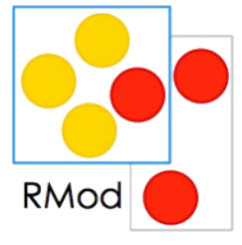


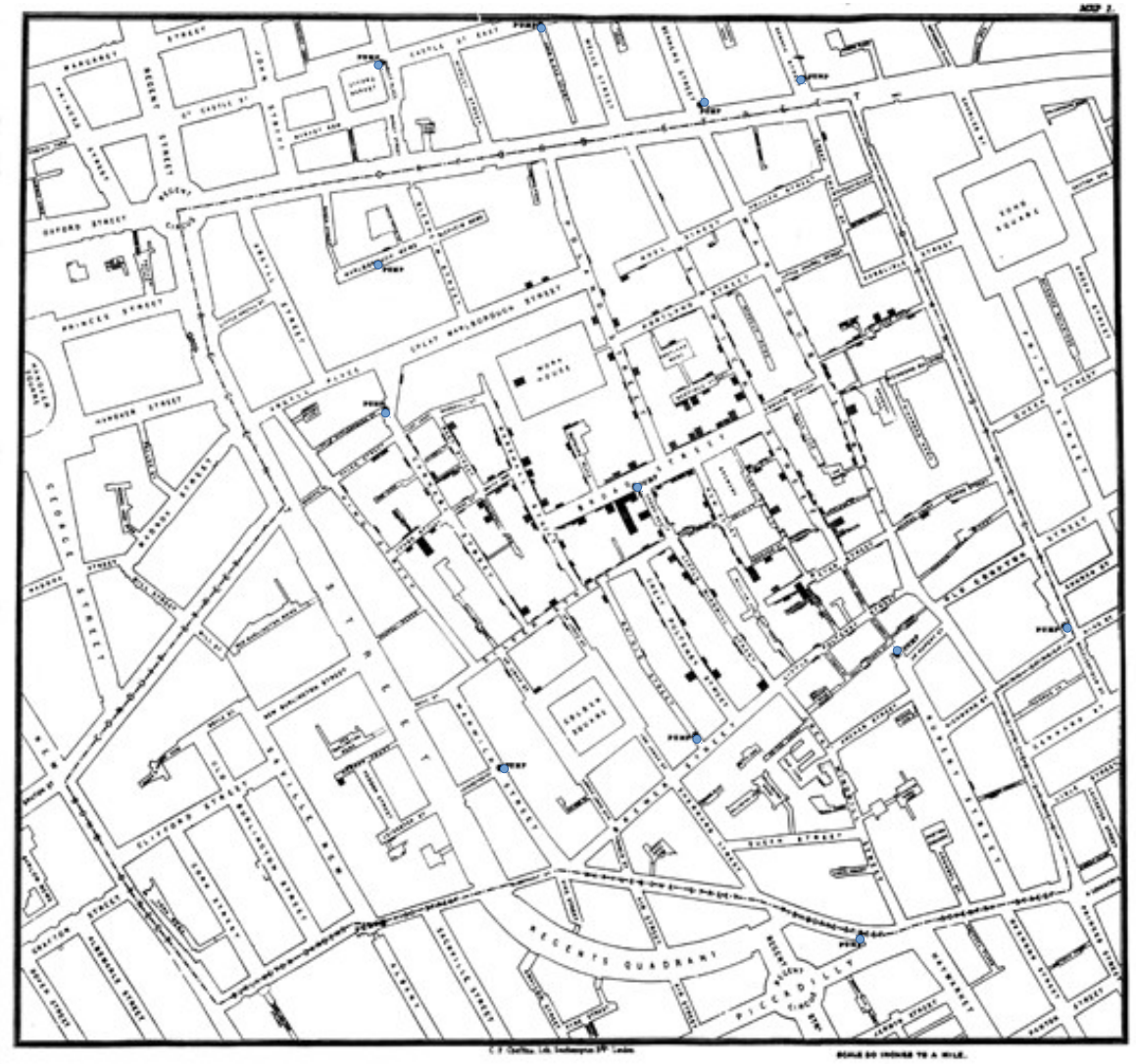

# Software Visualisation

**Nicolas Anquetil**

nicolas.anquetil@inria.fr



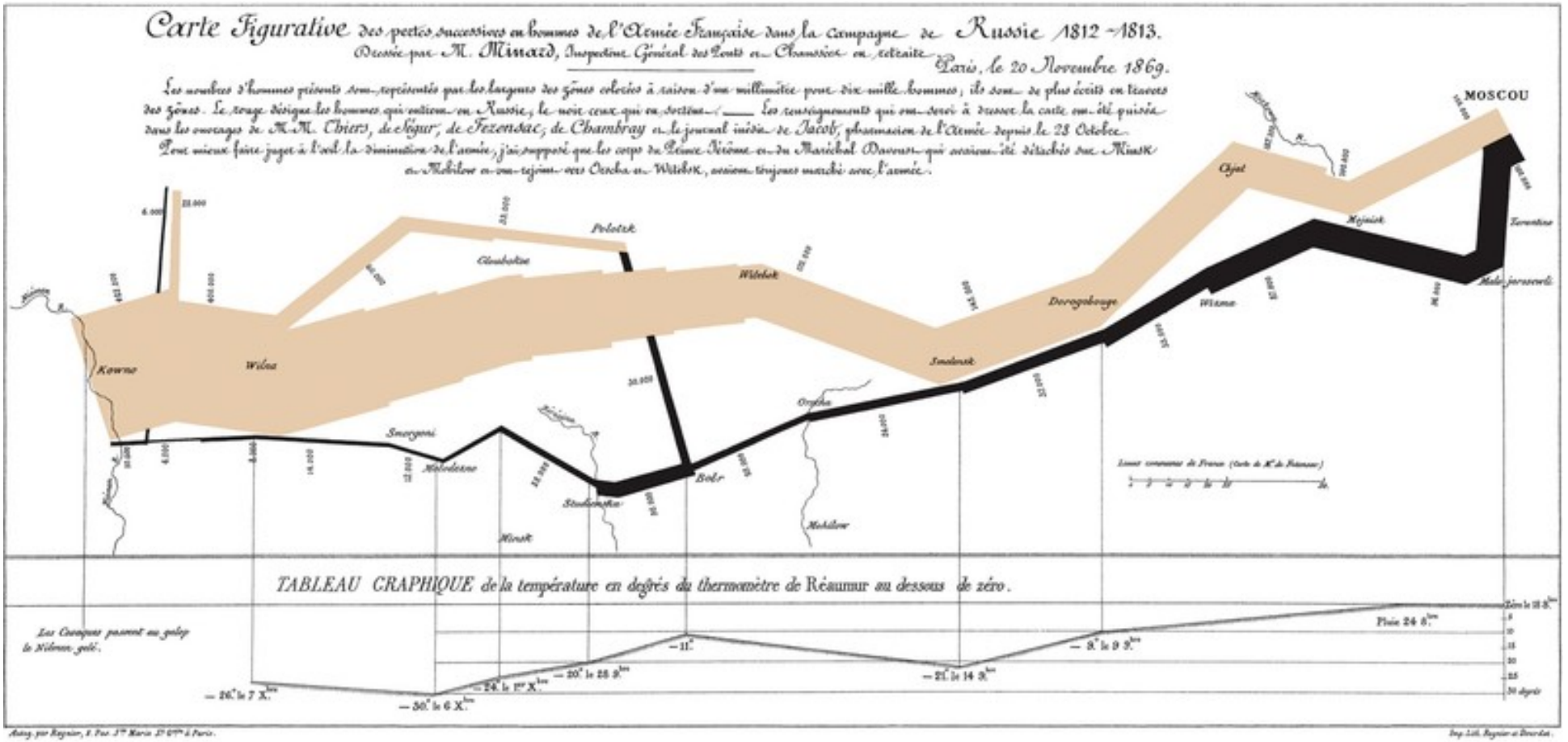
# On the Use of Visualization




Dr. Snow's map of the 1854
London cholera outbreak

RMod

# On the Use of Visualization

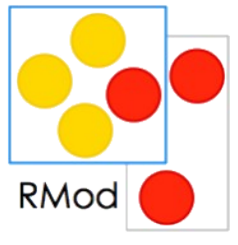



Napoleonian army in the 1812 russian campaign

RMod

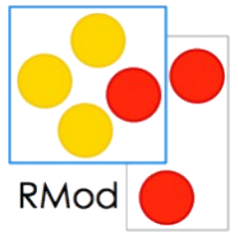
# On the Use of Visualization

- Reduction of complexity
- Generate different views on software system
- Visualization is powerful. But
  - Can be complex (active research area),
  - Efficient space use, crossing edges, focus...
  - Colors are nice but there is no convention
  - Nice pictures do not imply valuable information
  - Where to look? What is important?



# Information Visualization

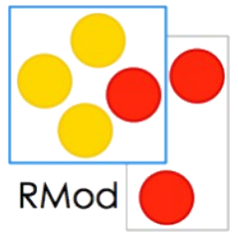
- Bertin [1983] assessed three levels of questions
  - Lower perception (one element)
  - Medium perception (several elements)
  - Upper perception (all elements/the complete picture)
- In Information Visualization it’s all about the reduction of complexity
  - What to visualize? (focus on what matters)
  - How to visualize? (graphical artefacts with meaning)



# Software Visualization

"*Software Visualization is the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software*."

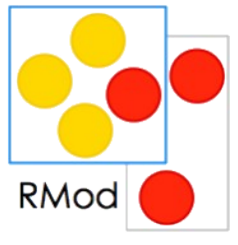
Price, Baecker and Small,
"Introduction to Software Visualization"



# The main conceptual problem

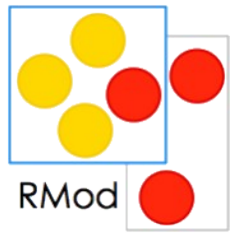
*“Software is intangible, having no physical shape or size. Software visualisation tools use graphical techniques to make software visible by displaying programs, program artifacts and program behaviour.”*

Thomas Ball

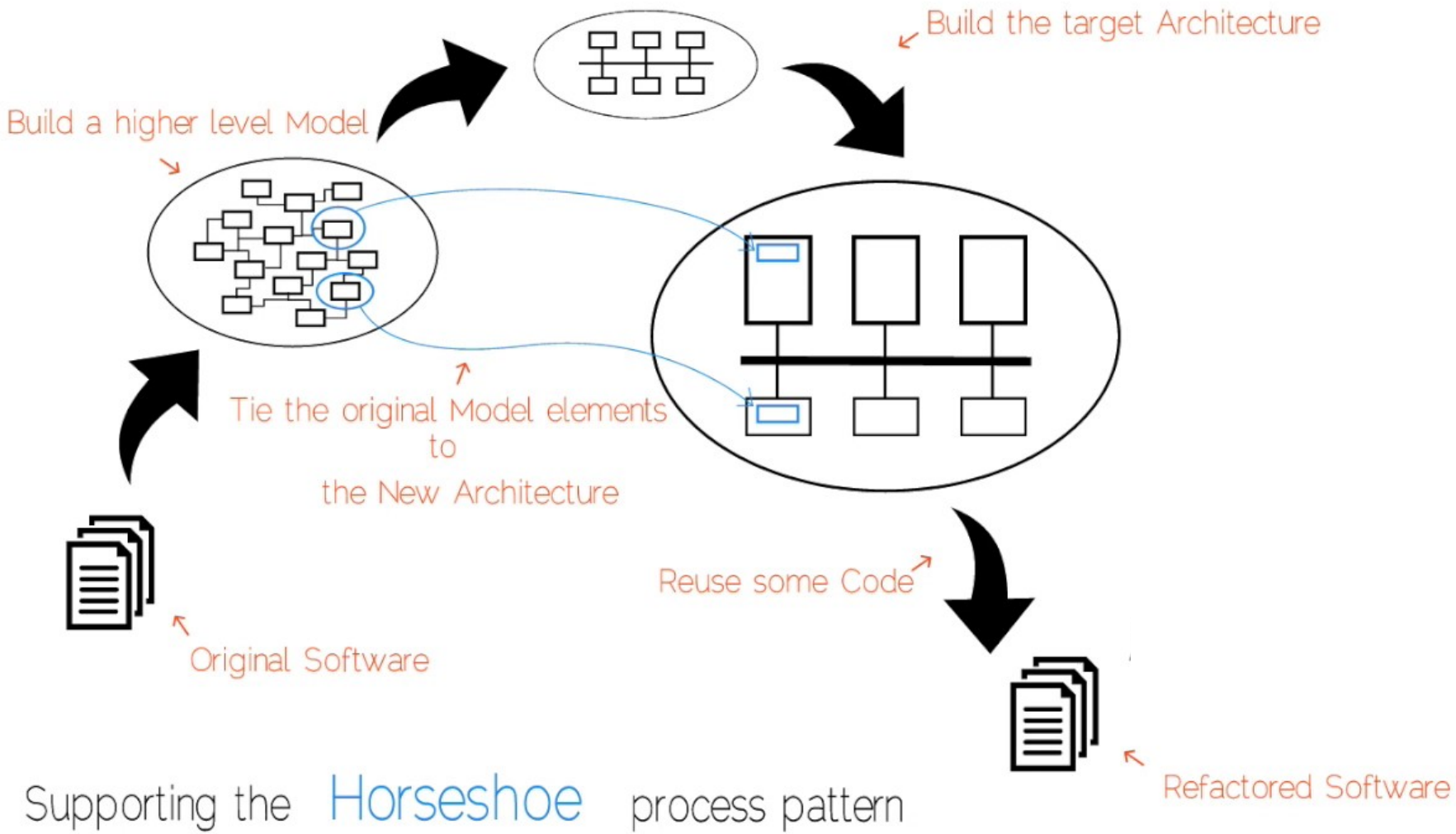


# Reengineering context

- Limited time, Limited resources
- Work on old systems, dialects (new tools are not processing them)
- Approaches
  - Scalability is crucial
  - Efficient (time/information obtained)
  - Need a clear focus (Visualization in support of a process)



# Reengineering context




Supporting the Horseshoe process pattern

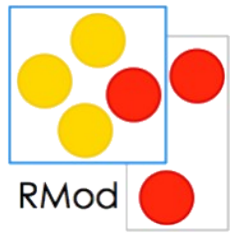
RMod

# Program Visualization

*“Program visualization is the visualization of the actual program code or data structures in either static or dynamic form”*

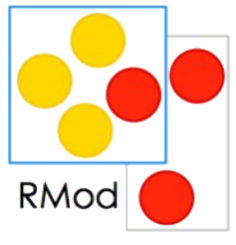
[Price, Baecker and Small]

- Static/Dynamic code visualization
- Generate different views of a system and infer knowledge based on the views
- Complex problem domain
  - Efficient space use, edge crossing problem, layout problem, focus, HCI issues, GUI issues, …
  - Lack of conventions (colors, symbols, interpretation, …)



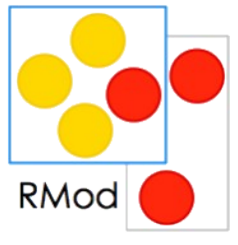
# Program Visualization

- Level of granularity?
  - Complete systems, subsystems, modules, classes, hierarchies,...
- When to apply?
  - First contact with a unknown system
  - Known/unknown parts?
  - Forward engineering?
- Methodology?



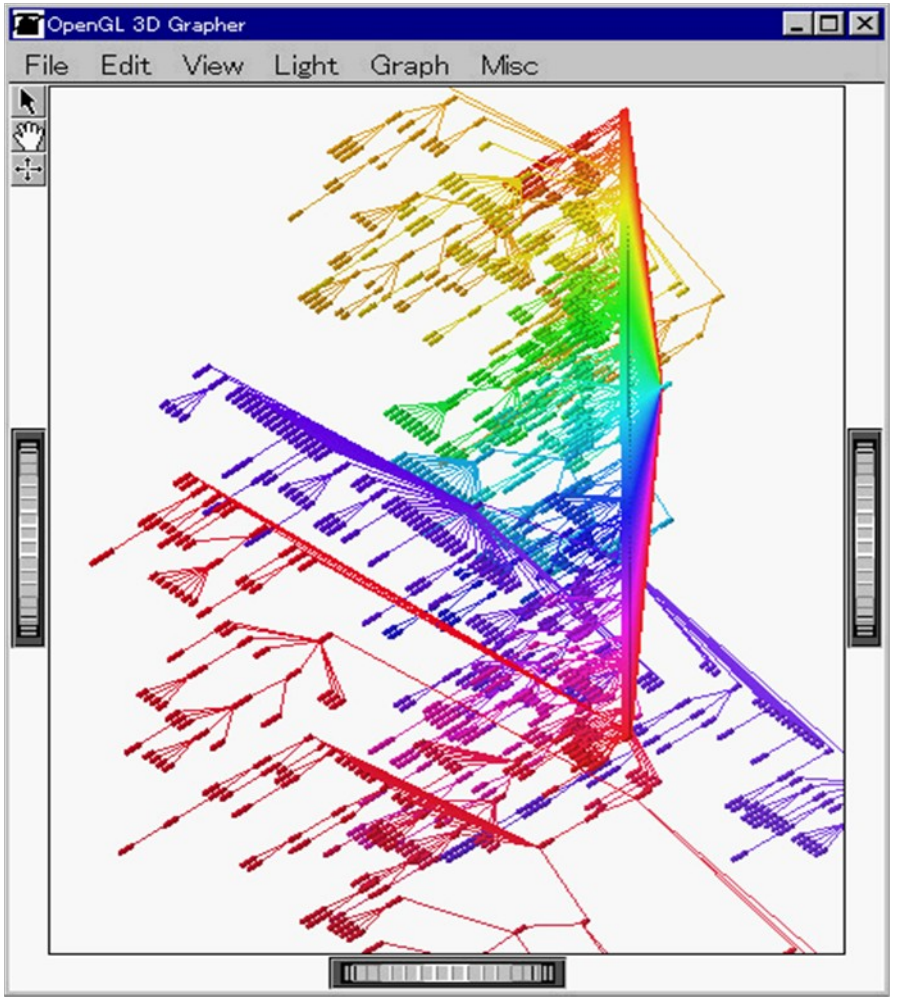
# Static Code Visualization

- The visualization of information that can be extracted from the static structure of a software system
- Depends on the programming language and paradigm:
  - Object-Oriented PL:
    - classes, methods, attributes, inheritance, …
  - Procedural PL:
    - procedures, invocations, …



# Example 1: Class Hierarchies

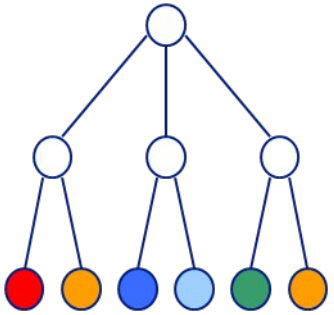
- The Smalltalk Class Hierarchy
- Problems:
  - Colors are meaningless
  - Visual Overload

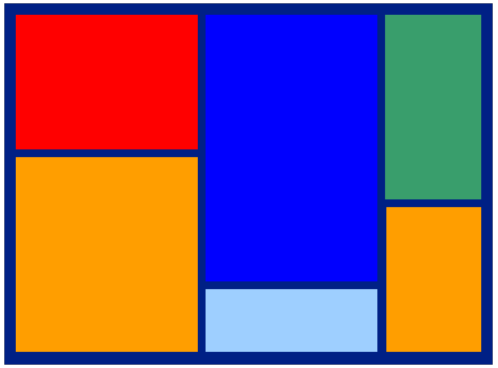


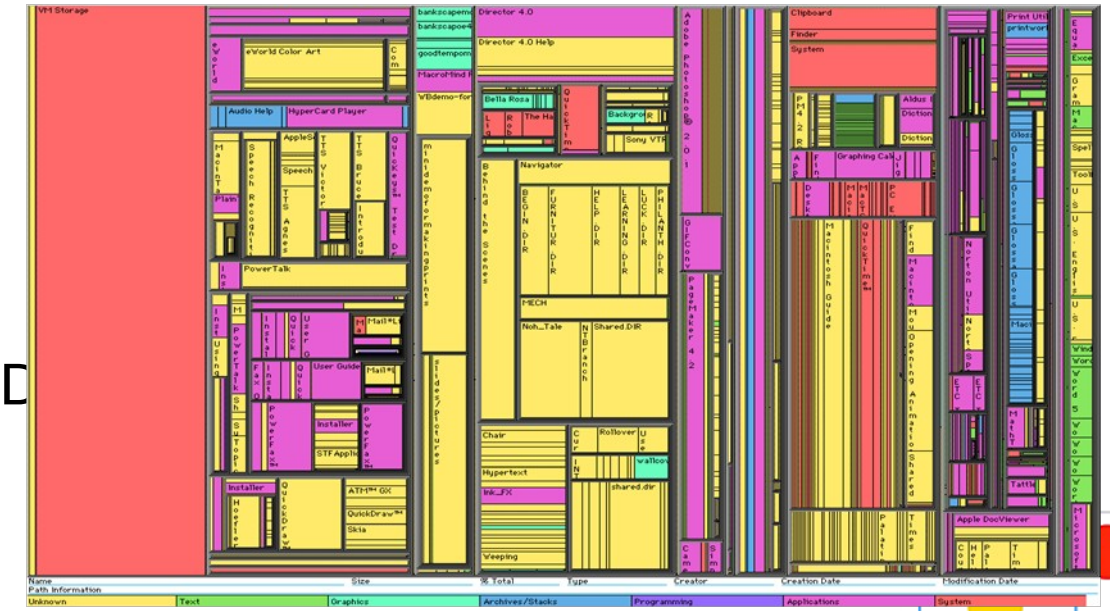

RMod

# Example 2: Tree Maps

- Pros
  - 100% screen
  - Large data
  - Scales well
- Cons
  - Boundaries
  - Cluttered display
  - Interpretation
  - Leaves only
  - Useful for the display of HDD

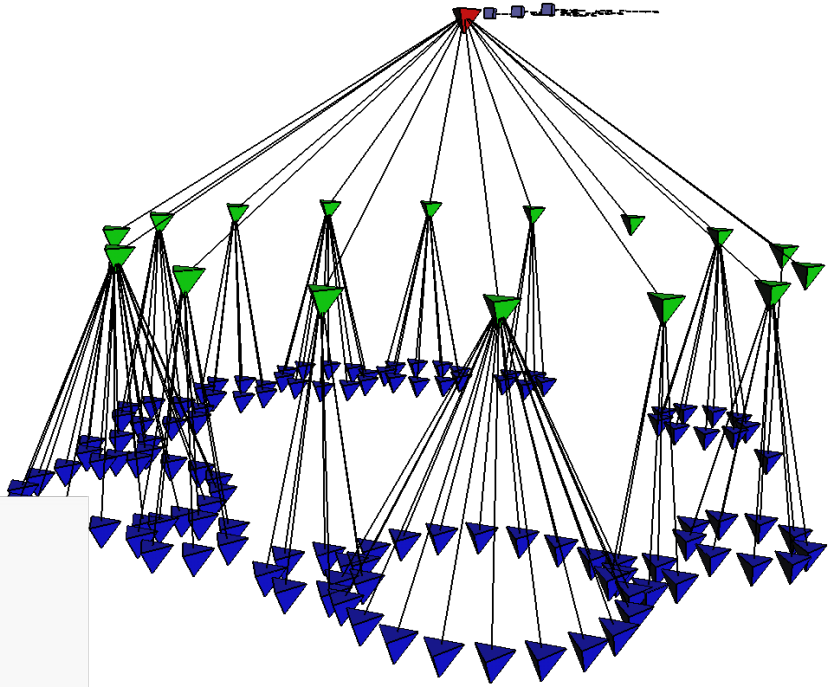


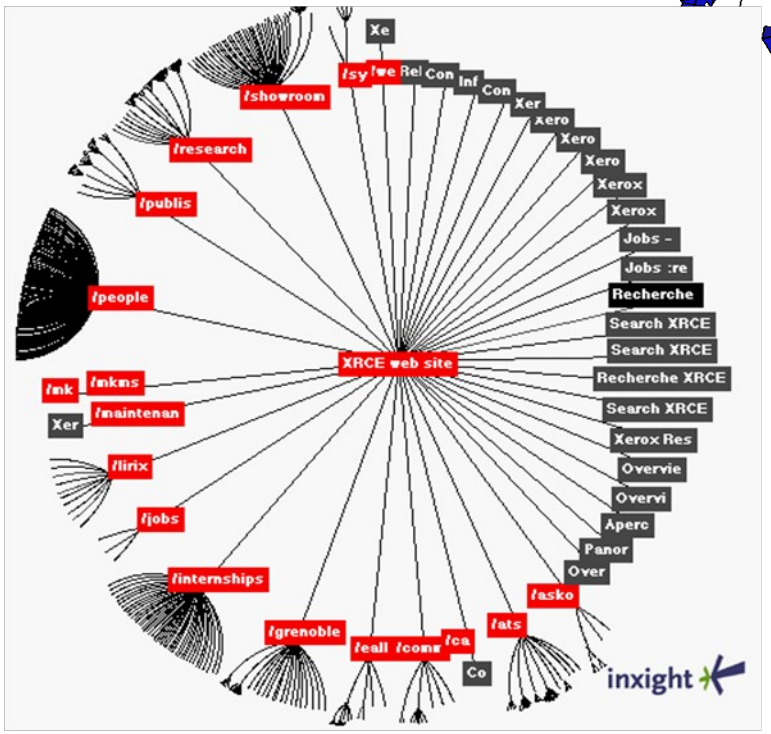




# Examples 3 & 4

- Euclidean cones
- Pros:
  - More info than 2D
- Cons:
  - Needs 3D or animation



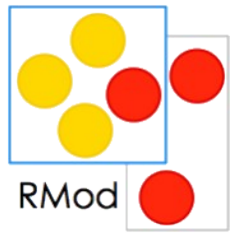



- Hyperbolic trees
- Pros:
  - Good focus
  - Dynamic
- Cons:
  - Copyright? (Xerox)

RMod

# Code Maps

- One “Dot” = one line of code
- Simple
- Overview
- File-based

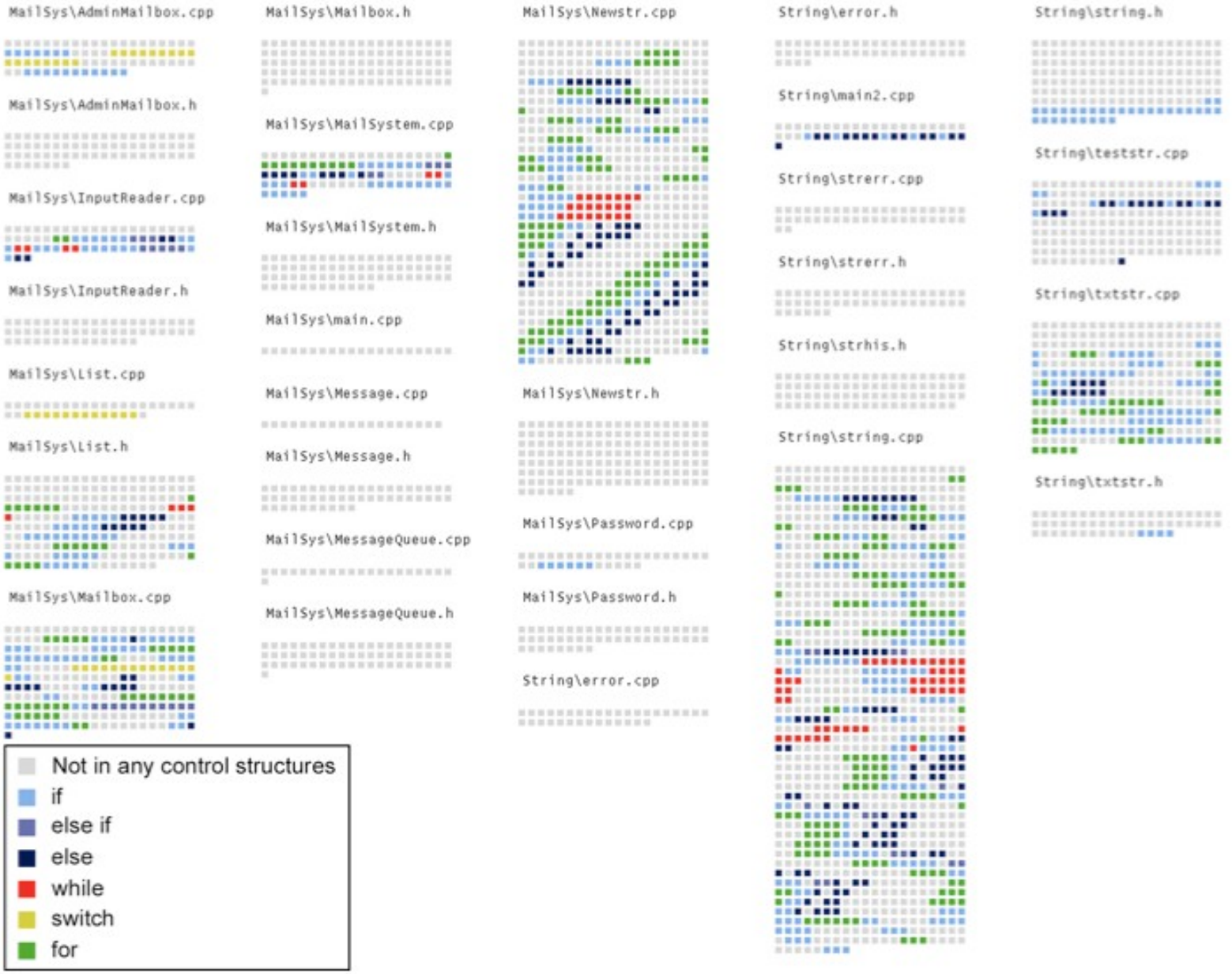


# Code Maps

- Nesting levels

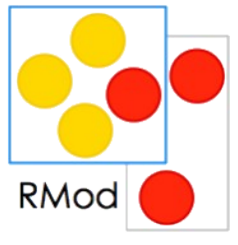
# Code Maps

- Control flows



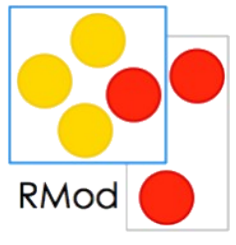
# Code Maps

- Pros
  - Simple to draw
  - Good overview
- Cons
  - Limited semantics
  - Patterns difficult to identify because of line breaks

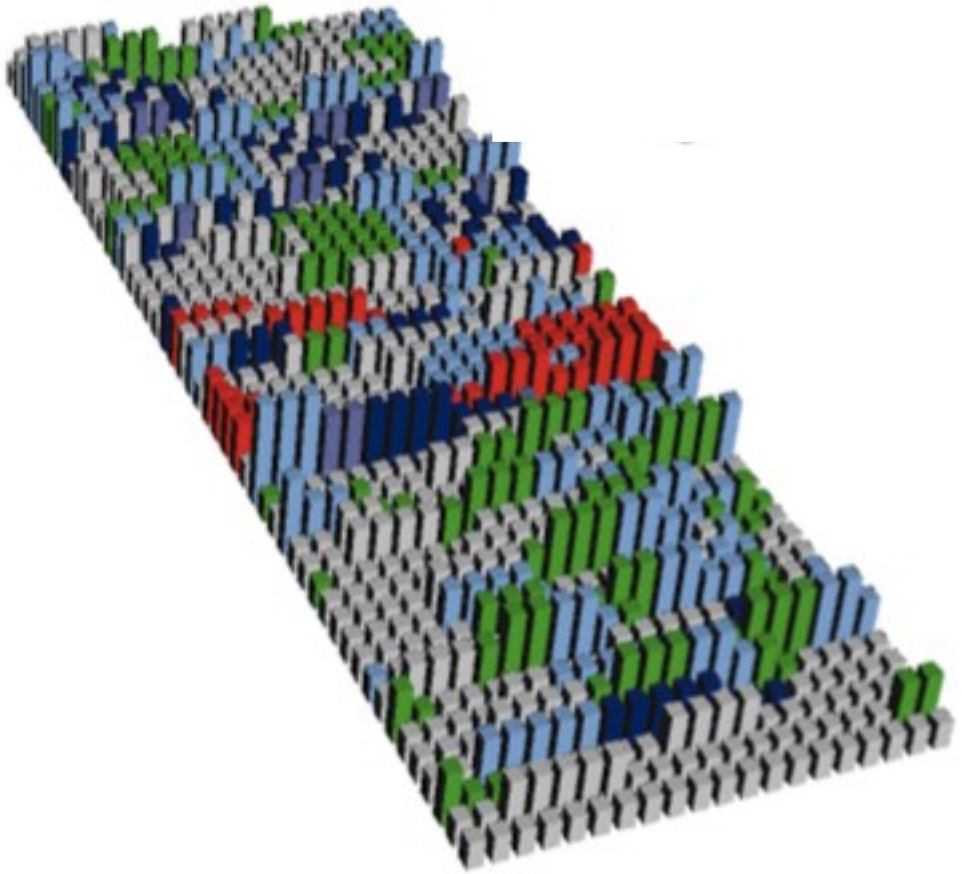


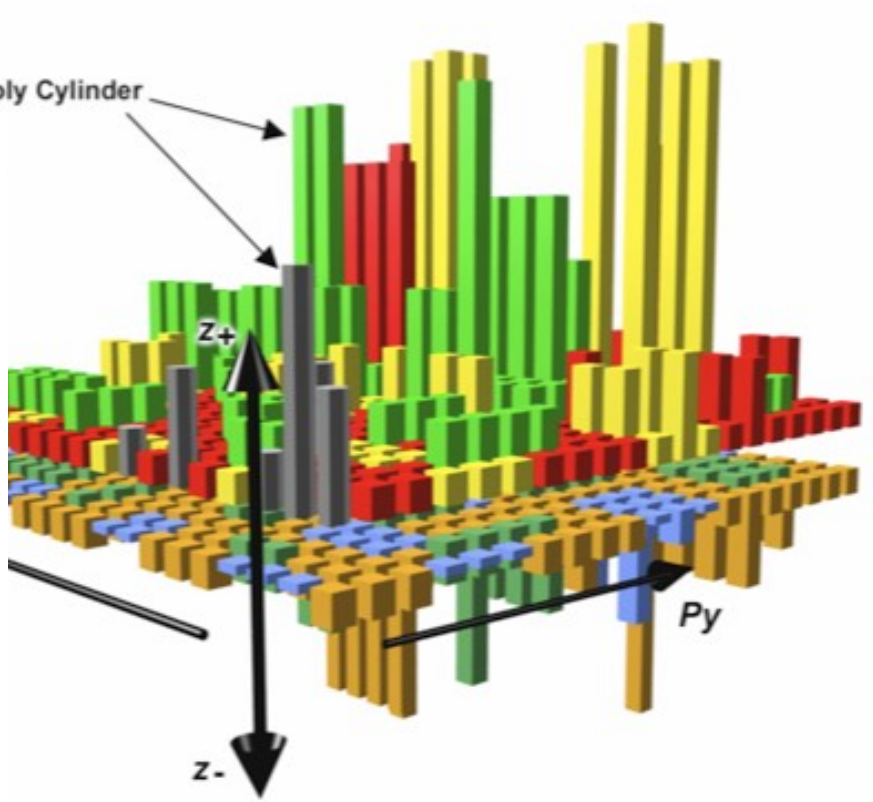
# Class Diagram Approaches

- For example UML diagrams…
- Pros:
  - OO Concepts
  - Good for small parts
- Cons:
  - Lack of scalability
  - Require tool support
  - Preconceived views



# What about 3D?

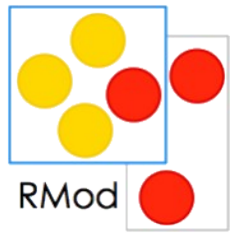





RMod

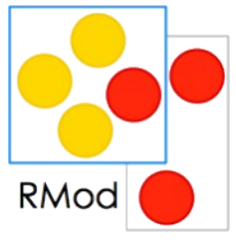
# What about 3D?

- Pros
  - Eye catching
  - One more dimension(?)
- Cons
  - No spatial semantics (is above better than below?)
  - Scalability
  - Extra effort
  - Space localization
  - Rendering in 2D



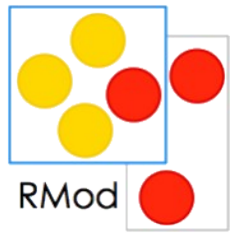
# Static Code Visualization

- General Pros
  - Intuitive approaches
  - Aesthetically pleasing results
- General Cons
  - Several approaches are orthogonal to each other
  - Too easy to produce meaningless results
  - Scaling up is sometimes possible, however at the expense of semantics



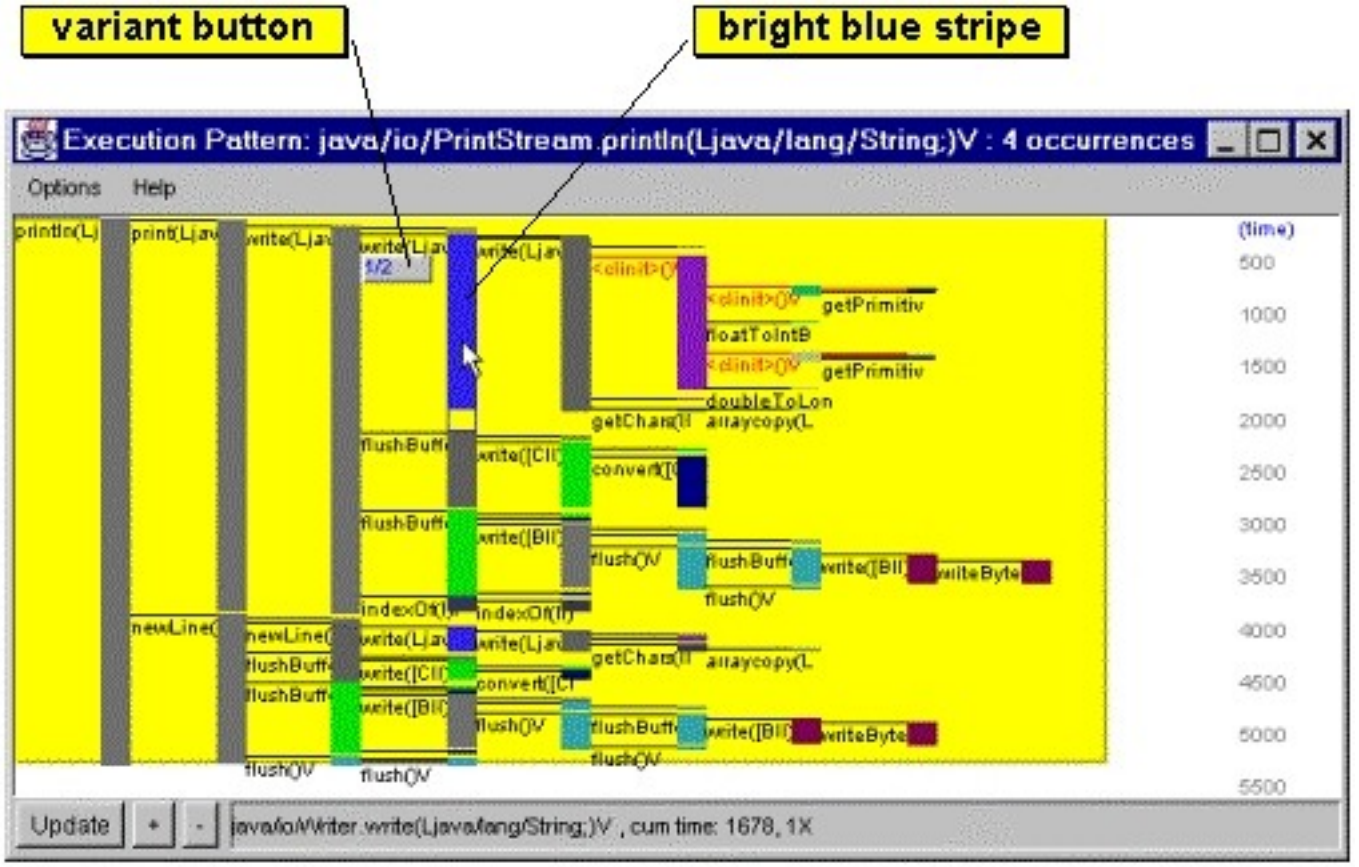
# Dynamic Code Visualization

- Visualization of dynamic behaviour of a software system
  - Code instrumentation
  - Trace collection
  - Trace evaluation
  - What to visualize?
    - Execution trace
    - Memory consumption
    - Object interaction
    - …



# Example 1: JInsight

- Visualization of execution trace




RMod

# Example 2: Inter-class call matrix

- Simple
- Reproducible
- Scales well

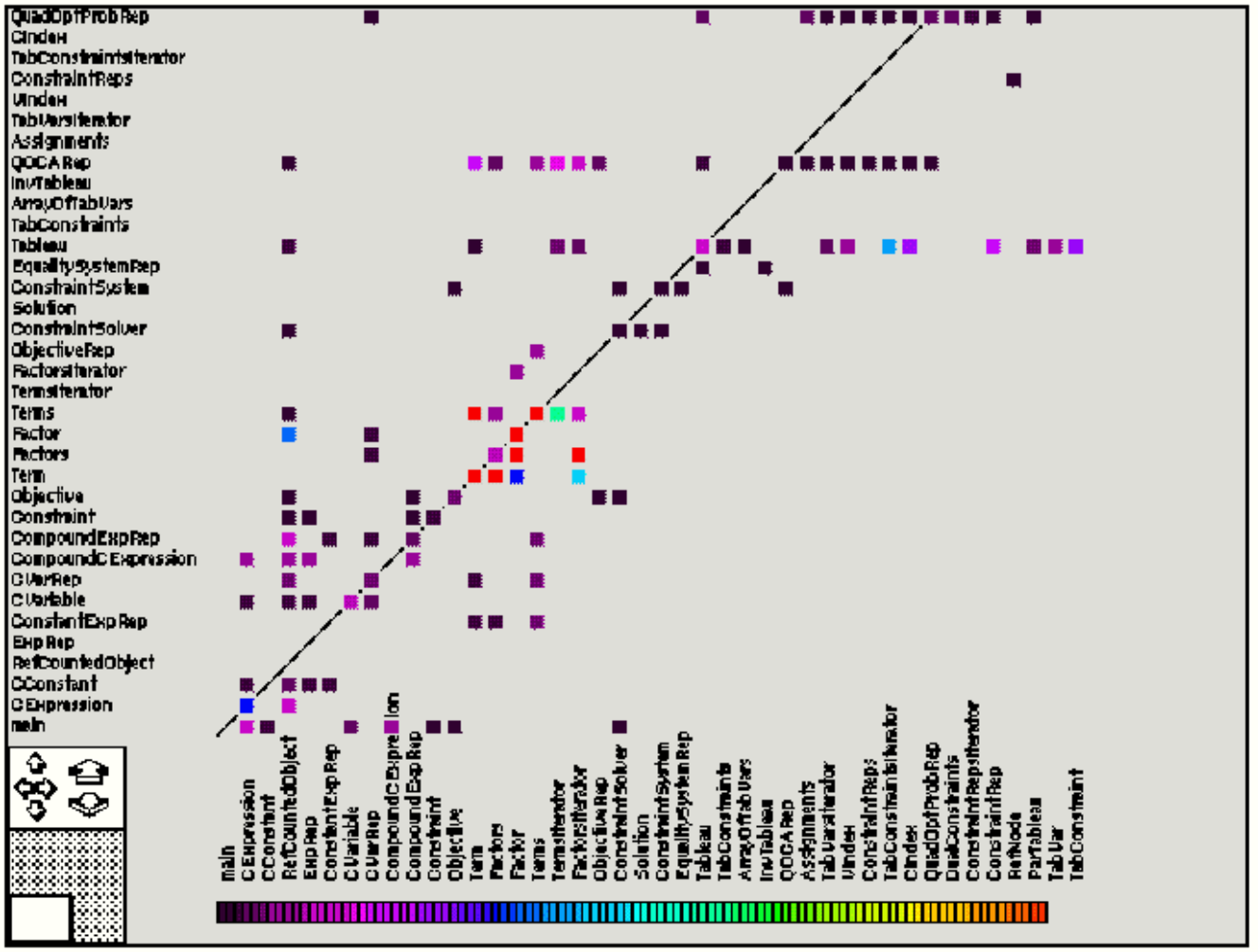
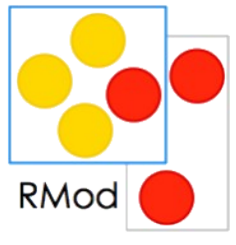



Figure 6: Inter-class call matrix

RMod

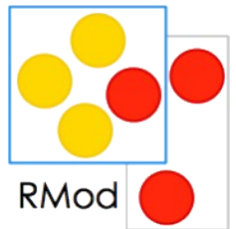
# Dynamic Code Visualization

- Code instrumentation problem
  - Logging, Extended VMs, Method Wrapping
- Scalability problem
  - Traces quickly become very big (1Mb/s)
- Completeness problem: scenario driven
- Pros:
  - Good for fine-tuning, problem detection
- Cons:
  - Tool support crucial
  - Lack of abstraction without tool support



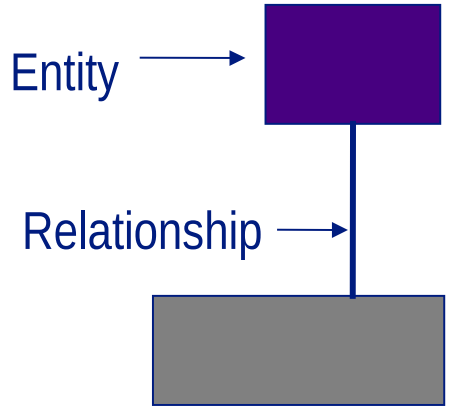
# *Do-It-Yourself* Considerations

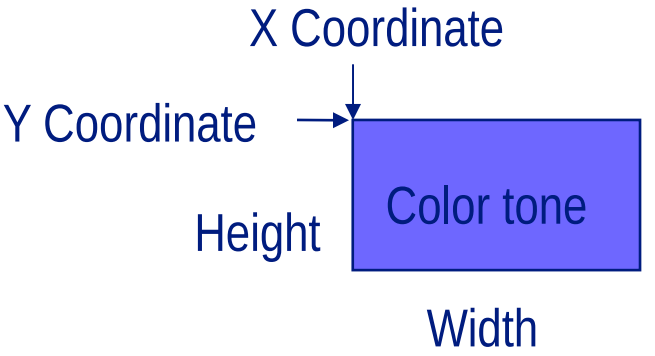
- A decent graph layout can be a hard task...
  - Algorithmic aspects may be important
  - Efficient space use (physical limits of a screen)
  - Colours are nice, but... there are no conventions!
  - Trade-off between usefulness and complexity
- Keeping a focus is hard:
  - Beautiful graphs are not always meaningful
  - Where should we look?
  - What should we look for?
- Which approach can be reproduced by reengineers in work context and provides useful information?



# One Solution: Combining Metrics and Visualization

- 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 (h=1; w=2)
    - Color (3)
    - Position (x=4; y=5)

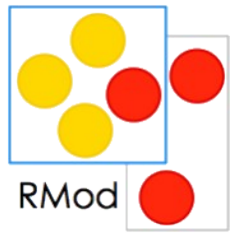


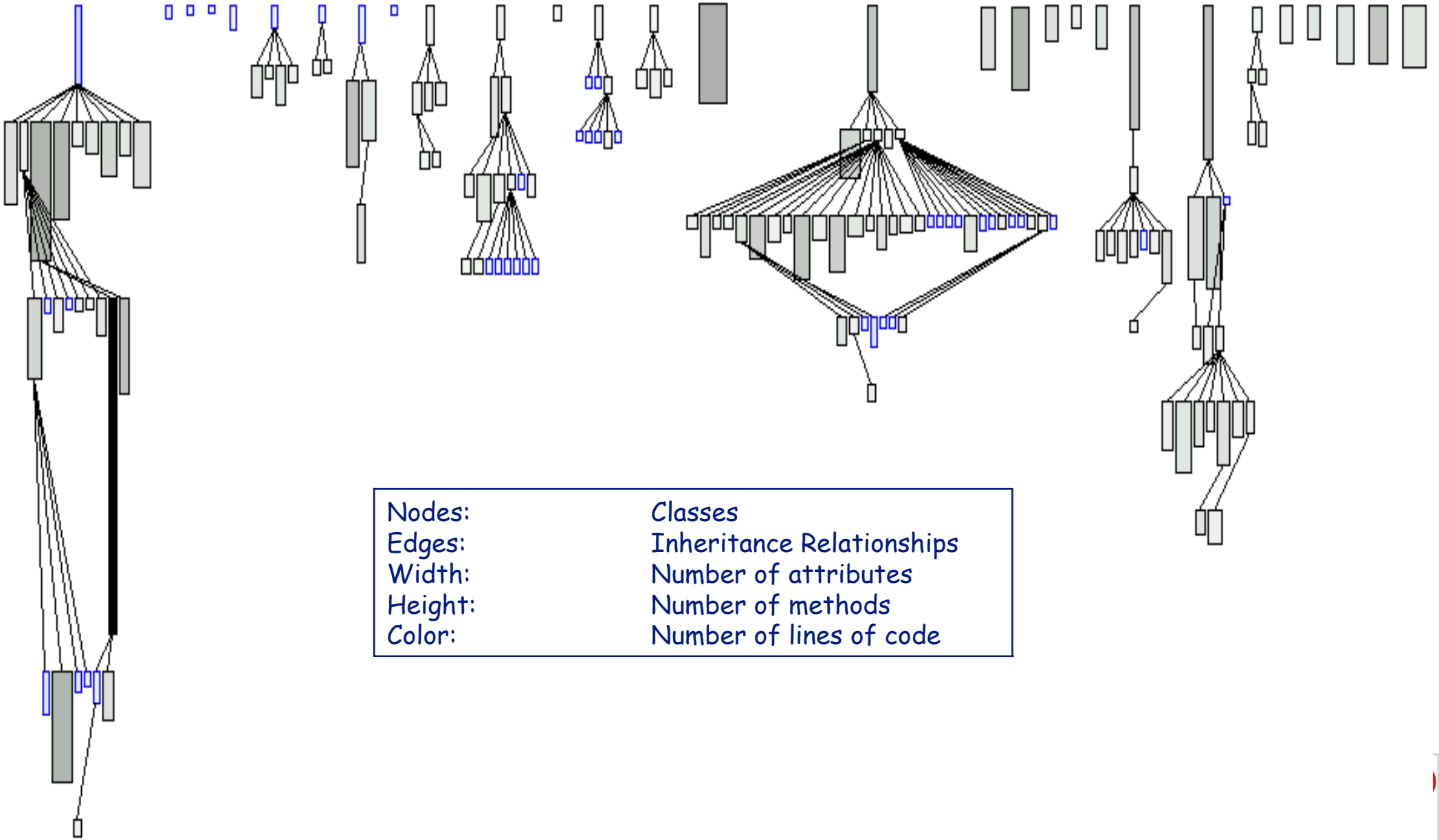

RMod

# Combining Metrics and Visualization

- Metrics
  - Scale well
  - Simple metrics ⇒ simple extraction (scripts)
  - But numerical combination is meaningless
- Simple Graphs
  - Easy to draw
  - Scale well
  - But not enough semantics

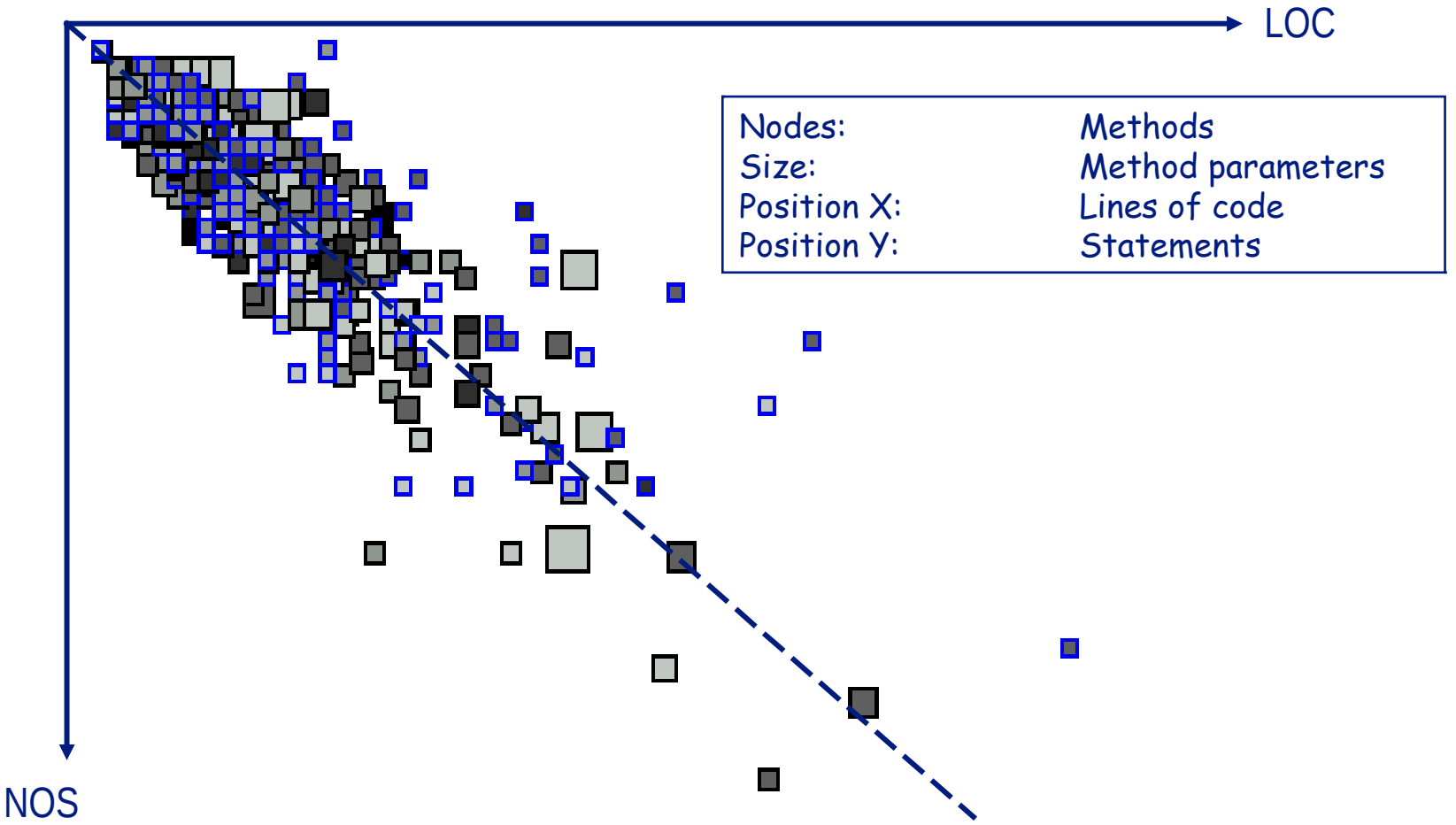


# System Complexity View



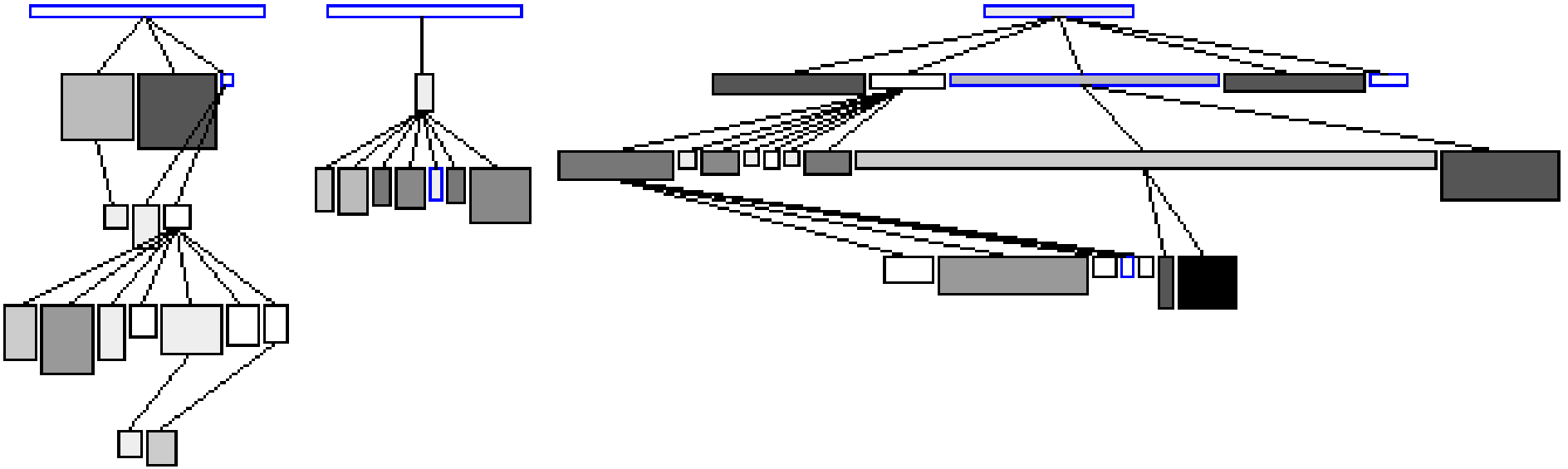
| | |
|---|---|
| Nodes: | Classes |
| Edges: | Inheritance Relationships |
| Width: | Number of attributes |
| Height: | Number of methods |
| Color: | Number of lines of code |

# Method Assessment




RMod

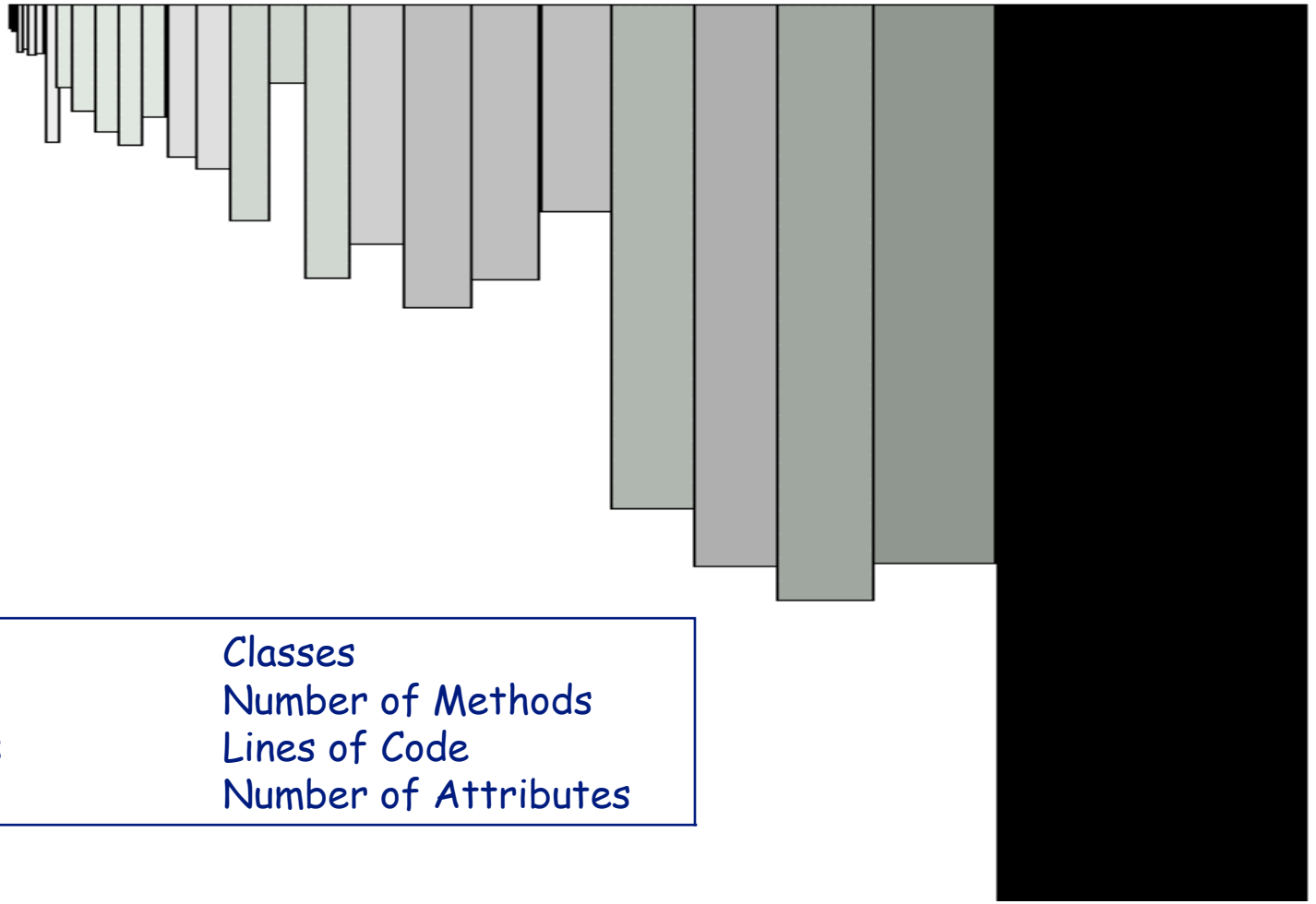
# Inheritance Classification



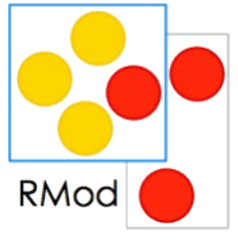
| | |
|---|---|
| Boxes: | Classes |
| Edges: | Inheritance |
| Width: | Number of Methods Added |
| Height: | Number of Methods Overridden |
| Color: | Number of Method Extended |

RMod

# Data Storage Class Detection

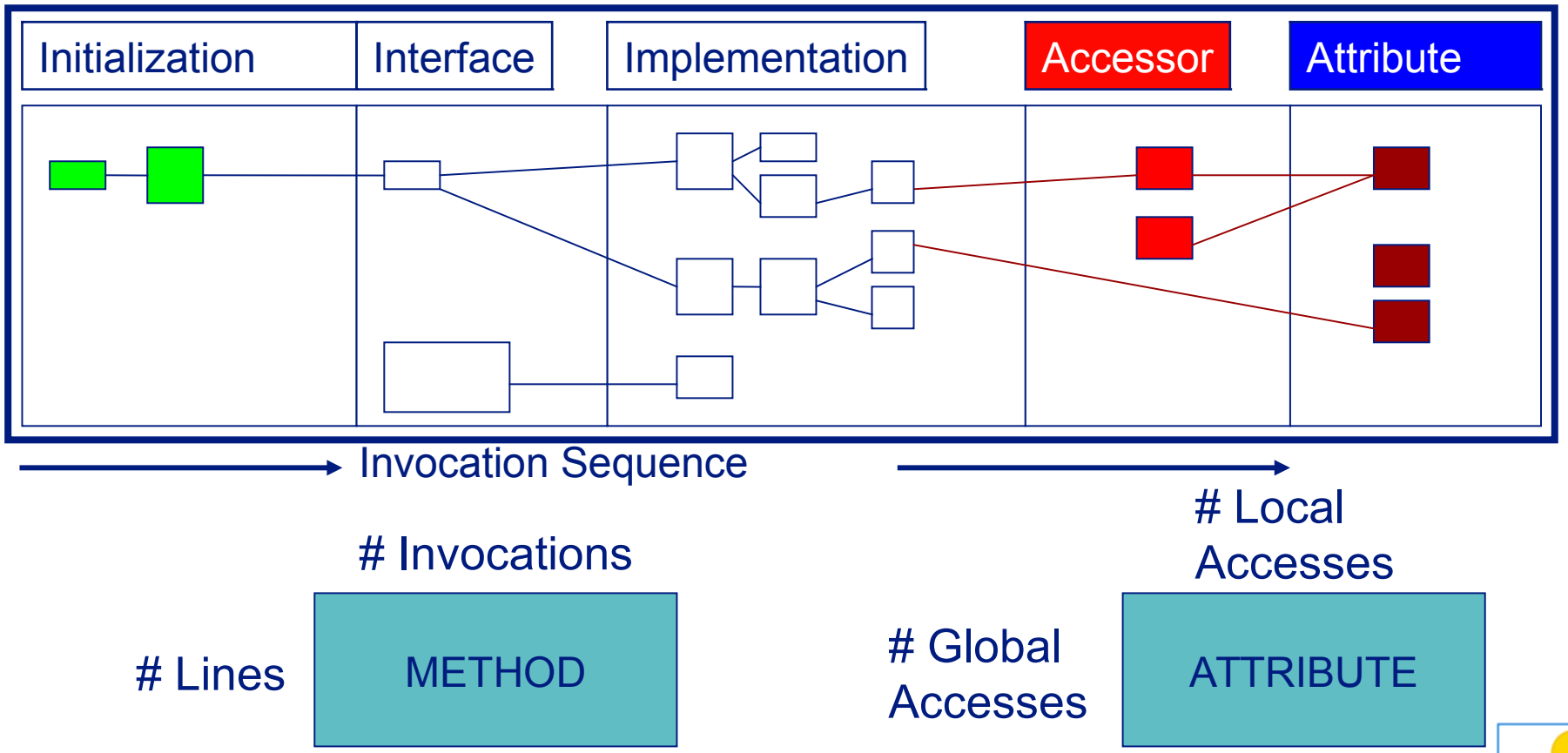


| | |
|---|---|
| Boxes: | Classes |
| Width: | Number of Methods |
| Height: | Lines of Code |
| Color: | Number of Attributes |



# Finer granularity

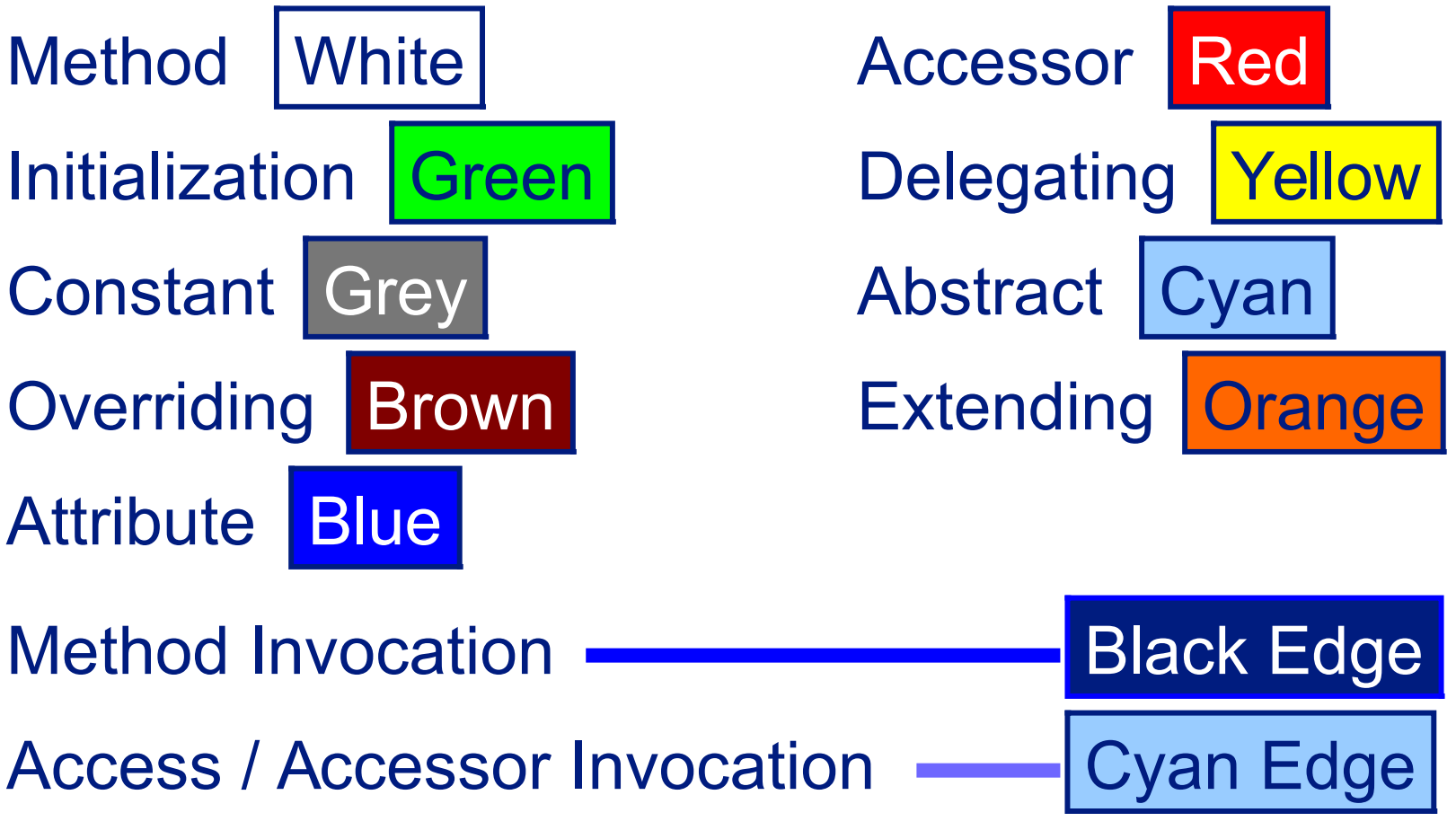
- Class Blueprint




RMod

# Finer granularity

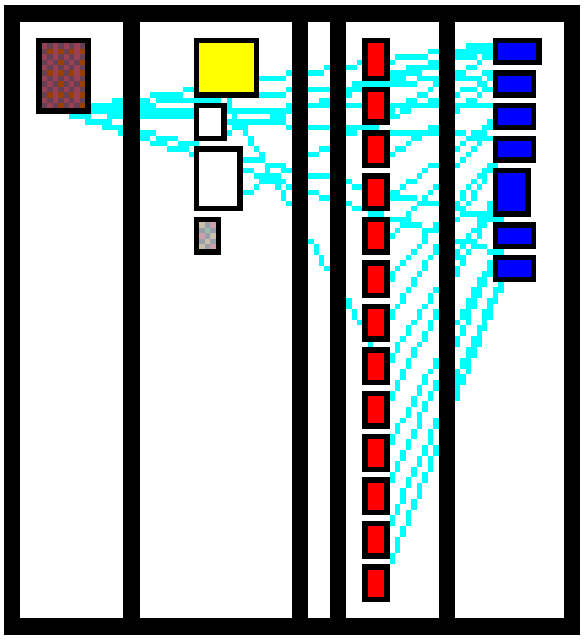
- Class Blueprint, adding Semantic Information




RMod

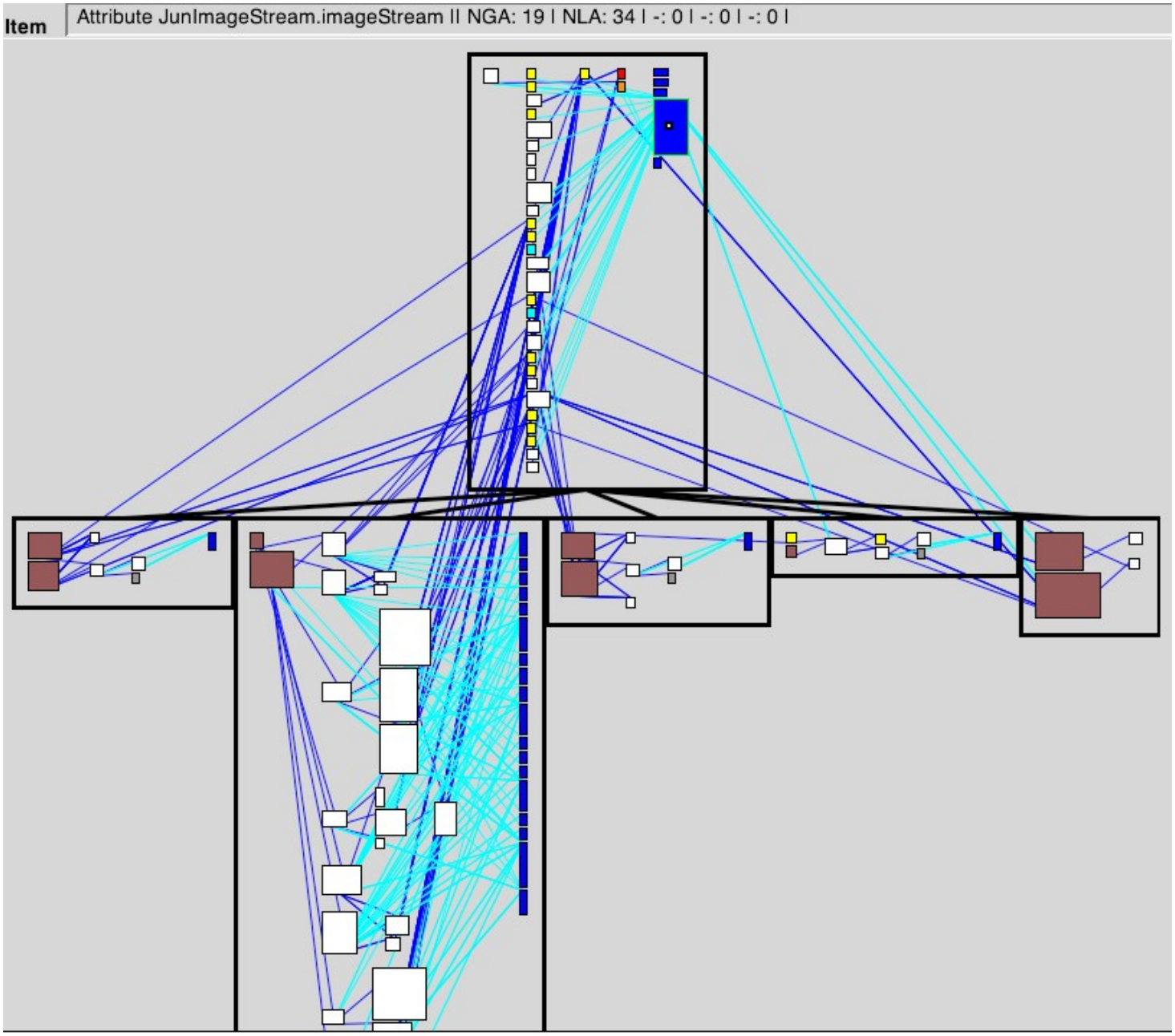
# Class Blueprint

- Data Storage
  - Has many attributes
  - May have many accessor methods
  - No complex behavior



RMod

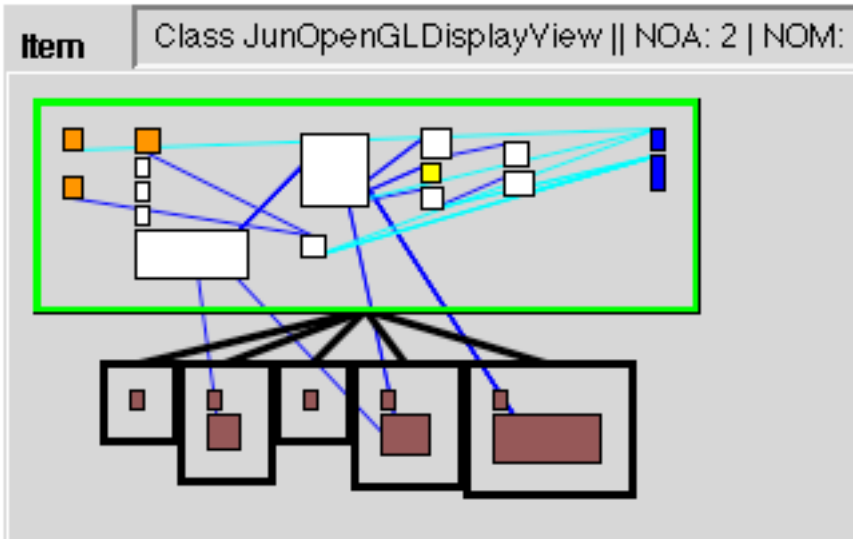
# Class Blueprint: Inheritance




RMod

# Class Blueprint: Inheritance

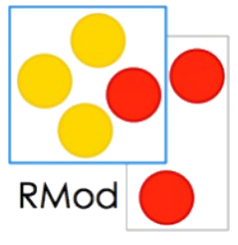
- Delegator:
  - Delegates functionality to other classes
  - May act as a “Façade”




RMod

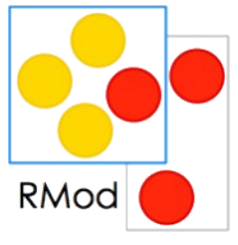
# Evaluation

- Pros
  - Quick insights
  - Scalable
  - Metrics add semantics
  - Interactivity makes the code “come nearer”
  - Reproducible
  - Industrial Validation is the acid test
- Cons
  - Simple
  - Useful in a first approach phase
  - Code reading needed
  - Level of granularity

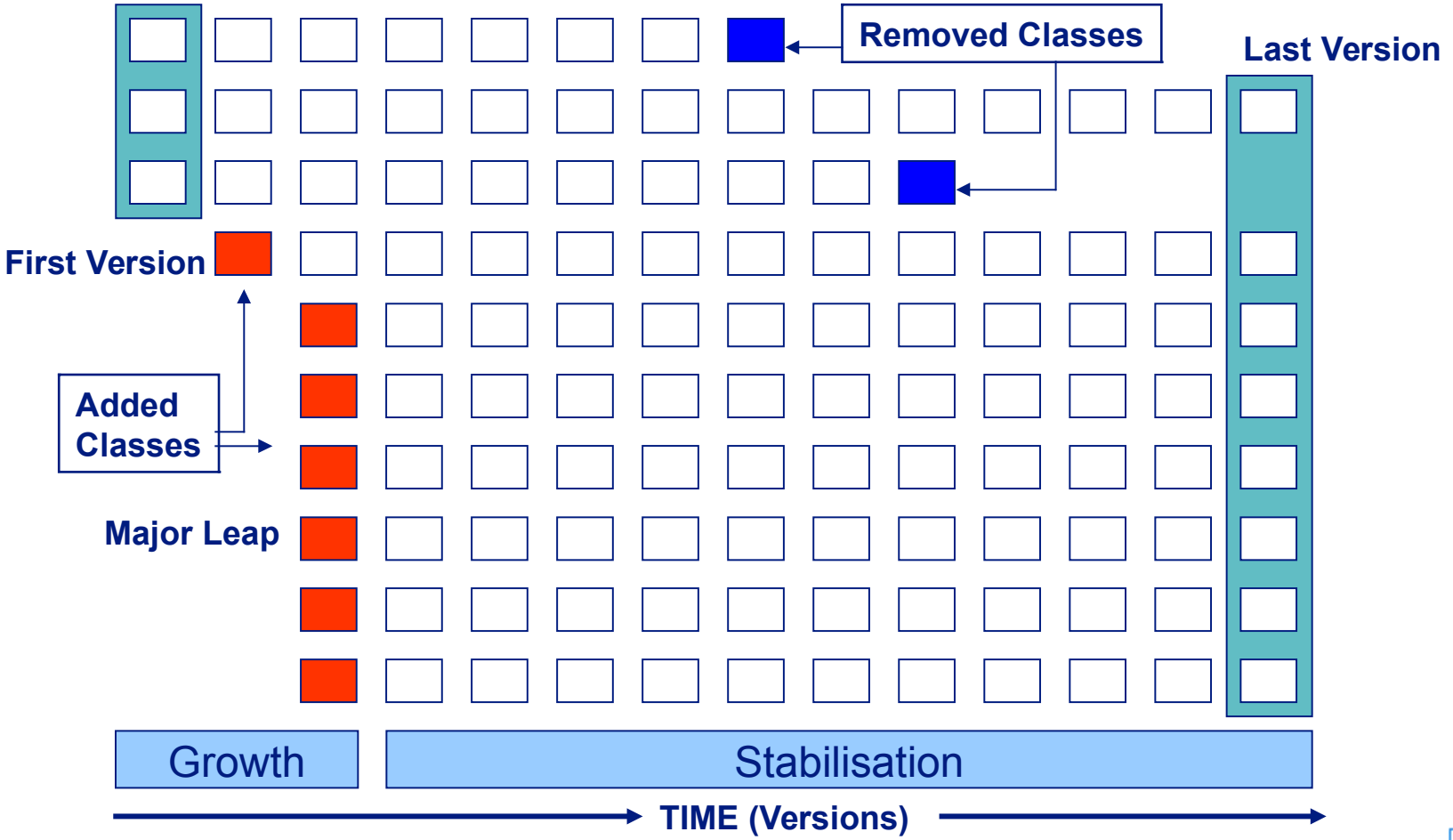


# Understanding Evolution

- Information is in the history!
- Overwhelming complexity
- How can we detect and understand changes?
- One solution: The Evolution Matrix

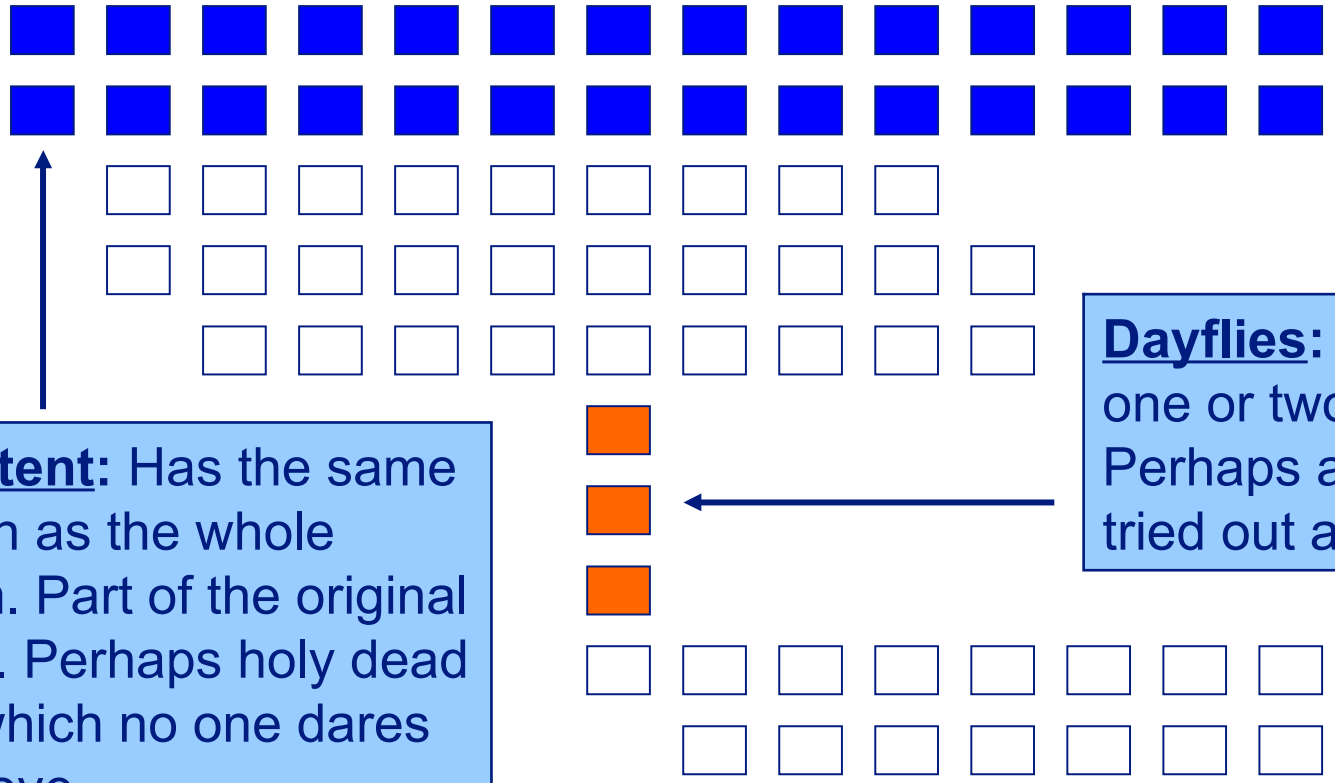


# One solution: The Evolution Matrix




RMod

# One solution:
# The Evolution Matrix



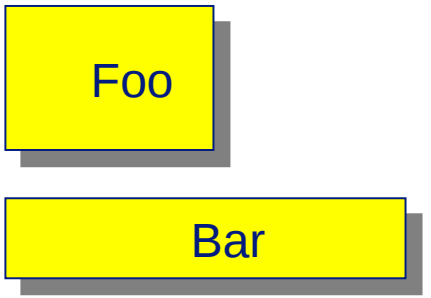
**Persistent:** Has the same lifespan as the whole system. Part of the original design. Perhaps holy dead code which no one dares to remove.

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

RMod

# History from metrics

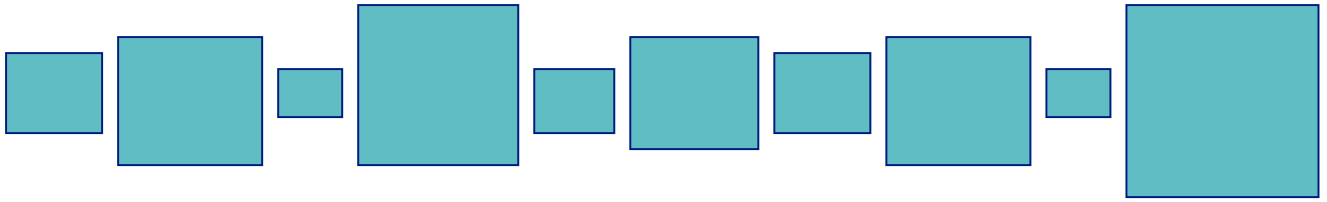
- Object-Oriented Programming  is about “state” and “behavior”:
  - State is encoded using attributes
  - Behavior is encoded with methods
- Classes are rectangles:
  - width=NOM (number of methods)
  - height=NOA (number of attributes)



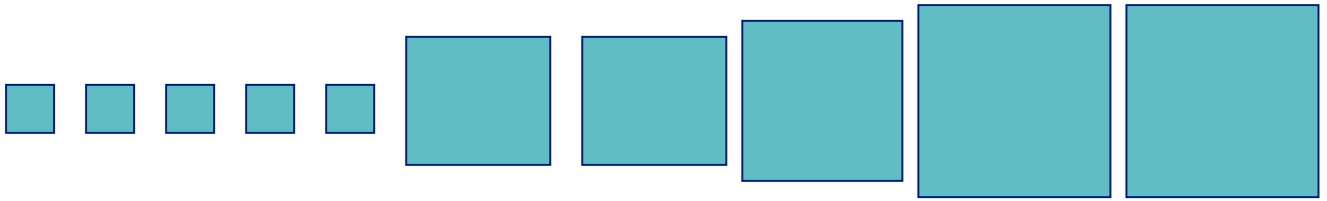

- The Classes can be categorized according to their “personal evolution” and to their “system evolution”: Pulsar, Supernova,Red Giant, Stagnant,DayflyPersistent

RMod

# History from metrics



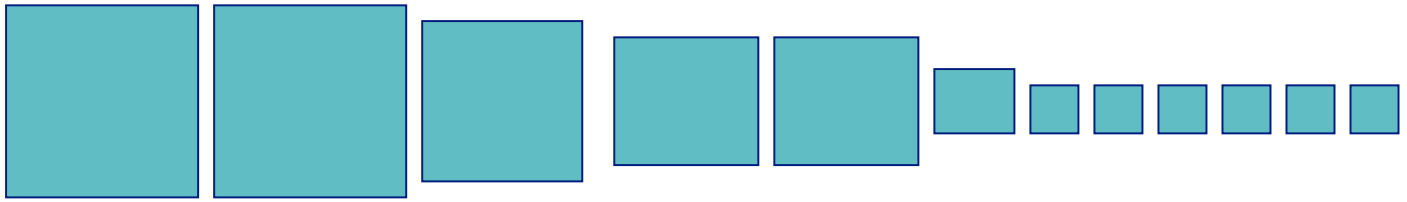
**Pulsar:** Repeated Modifications make it grow and shrink.
System Hotspot: Every System Version requires changes.



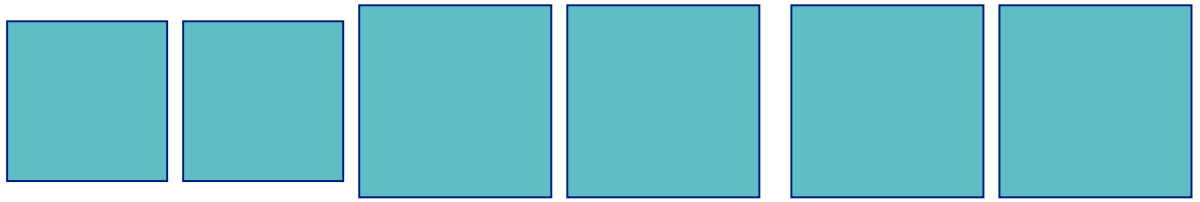
**Supernova:** Sudden increase in size. Possible Reasons:
- Massive shift of functionality towards a class.
- Data holder class for which it is easy to grow.
- *Sleeper*: Developers knew exactly what to fill in.

RMod

# History from metrics



**White Dwarf:** Lost the functionality it had and now trundles along without real meaning. Possibly dead code.

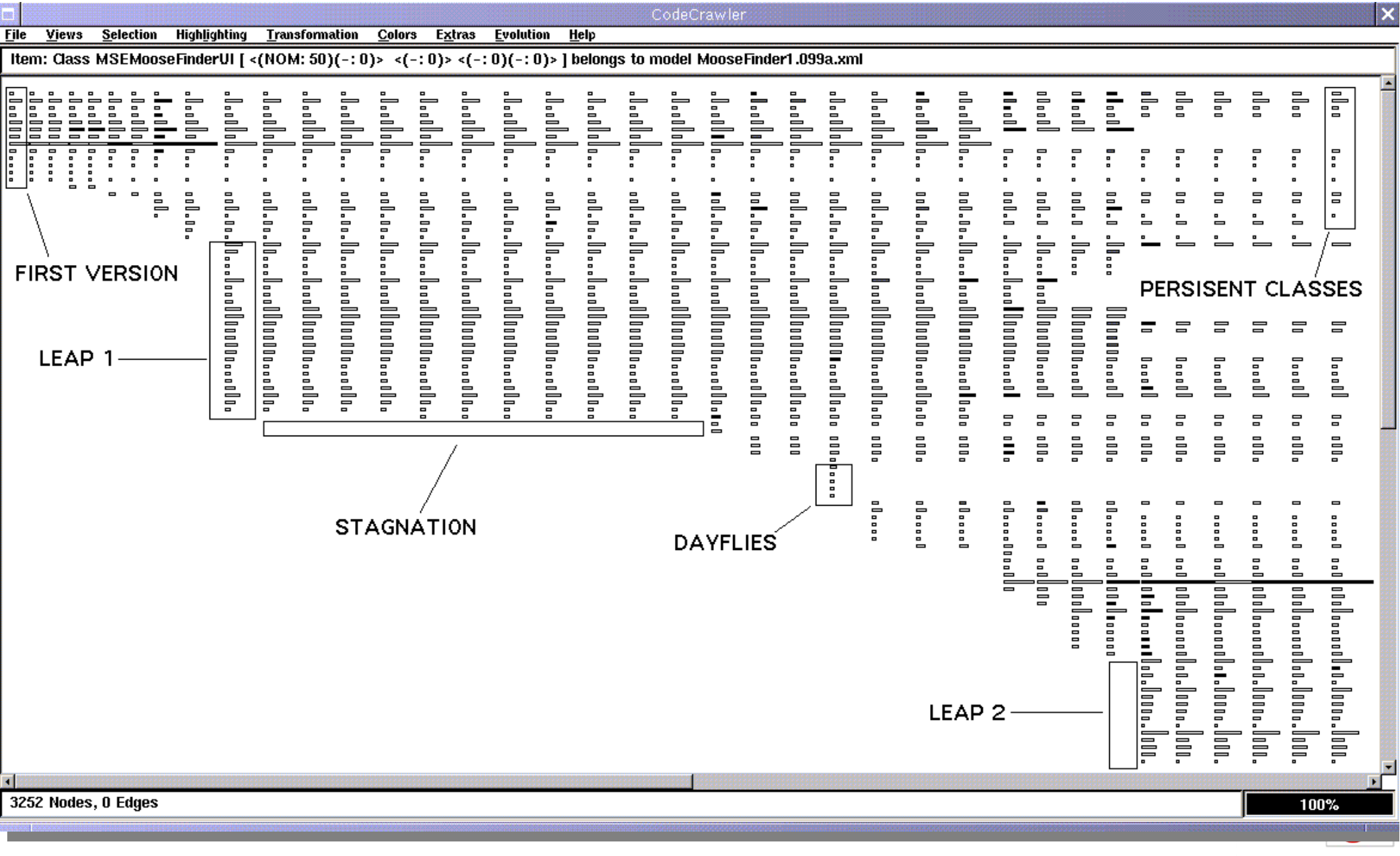


**Red Giant:** A permanent god class which is always very large.

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

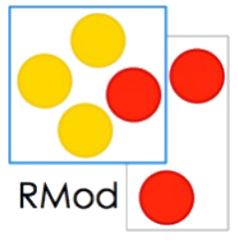
RMod

# Example: MooseFinder (38 Versions)



# Conclusions

- Visualization can be very useful when used correctly
- An integrated approach is needed, just having nice pictures is not enough
  - Interaction
  - Access to the code
- Only people that know what they see can react on it (or expert/advanced developers)



# Lessons Learned

- Visualization is not just smoke and mirrors!
  - Complexity reduction, abstraction
- But it should be adapted to
  - your goal (ex: first contact, deep understanding),
  - time (2 days / 1 month),
  - size (a complete system or 3 classes)
- Minimize tool support if you are not familiar
- Simple approaches give 80%, the last 20% are hard to get