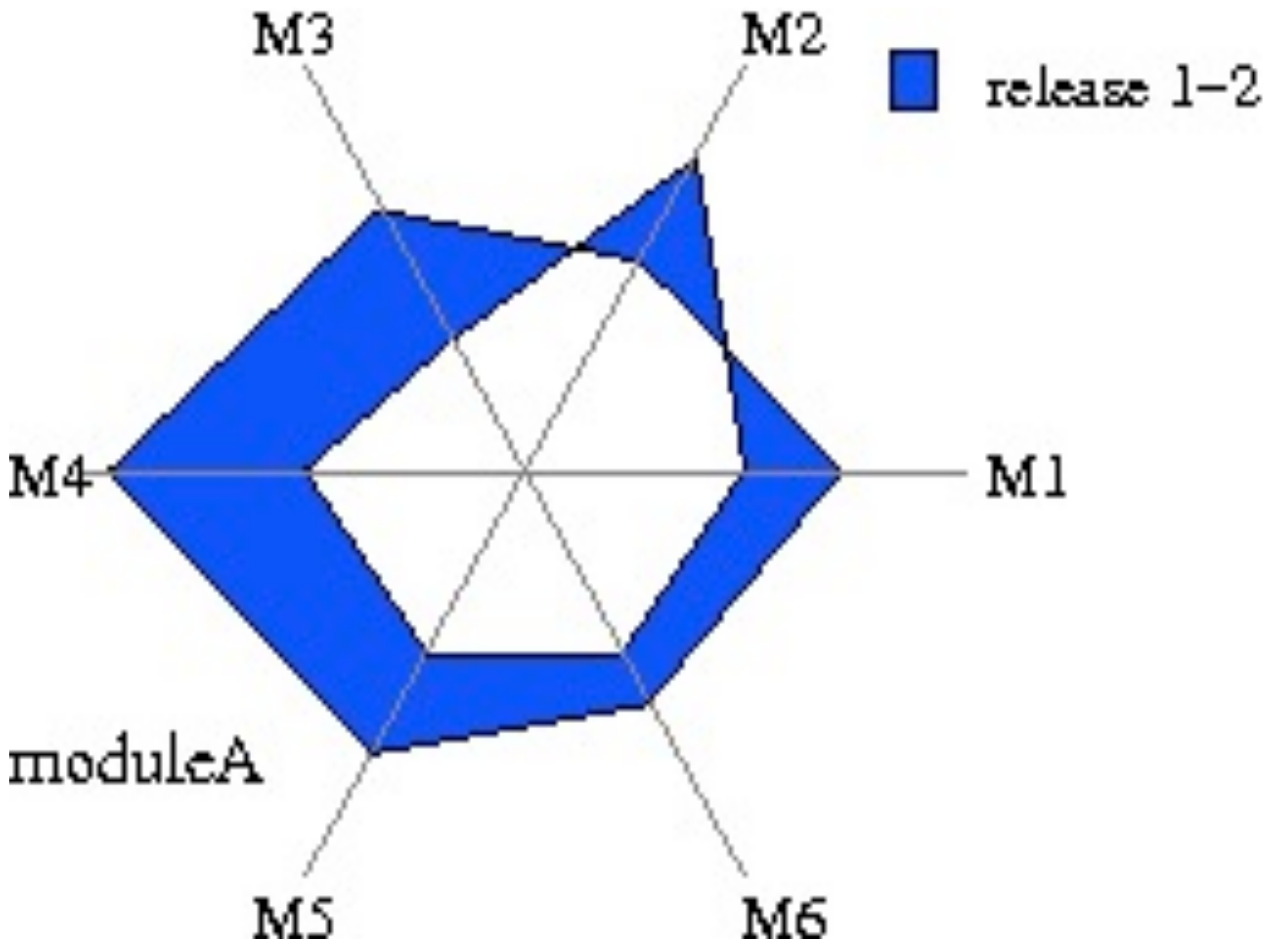
Solution: Kiviat Diagrams (Radar Charts)

- Each ray represents a metric
- Encode releases with different colors

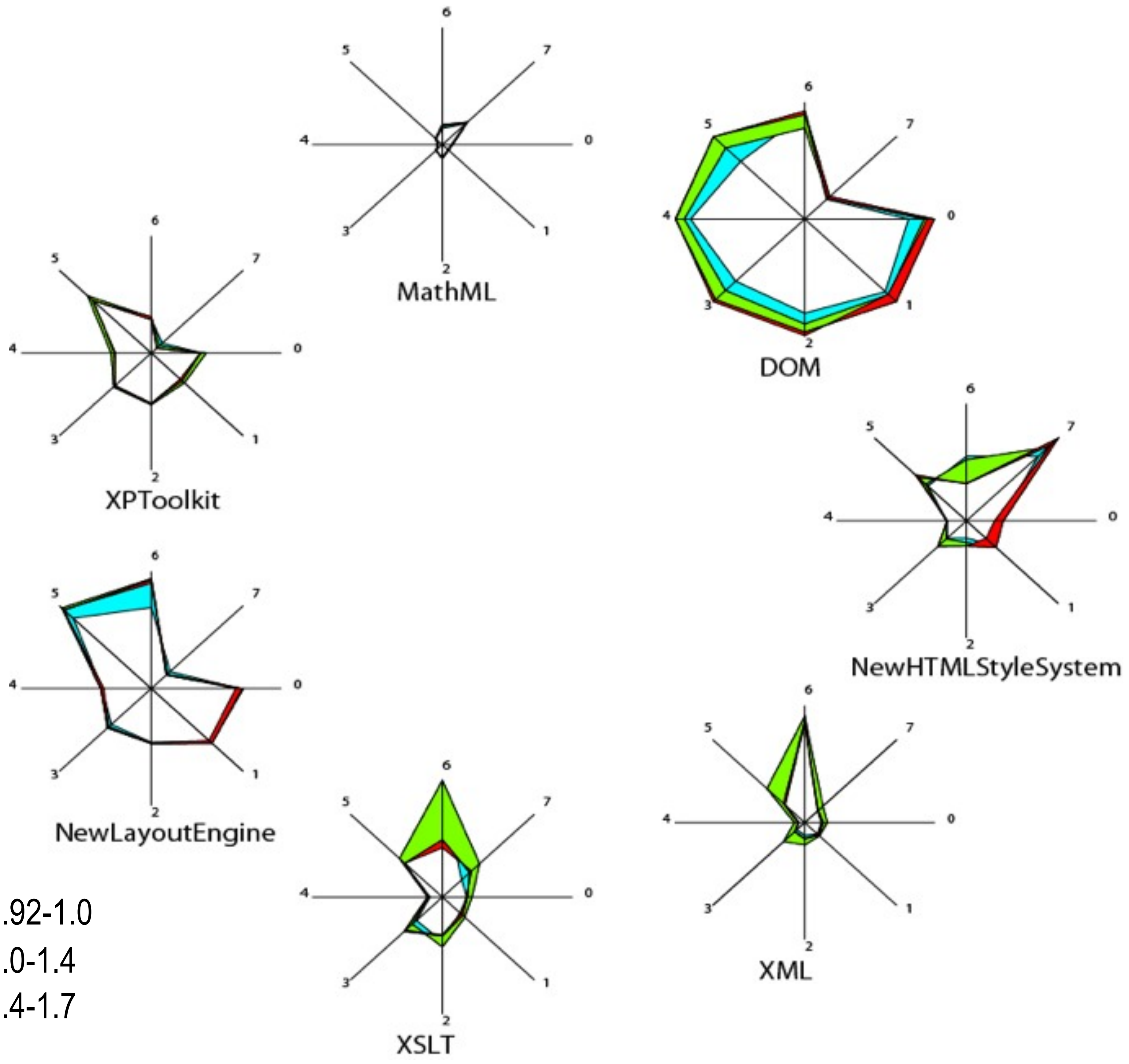
Kiviat Diagram



Highlight the Change



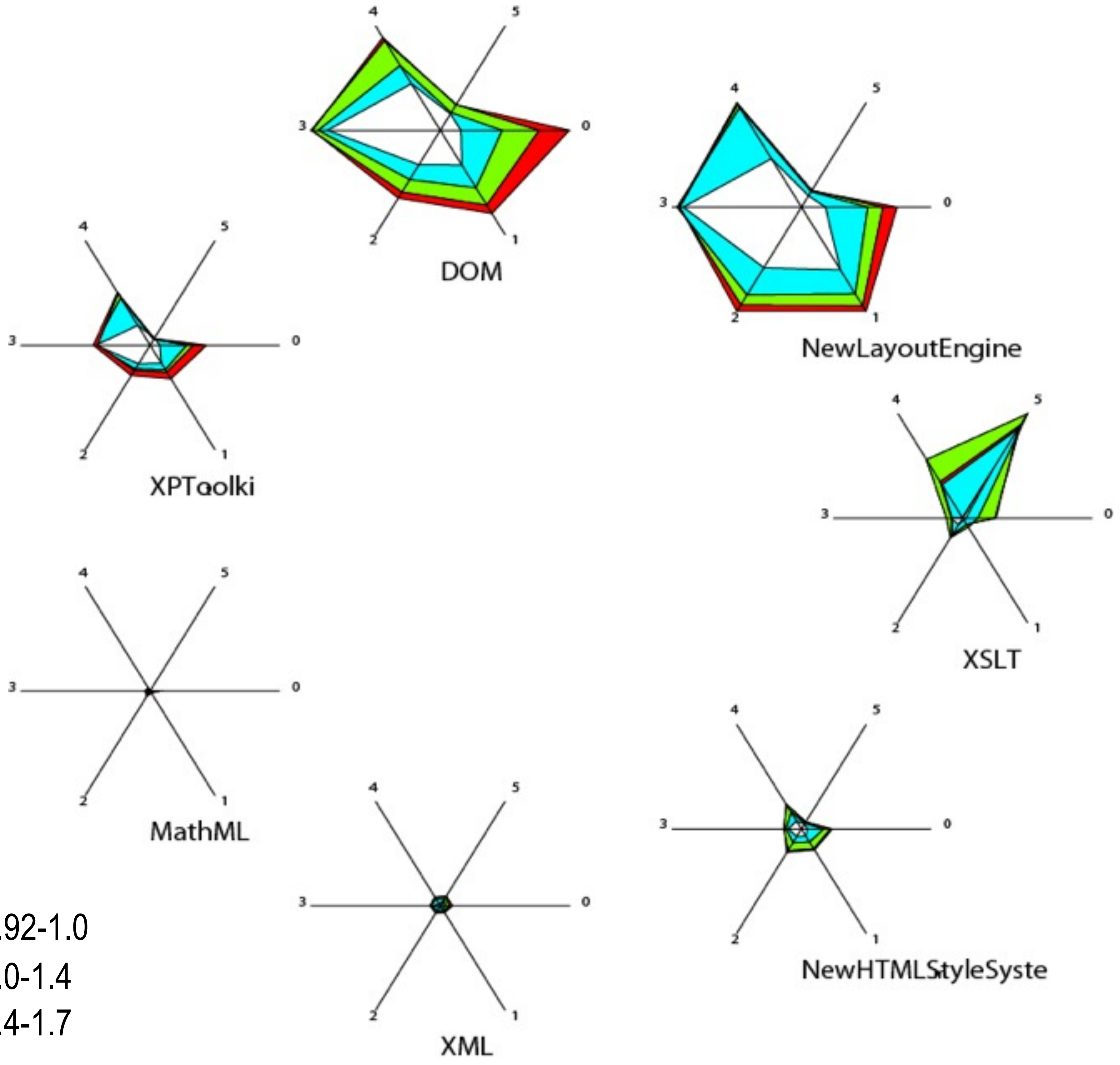
Size & Complexity Metrics



🟦 release 0.92-1.0
🟩 release 1.0-1.4
🟥 release 1.4-1.7

Metrics:
0:nrStmts
1:CCMPLX
2:nrFiles
3:nrClasses
4:nrMeths
5:nrAttrs
6:nrGlobFuncs
7:nrGlobVars

Problem Report Metrics



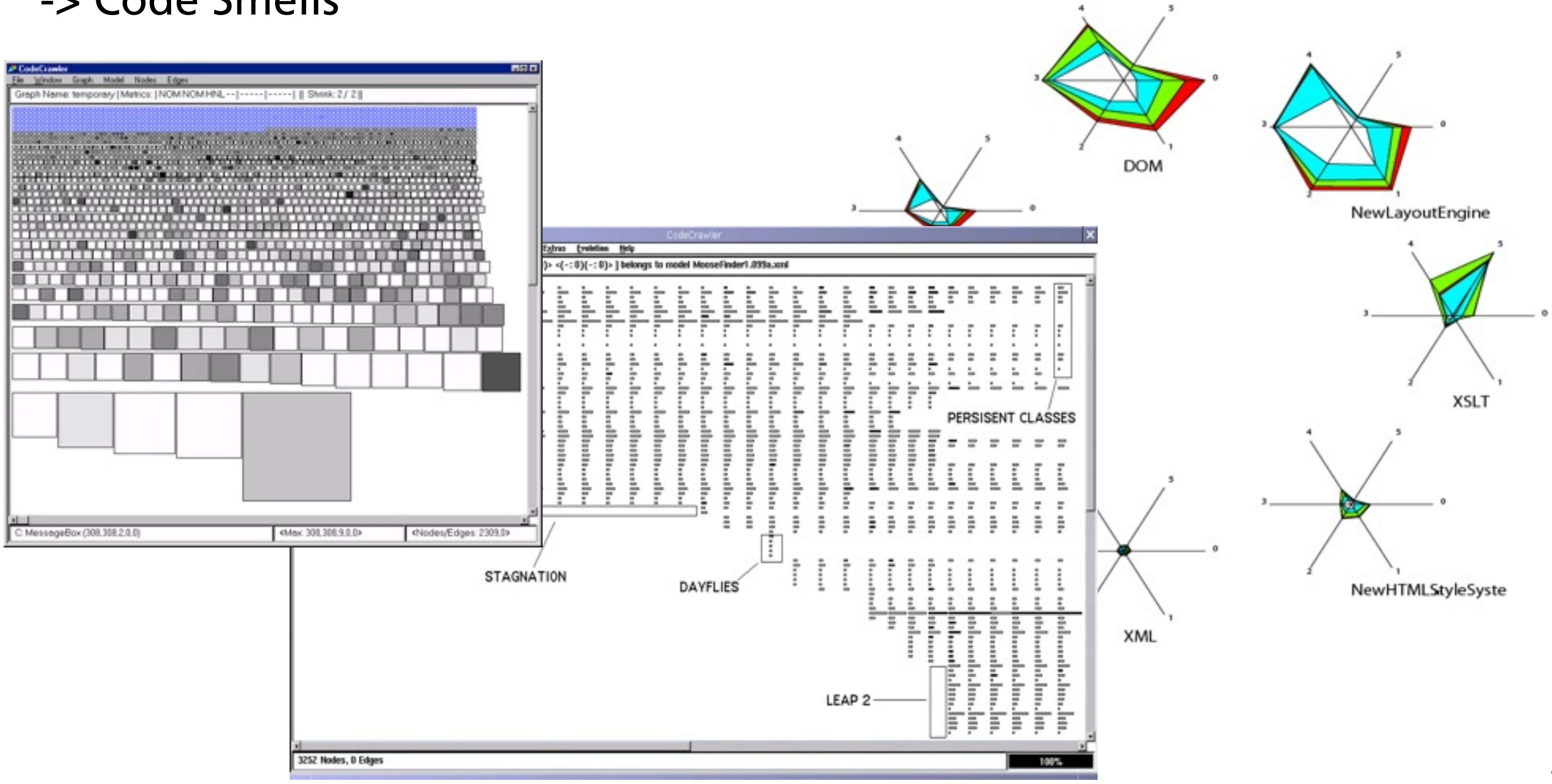
🟦 release 0.92-1.0
🟩 release 1.0-1.4
🟥 release 1.4-1.7

Metrics:
0:nrPrio_undef
1:nrPrio_1
2:nrPrio_2
3:nrPrio_3
4:nrPrio_4
5:nrPrio_5

Conclusions

Design Problems

Result from duplicated, unclear, complicated source code
-> Code Smells


DOM
NewLayoutEngine
XSLT
NewHTMLStyleSyste
XML
PERSISTENT CLASSES
STAGNATION
DAYFLIES
LEAP 2
CodeCrawler
3252 Nodes, 0 Edges