Software Evolution Analysis & Visualization

Harald C. Gall

s.e.a.l. - software evolution & architecture lab
University of Zurich, Switzerland
http://seal.ifi.uzh.ch/gall

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Abstract

Software repositories such as versioning systems, defect tracking systems, and archived communication between project personnel are used to help manage the progress of software projects. There is great potential in mining this information to support the evolution of software systems, improve software design or reuse, and empirically validate novel ideas and techniques. Research is now proceeding to uncover ways in which mining these repositories can help to understand software development, to support predictions about software development, and to plan various evolutionary aspects of software projects.

This seminar presents some analysis and visualization techniques to understand software evolution by exploiting the rich sources of artifacts that are available. Based on the data models, that need to be developed to cover sources such as modification and bug reports, we describe some of our recent efforts to extract and analyze developer patterns, change couplings, and fine-grained change types.

Instructor Biographies

Harald C. Gall

Professor of Software Engineering, Department of Informatics, University of Zurich, Switzerland.

Prior, Associate Professor at the TU Vienna

Research interests are in:

- software engineering with focus on
- software evolution, software architecture,
- reengineering, program families, and
- distributed and mobile software engineering processes.

Program chair of ESEC-FSE 2005, IWPC/ICPC 2000 & 2005, IWPSE 2004, and MSR 2006 & 2007.

Program co-chair of ICSE 2011

Michael Würsch

Research Assistant, Department of Informatics, University of Zurich, Switzerland

MSc in Informatics, UZH

Research interests in:

- software design
- software evolution analysis
- developer support
- search-driven software engineering

Objectives of the Course

Goal: Investigate means to analyze and control the evolution of object-oriented software systems at various levels.

Specifically, the course aims to answer the following questions:

- How does the architecture of a software system evolve over time? What are signs of architectural decay and how can they be tracked down?
- How can hidden dependencies in a system that complicate and hinder its evolution be discovered?
- How can the plethora of software data (such as source code, change and bug history, release data) be filtered and visualized? What are effective visualization models and techniques for that?

Agenda

I. Software Analysis

- Techniques and Tools
- Reengineering Patterns

II. Software Visualization

- Polymetric Views
- Class Blueprints
- Software as a City

III. Software Evolution Analysis

- Release History Data
- Change Coupling

IV. Software Quality Assessment

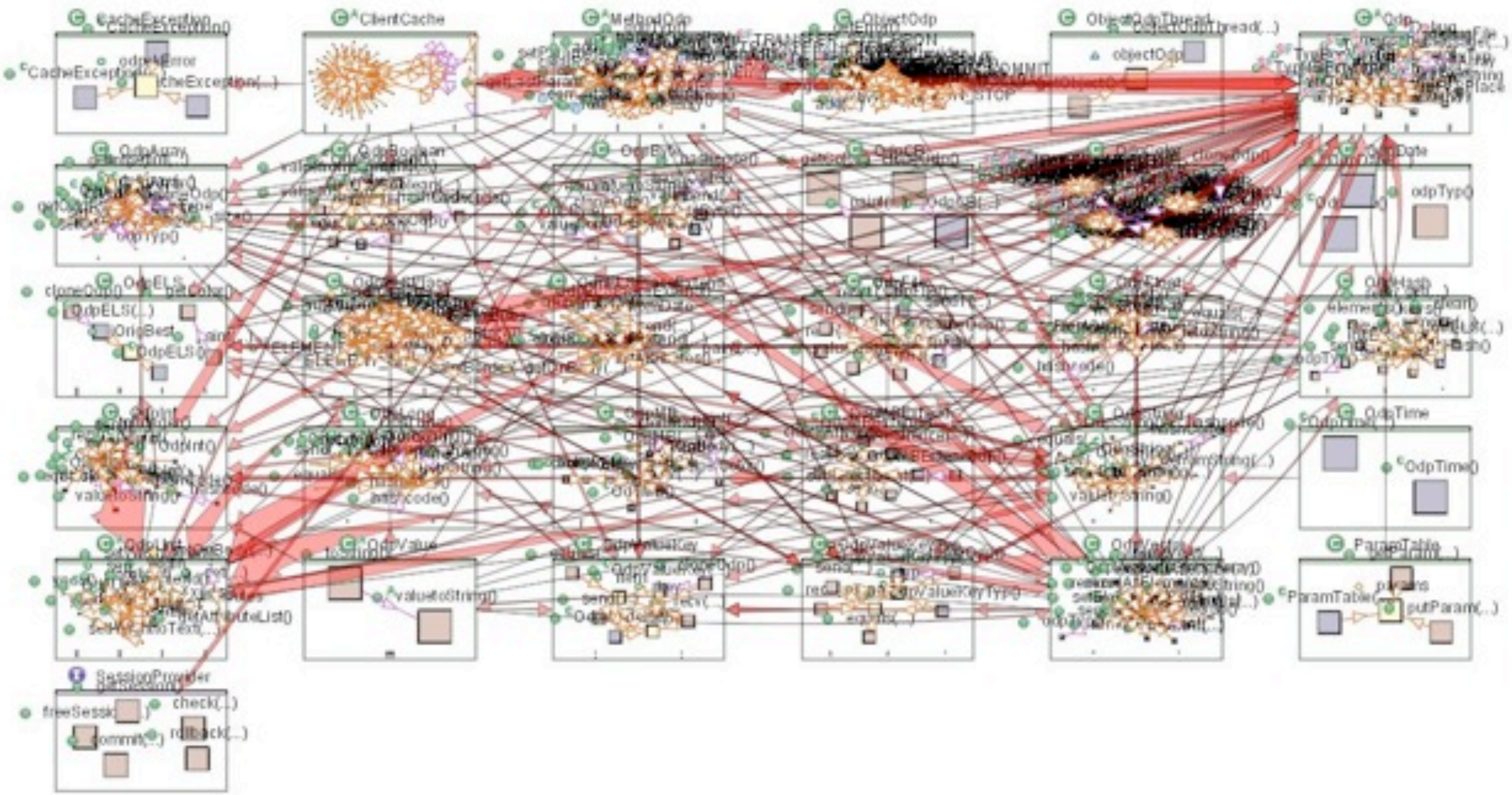
- Design Heuristics
- Software Metrics
- Code Clones

V. Empirical Studies

- Developer networks
- Cross-project failure prediction
- Distributed Development

Background & Motivation

Real life is complex



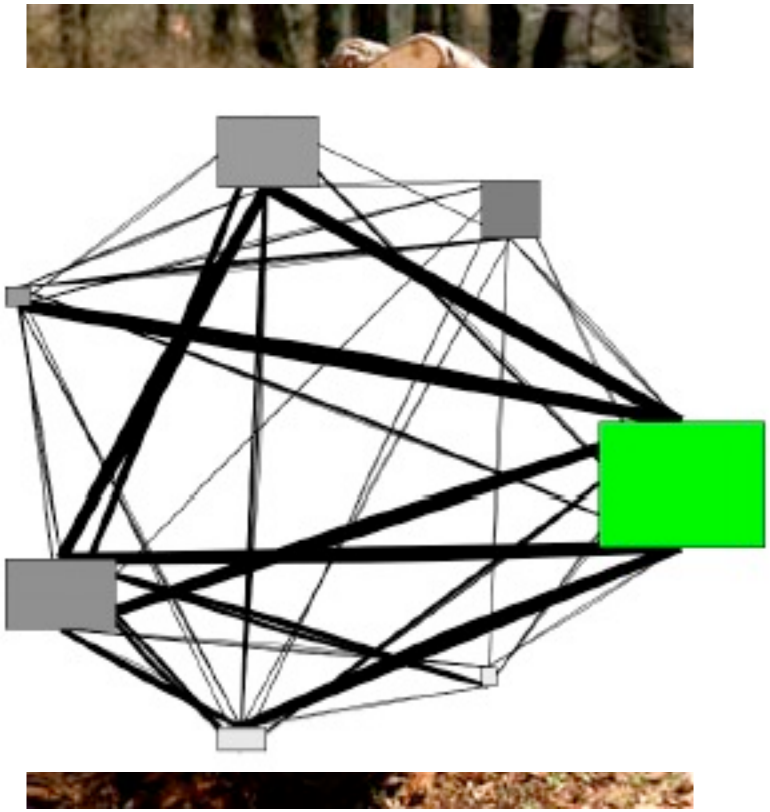
Software evolves ...

Software evolves ...

Trees:
annual rings

Software evolves ...

Trees:
annual rings



Software:
structural
changes

It‘s about complexity ...

Corollary to Moore's Law:
The complexity of software doubles every two years.

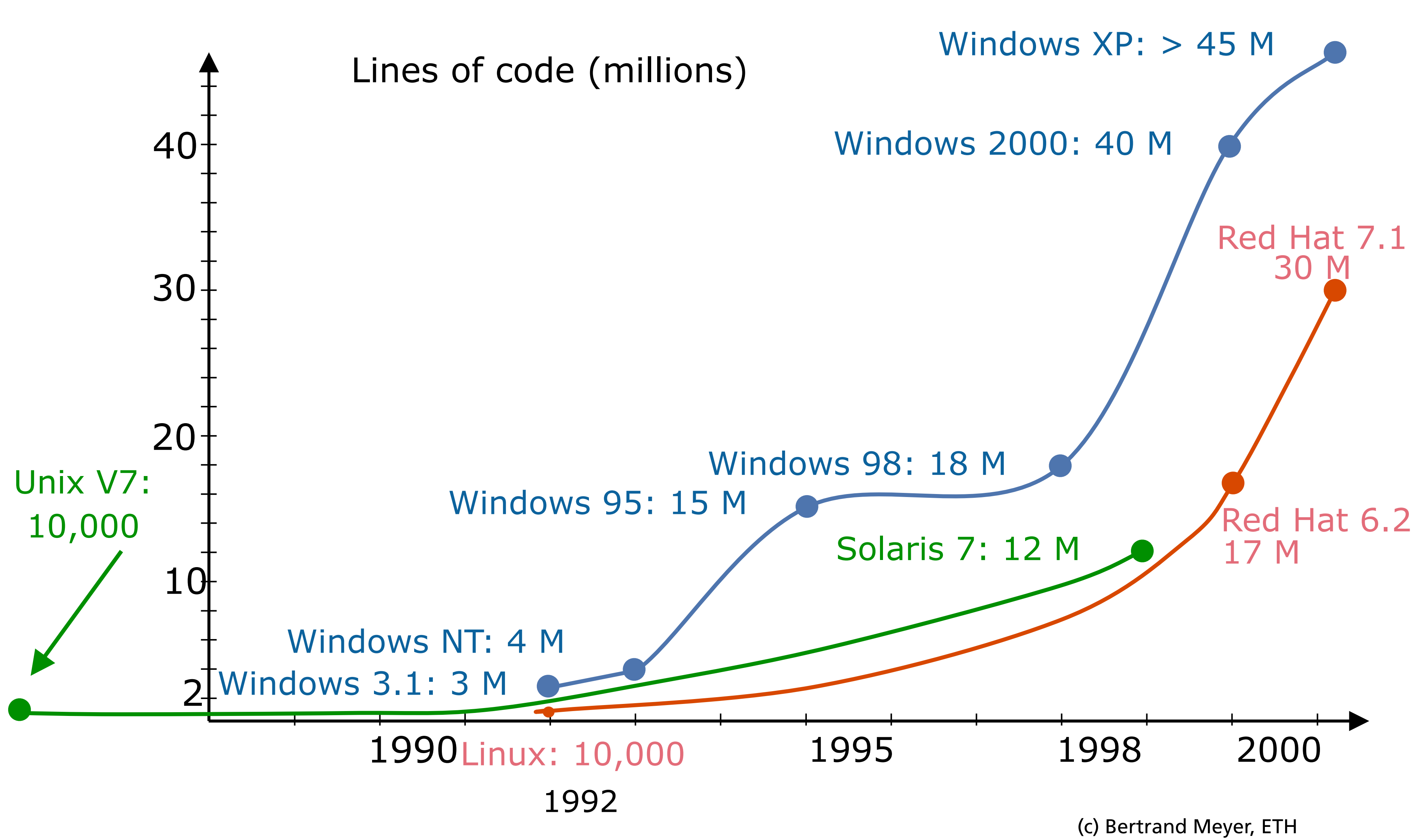
IDC study

- 15 years ago, firms were spending 75% of their IT budget on new hardware and software ...
- ... now that ratio has been reversed to fixing things

In Siemens (Reinhold Achatz, ICSE 2006)

- Only 40% is new development, the rest is evolution and maintenance
- 80% of products is software
- > 80% of Siemens internal companies are CMM 3+

Size of Operating Systems (LOC)



Why analyze Software Evolution?

„Nevertheless, the industrial track record raises the question, why, despite so many advances, [...]

- satisfactory functionality, performance and quality is only achieved over a lengthy evolutionary process,
- software maintenance never ceases until a system is scrapped
- software is still generally regarded as the weakest link in the development of computer-based systems".

Lehman et al., 1997

Software entropy

Laws of Software Evolution [Lehman and Belady]

- Continuing change
- Increasing entropy/complexity
- Increasing size

Maintenance increases „software entropy“

- Erosion of architecture, design, modularization
- Increase of interdependencies between software parts („Coupling“)
- Decrease of orthogonal separation of concerns („Cohesion“)

What is Software Evolution Analysis?

Investigating the evolution of a software system to identify potential shortcomings in its architecture or logical structure.

Structural shortcomings can then be subject to reengineering or restructuring.

Reverse Engineering: What and Why ?

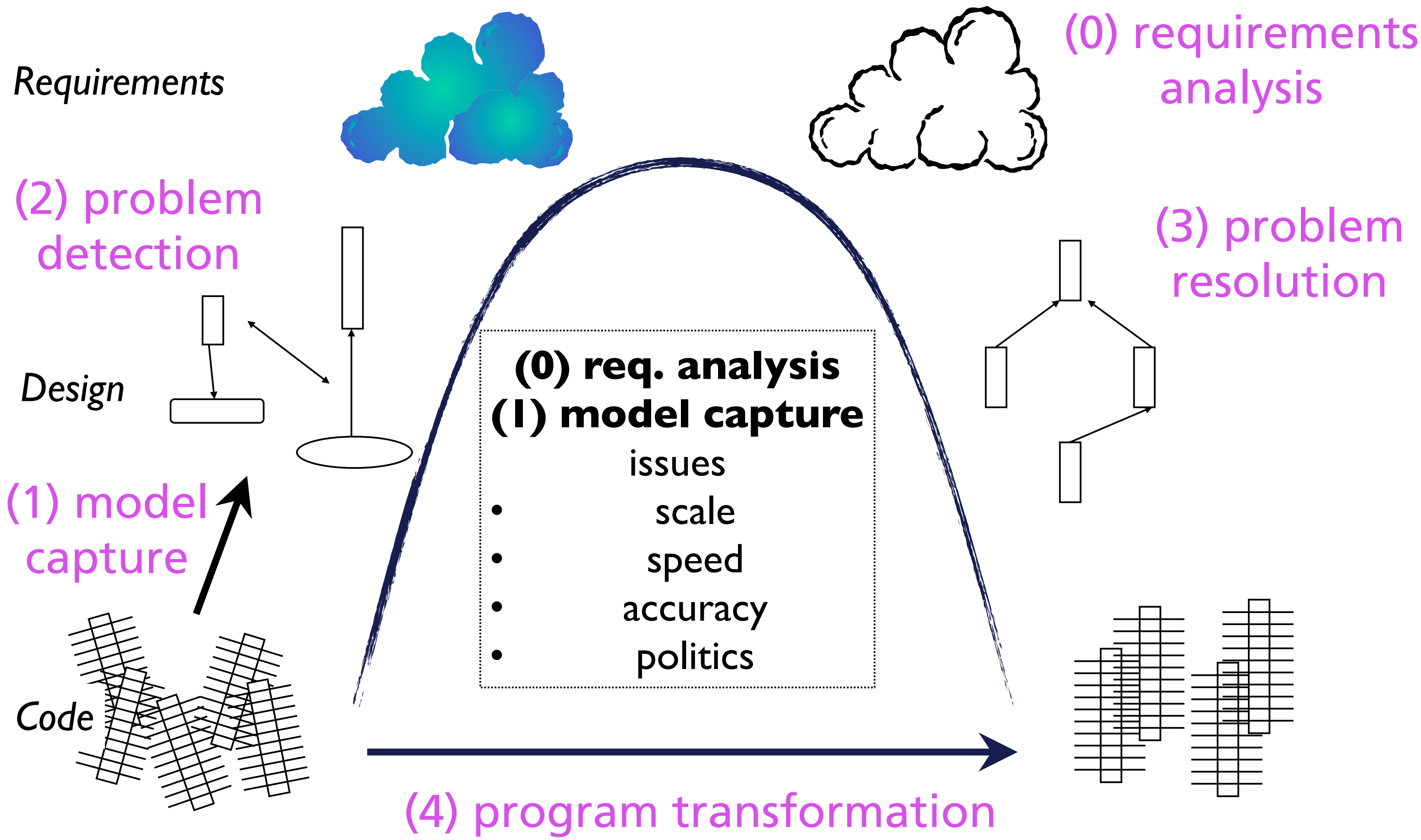
Definition: Reverse Engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. — Chikofsky & Cross, '90

Motivation: Understanding other people's code (cf. newcomers in the team, code reviewing, original developers left, ...)

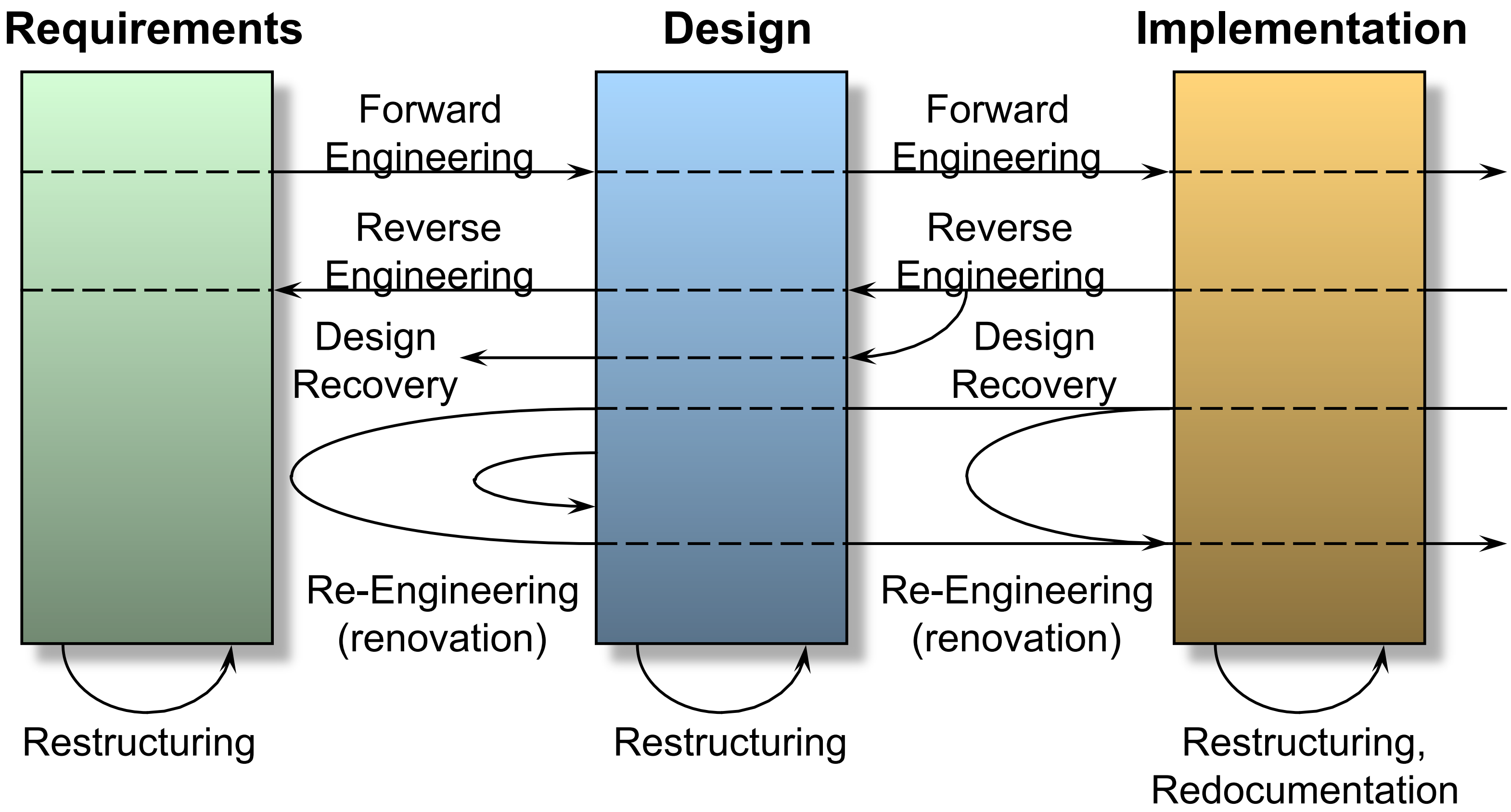
Generating UML diagrams is NOT reverse engineering
... but it is a valuable support tool

I. Software Analysis

The Reengineering Life-Cycle



Reverse Engineering Terminology



Chikofsky, Cross '90

Reverse Engineering Patterns

Reverse engineering patterns encode expertise and trade-offs in extracting design from source code, running systems and people.

Even if design documents exist, they are typically out of sync with reality.

Example:

Read all the Code in One Hour

Speculate about the Design

Interview During Demo

www.iam.unibe.ch/~scg/OORP/

Reengineering Patterns

Reengineering patterns encode expertise and trade-offs in transforming legacy code to resolve problems that have emerged.

- These problems are typically not apparent in original design but are due to architectural drift as requirements evolve

Example:

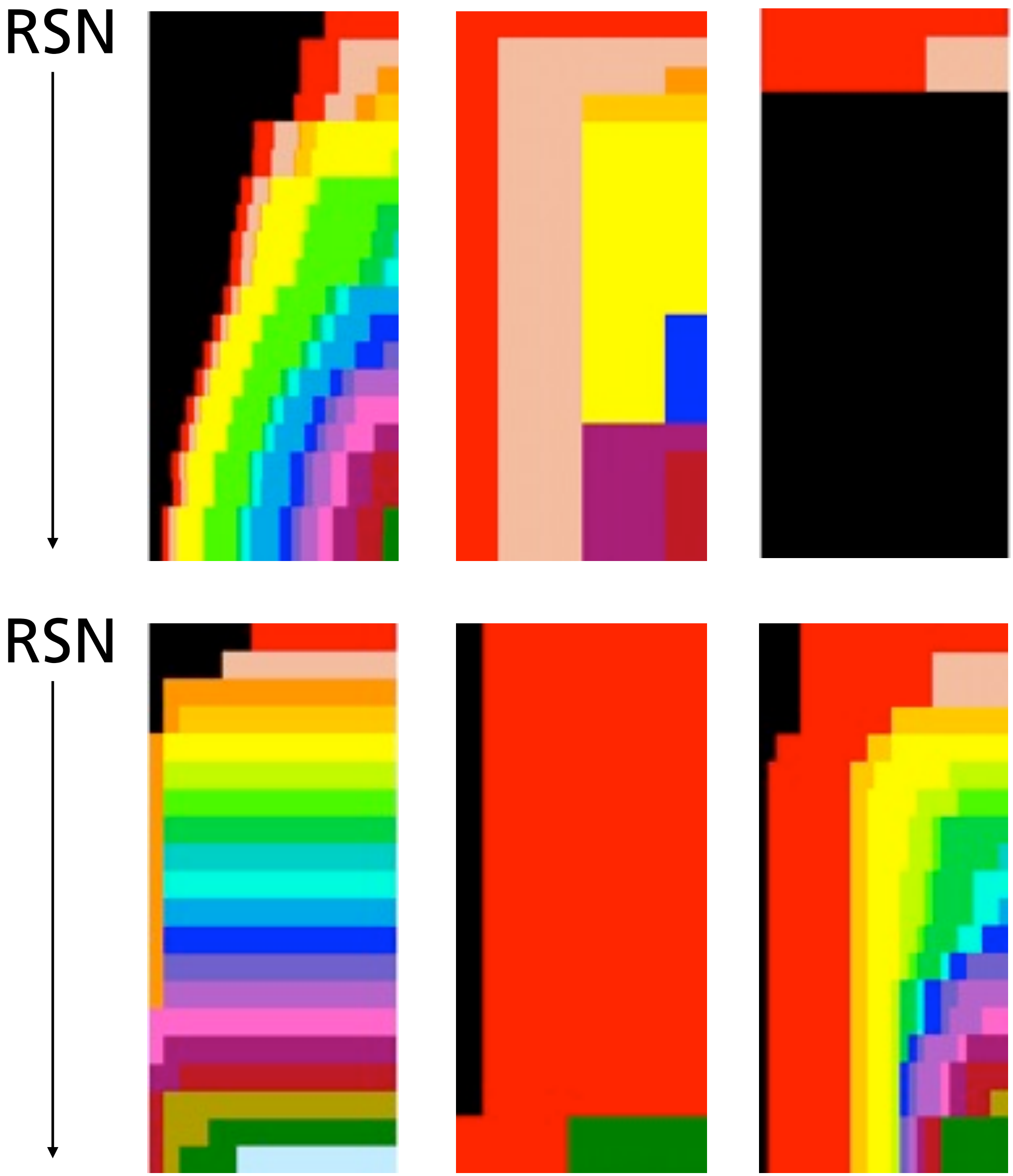
- Move Behavior Close to Data
- Build a Bridge to the New Town

Case Study: Telecom Switching System

10 Million LOC

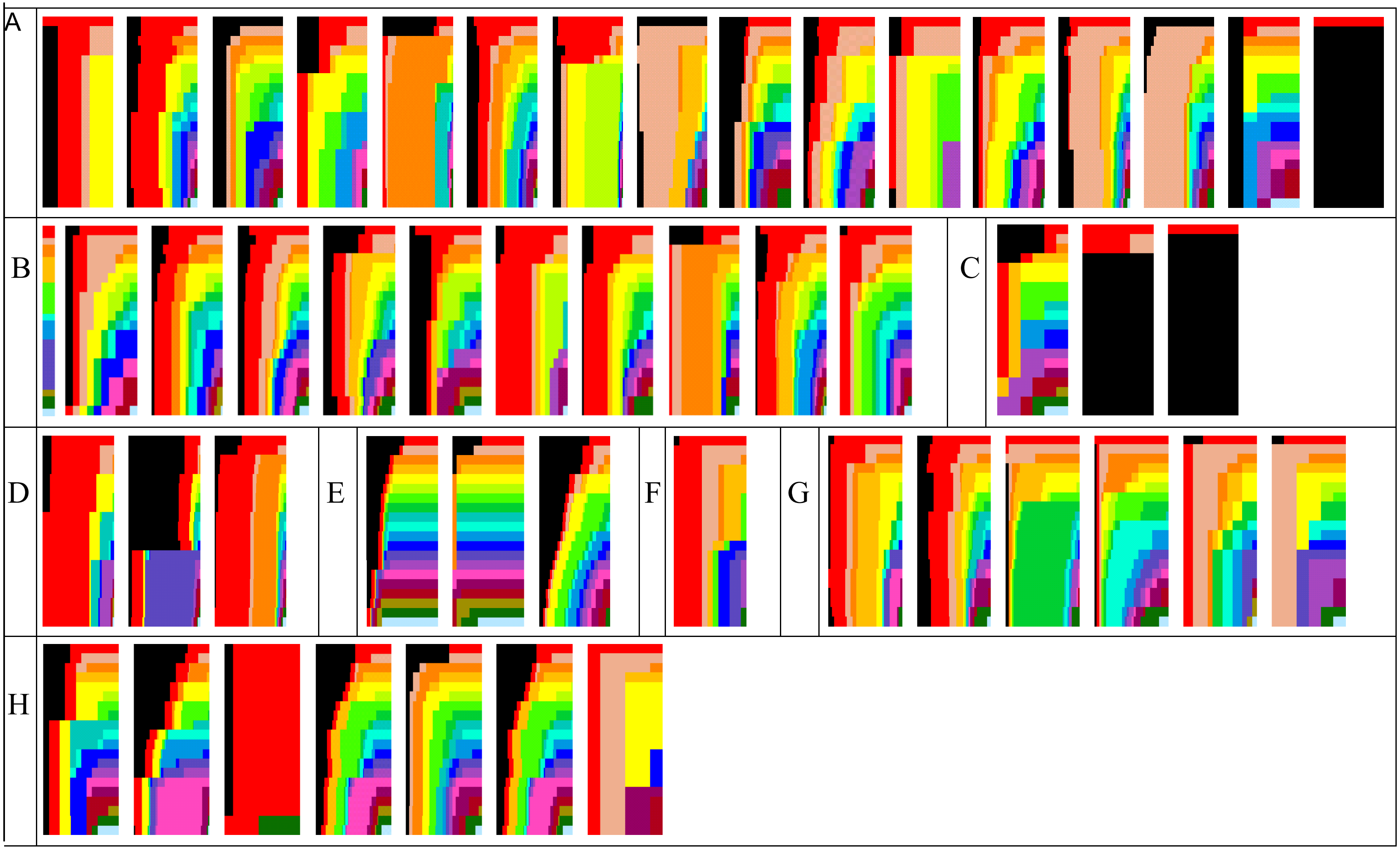
4 programming languages

20 releases



RSN ... Release Sequence Number

TSS visualized



II. Software Visualization

Contents

Information Visualization

The human eye and brain interpret visual information in order to “react to the world”

We want to answer questions on what we perceive

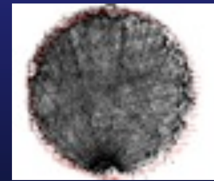
J. Bertin inferred three levels of questions

- Lower perception (one element)
- Medium perception (several elements)
- Upper perception (all elements/the complete picture)

Information Visualization is about

- how to display information
- how to reduce its complexity

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"

2 main fields:

(Algorithm Animation)

Program Visualization

Conceptual Problem

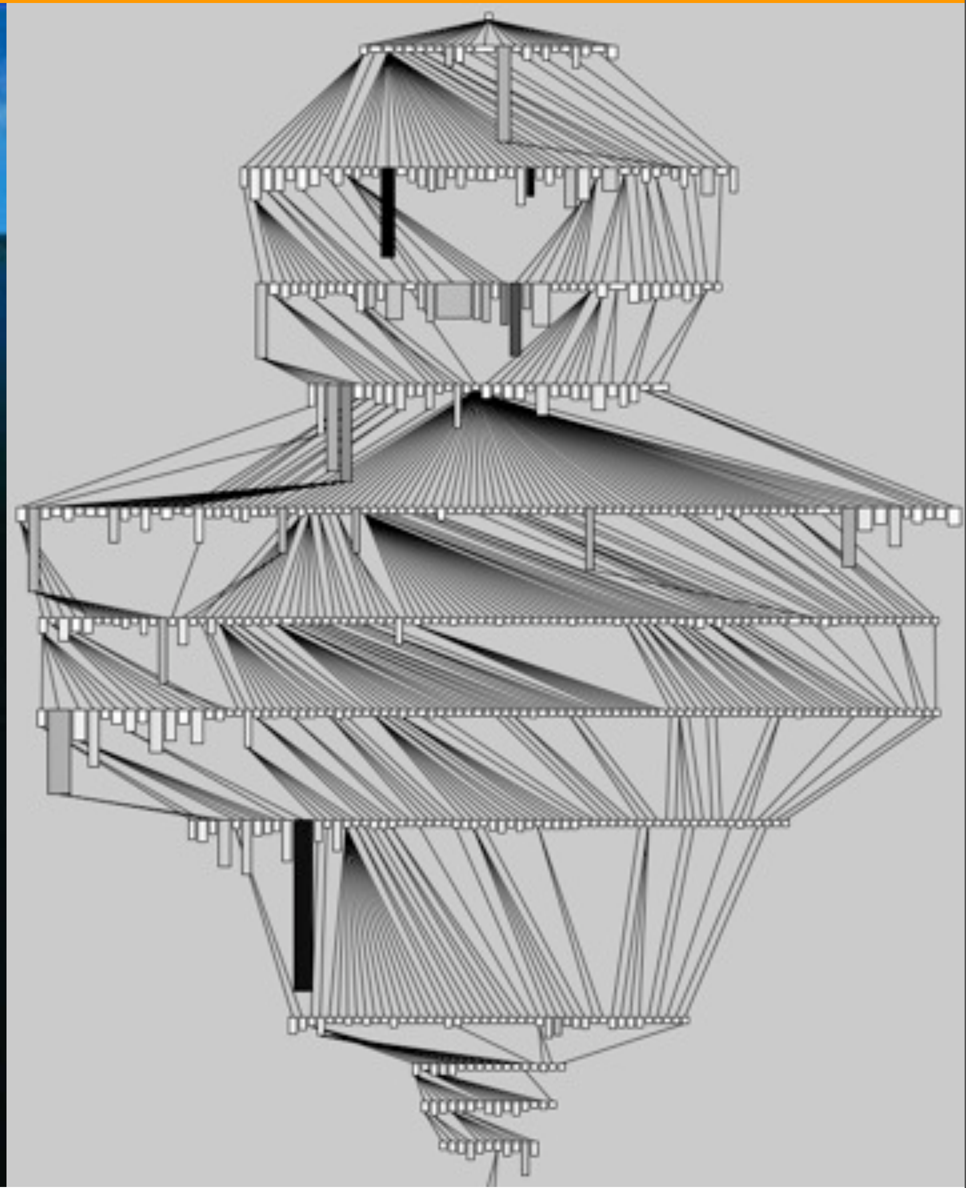
"Software is intangible, having no physical shape or size. Software visualization tools use graphical techniques to make software visible by displaying programs, program artifacts and program behavior."

Thomas Ball

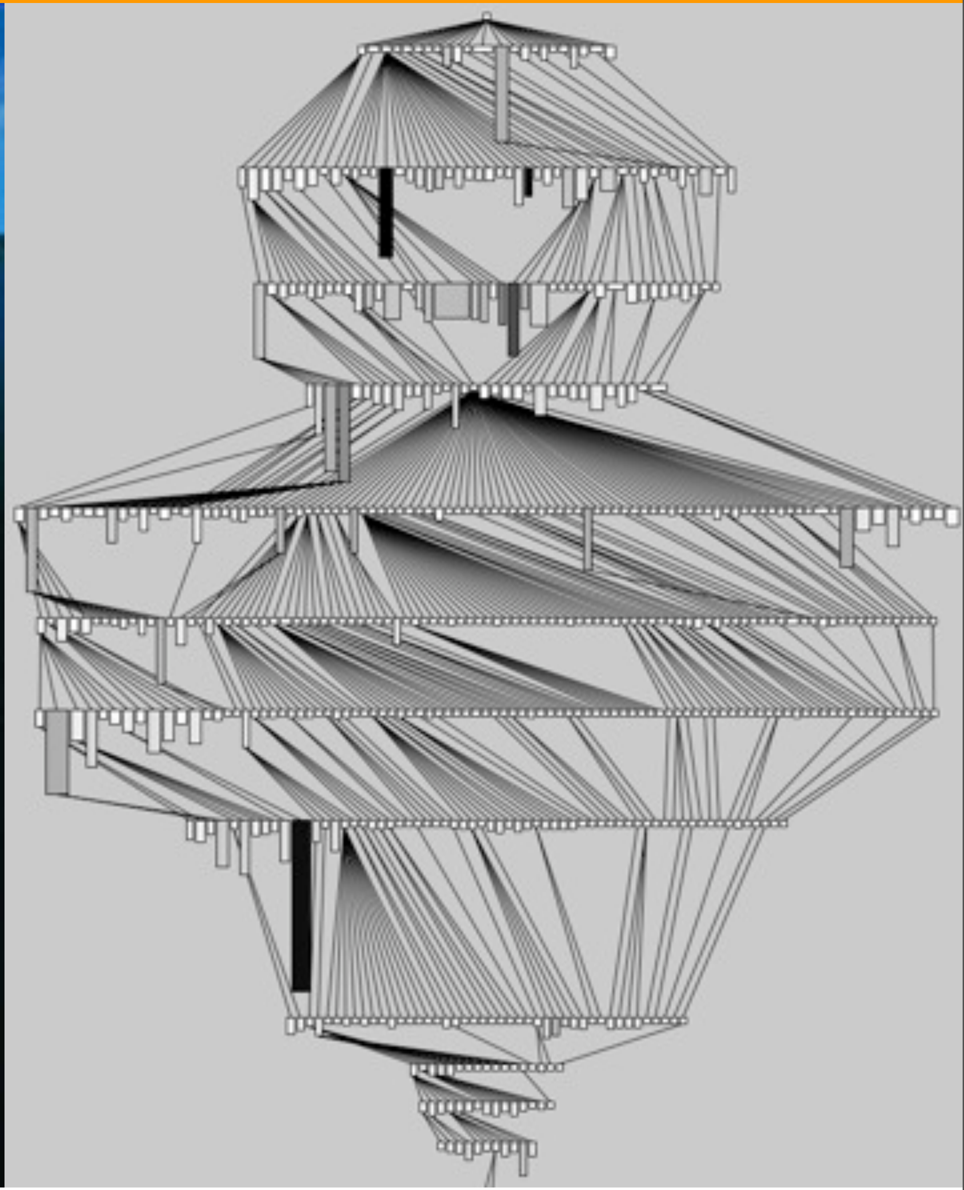
…software is intangible, having no physical shape or size…

…software is intangible, having no physical shape or size…

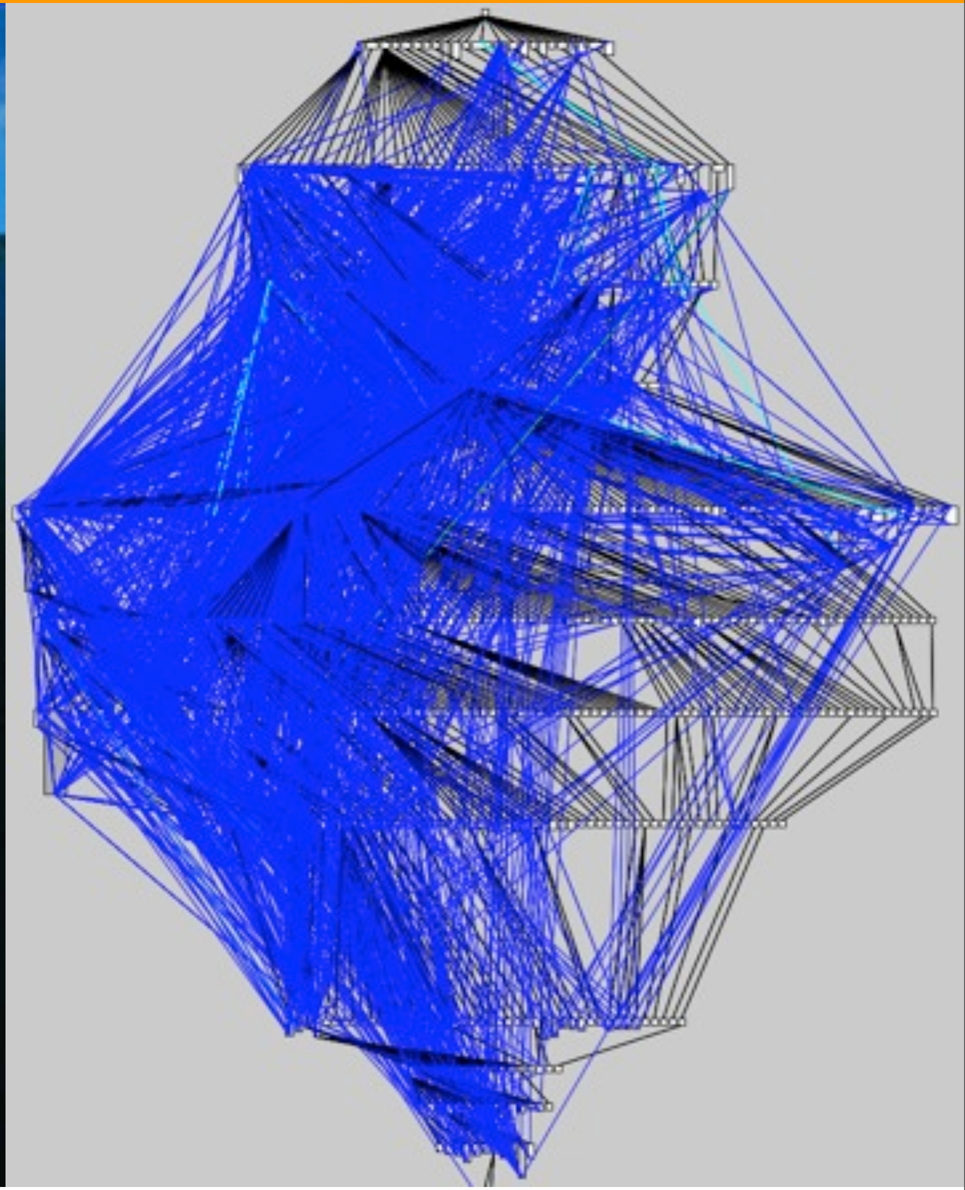
…software is intangible, having no physical shape or size…



…software is intangible, having no physical shape or size…



…software is intangible, having no physical shape or size…

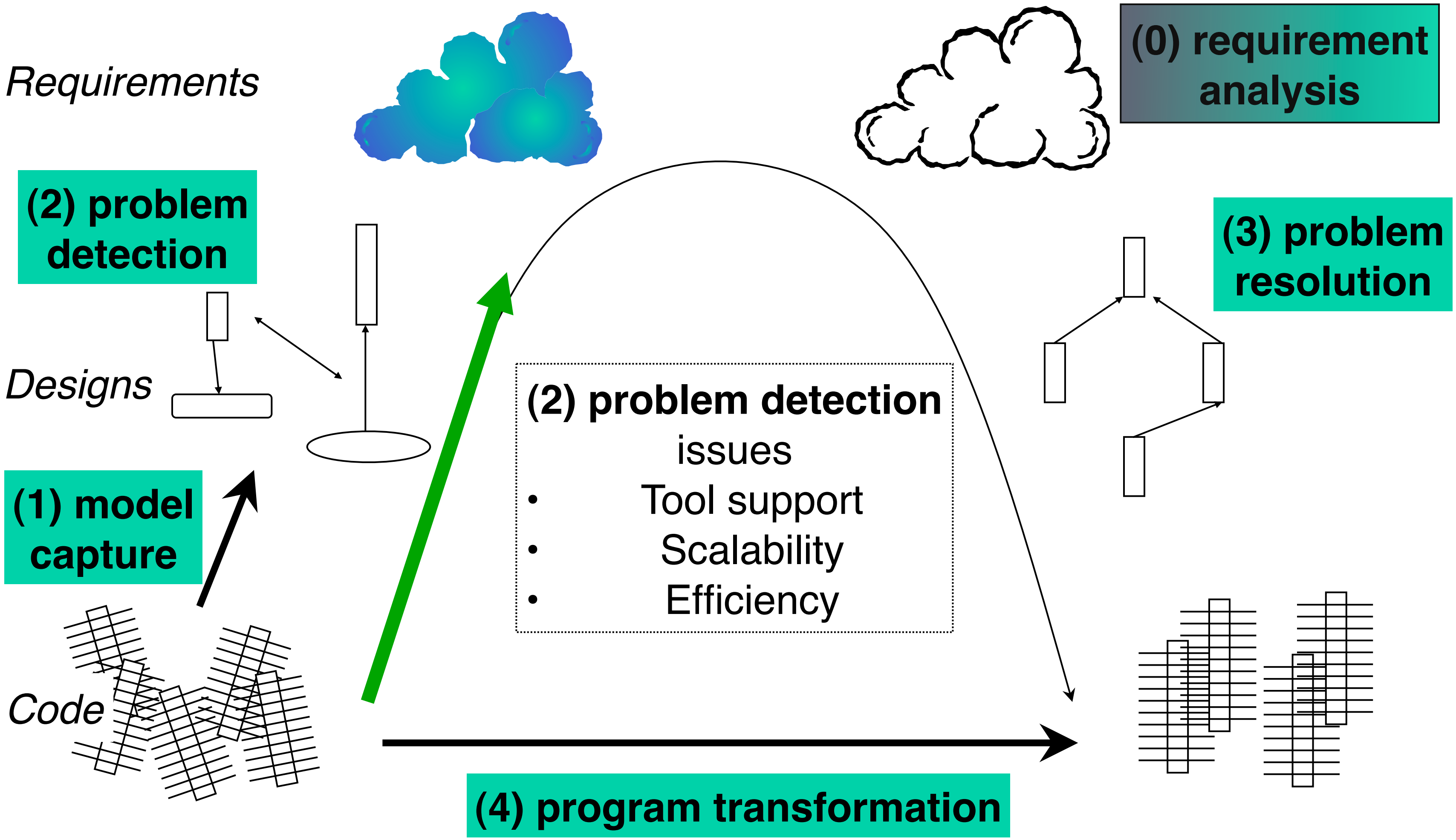


Software Visualization in Context

There are many good-looking visualization techniques, but..when it comes to software maintenance & evolution, there are several problems:

- Scalability
- Information Retrieval
- What to visualize
- How to visualize
- Reengineering context constraints
  - Limited time
  - Limited resources

The Reengineering Life-cycle



Program Visualization

"The visualization of the actual program code or data structures in either static or dynamic form" [Price, Baecker and Small, "Introduction to Software Visualization"]

Static Visualization and/or Dynamic Visualization

Overall Goal: Generate views of a system to understand it

Complex Problem Domain/Research Area

- Visual Aspects
  - Efficient use of space, overplotting problems, layout issues, HCI issues, GUI issues, lack of conventions (colors, shapes, etc.)
- Software Aspects
  - Level of granularity?
    - Complete systems, subsystems, modules, classes, hierarchies,...
  - When to apply?
    - First contact with an 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

- In other words: information obtained at compile-time

Depends on the programming language and paradigm:

- Object-Oriented PL:
  - classes, methods, attributes, inheritance, …
- Procedural PL:
  - procedures, invocations, …
- Functional PL:
  - functions, function calls, …

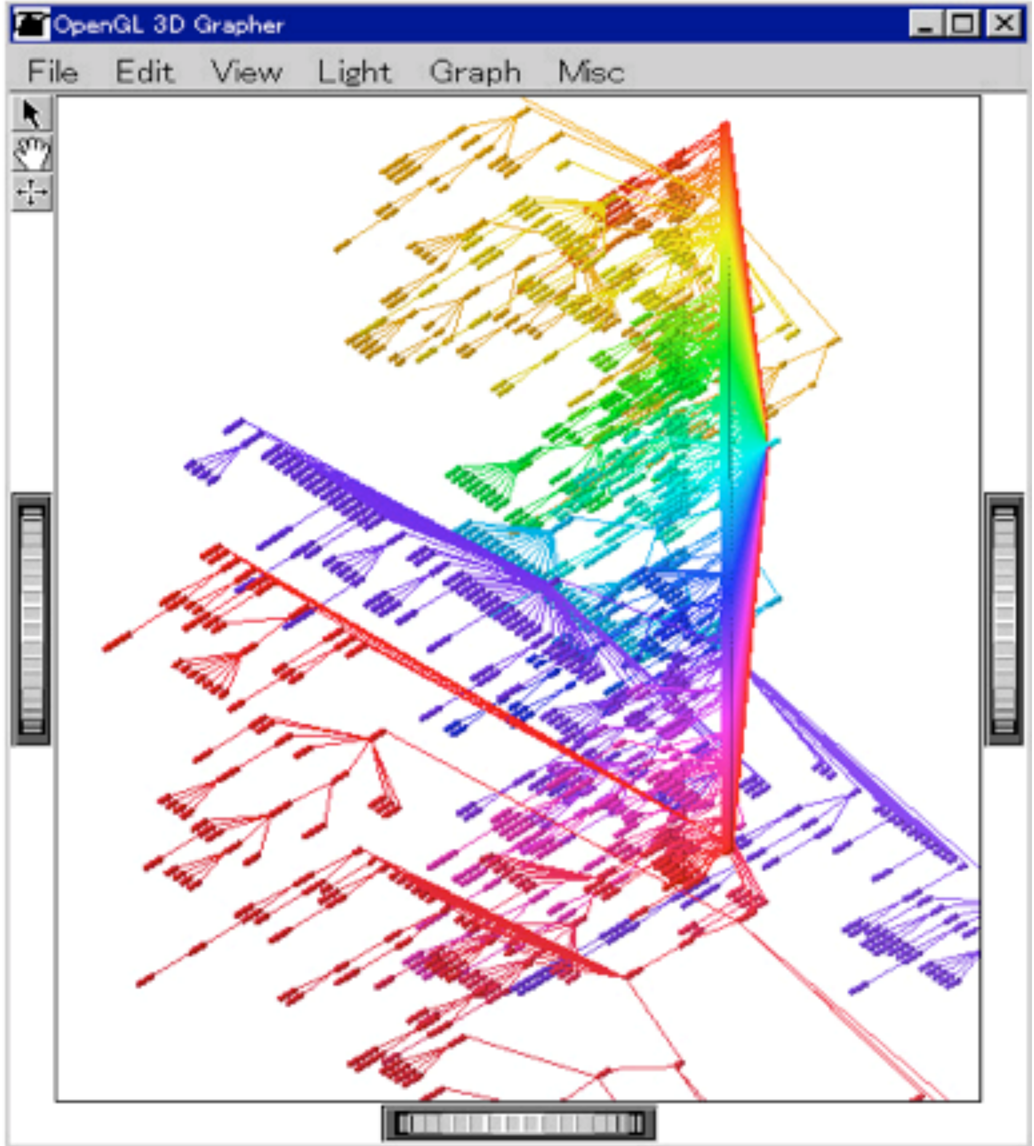
Example 1: Class Hierarchies

Jun/OpenGL

The Smalltalk Class Hierarchy

Problems:

- Colors are meaningless
- Visual Overload
- Navigation



Example 2: Tree Maps

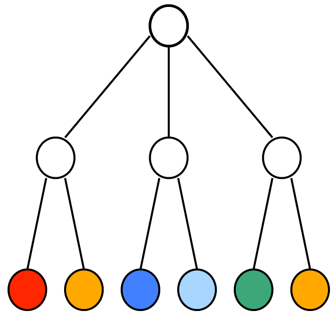
Pros

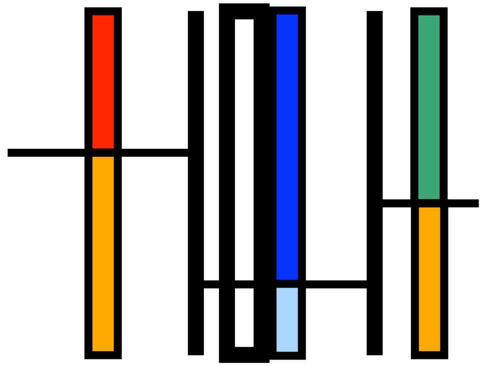
- 100% screen
- Large data
- Scales well

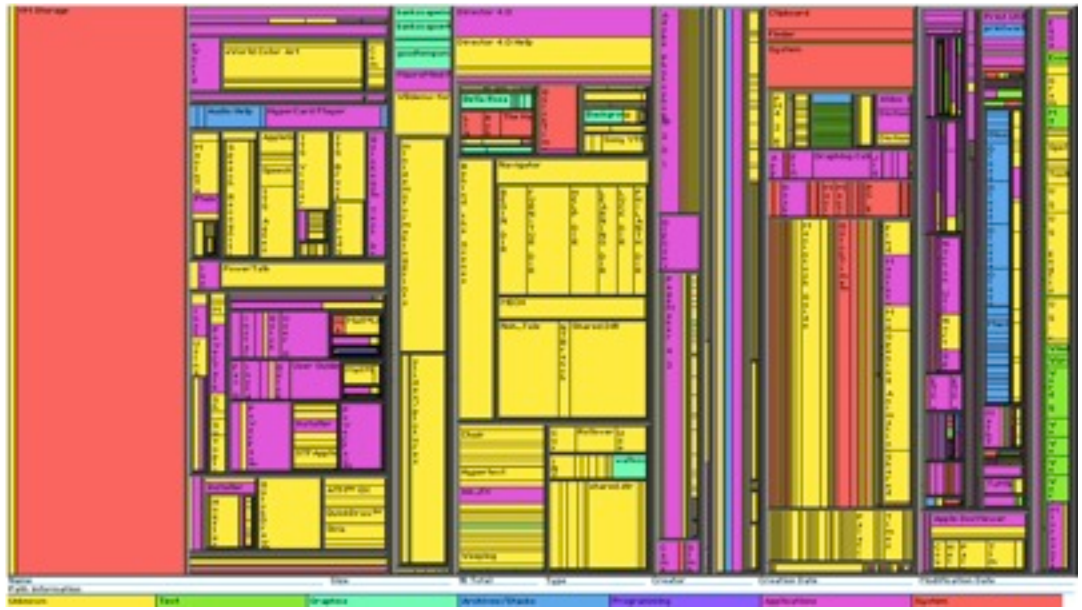
Cons

- Boundaries
- Cluttered display
- Interpretation
- Leaves only

Useful for the display of hard disks







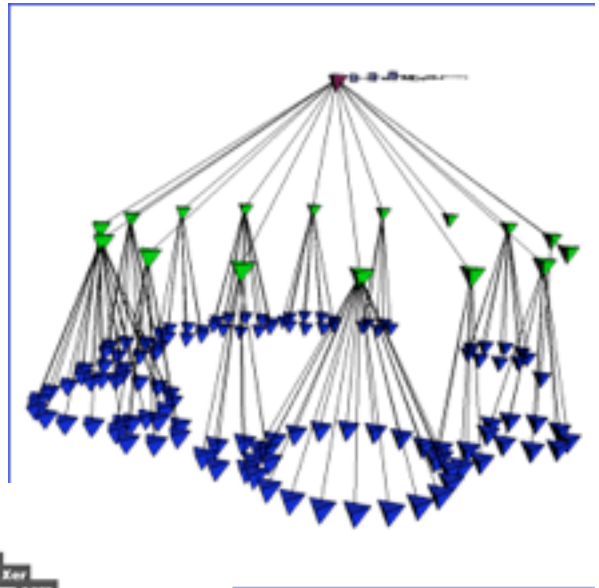
Examples 3 & 4

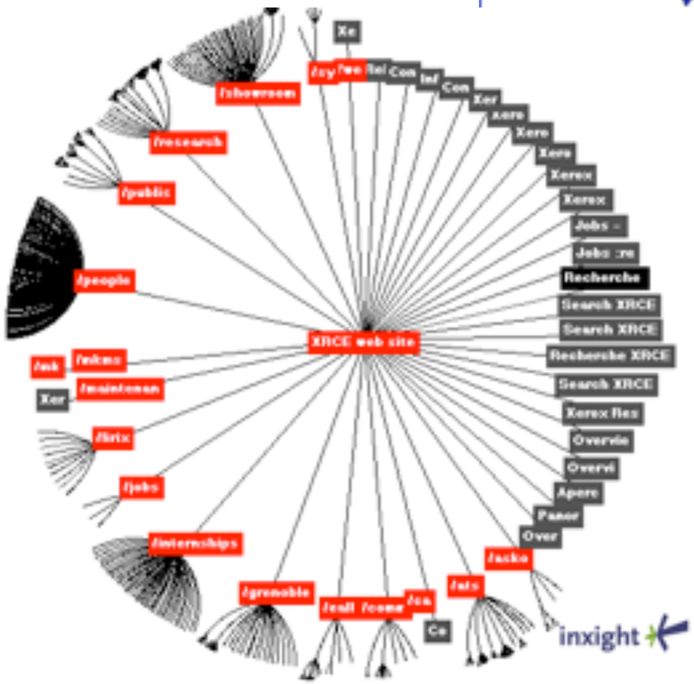
Euclidean cones

- Pros:
  - More info than 2D
- Cons:
  - Lack of depth
  - Navigation

Hyperbolic trees

- Pros:
  - Good focus
  - Dynamic
- Cons:
  - Copyright





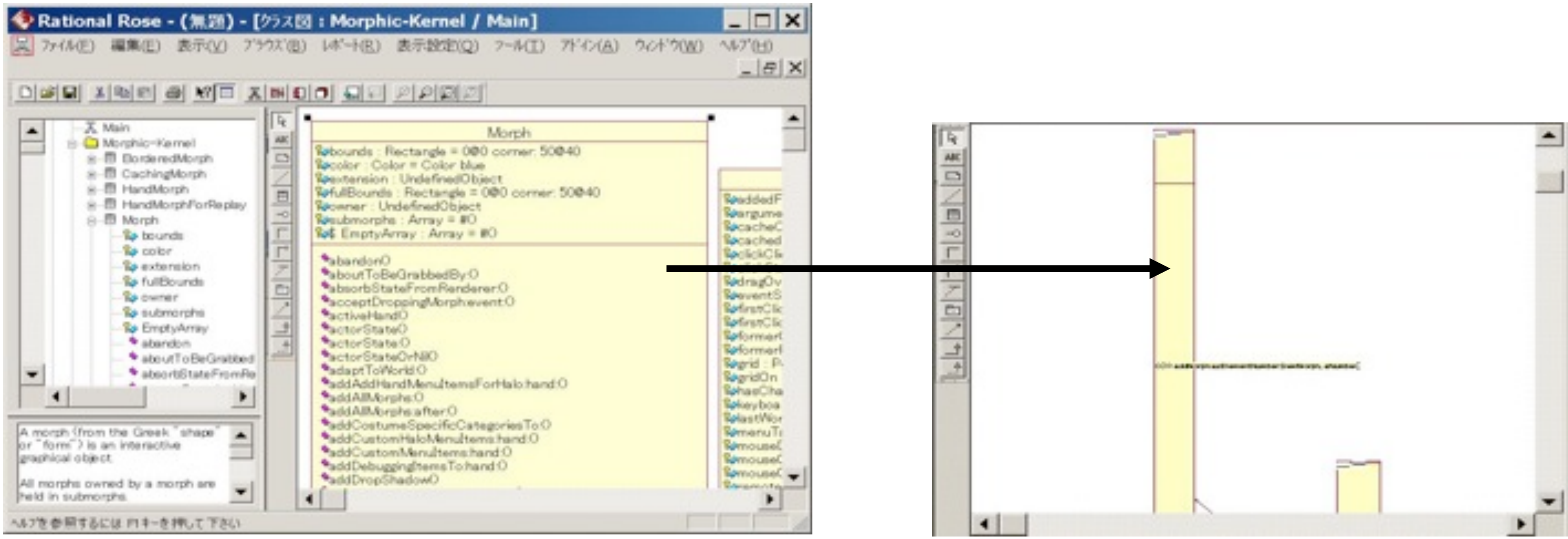
Example 5: UML and derivates

Pros

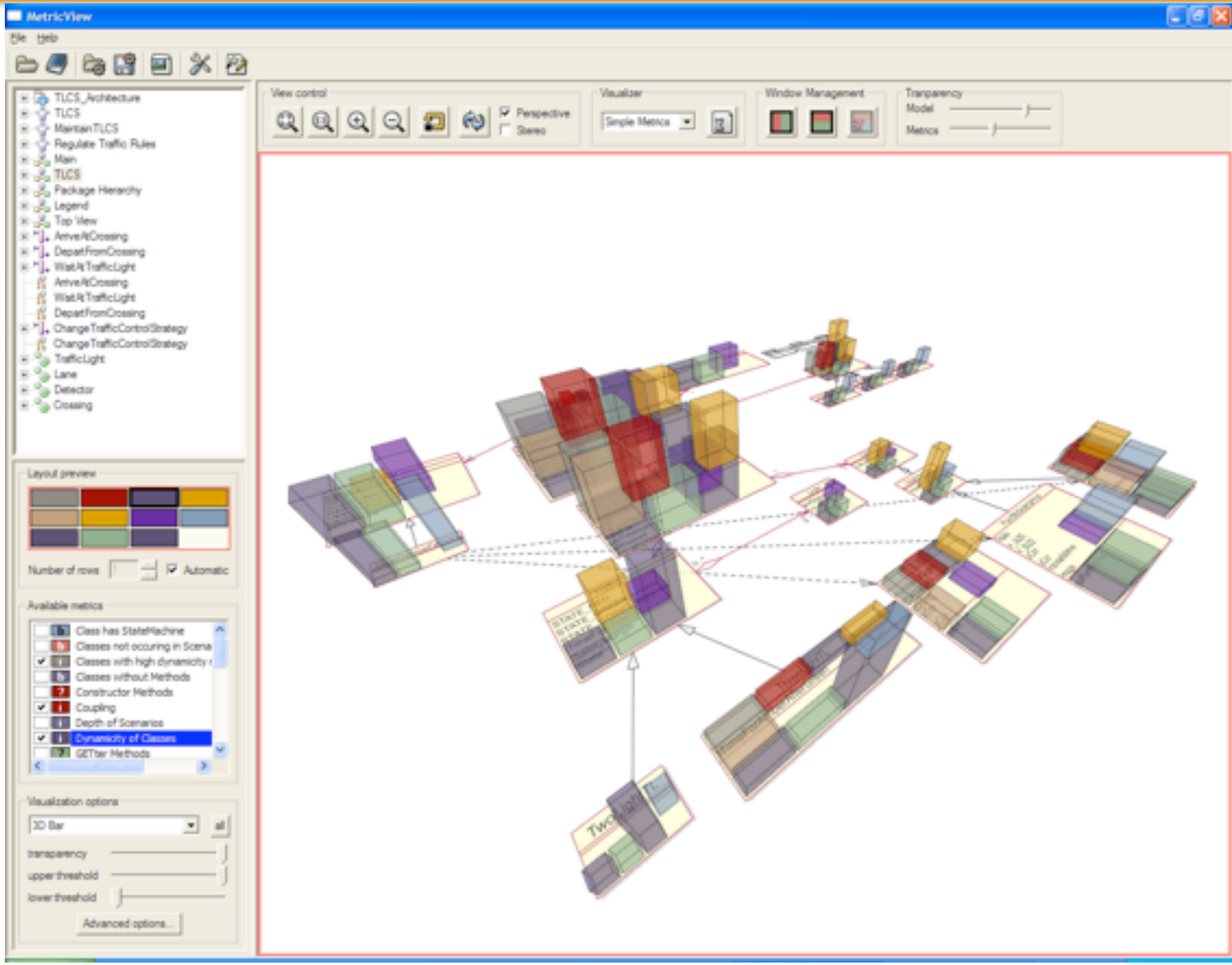
- OO concepts
- Works very well for small parts

Cons

- Lack of scalability
- Requires tool support
- Requires mapping rules to reduce noise
- Hardly extensible



Example 6: UML goes 3D



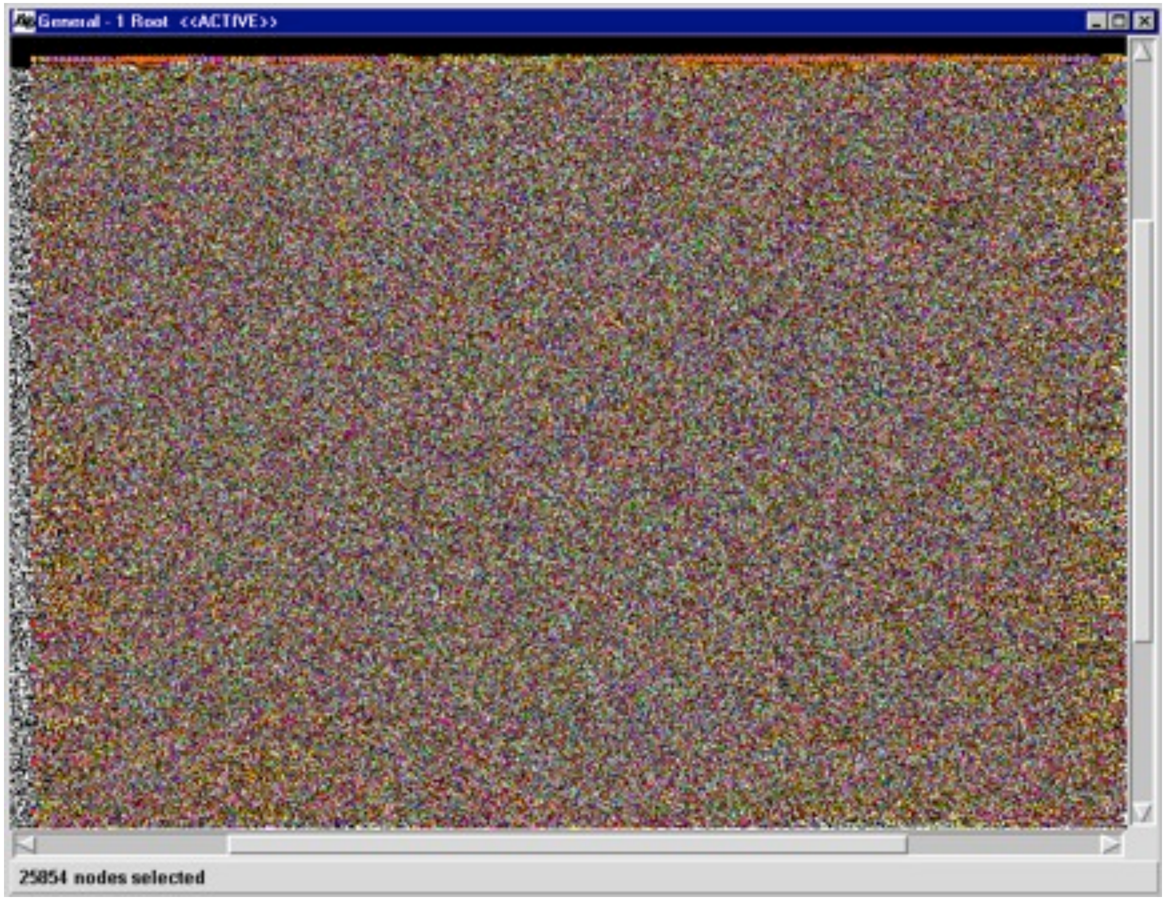
Example 6a: Rigi

Scalability problem

Entity-Relationship visualization

Problems:

- Filtering
- Navigation



Example 6b: Rigi

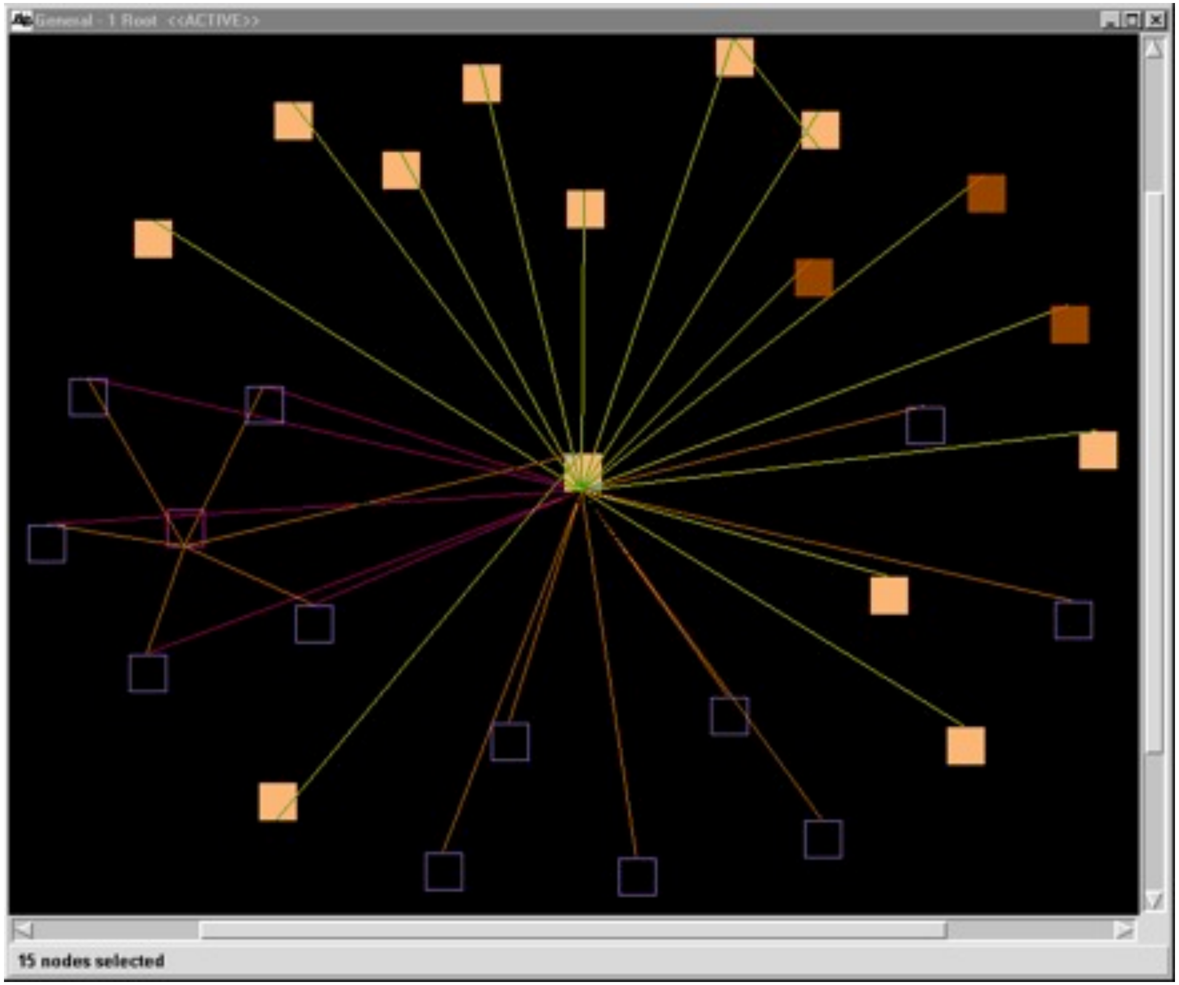
Entities can be grouped

Pros:

- Scales well
- Applicable in other domains

Cons:

- Not enough code semantics



Static SV: Evaluation

Pros

- Intuitive approaches
- Aesthetically pleasing results

Cons

- Several approaches are orthogonal to each other
- Too easy to produce meaningless results
- Scaling up is sometimes possible, but at the expense of semantics

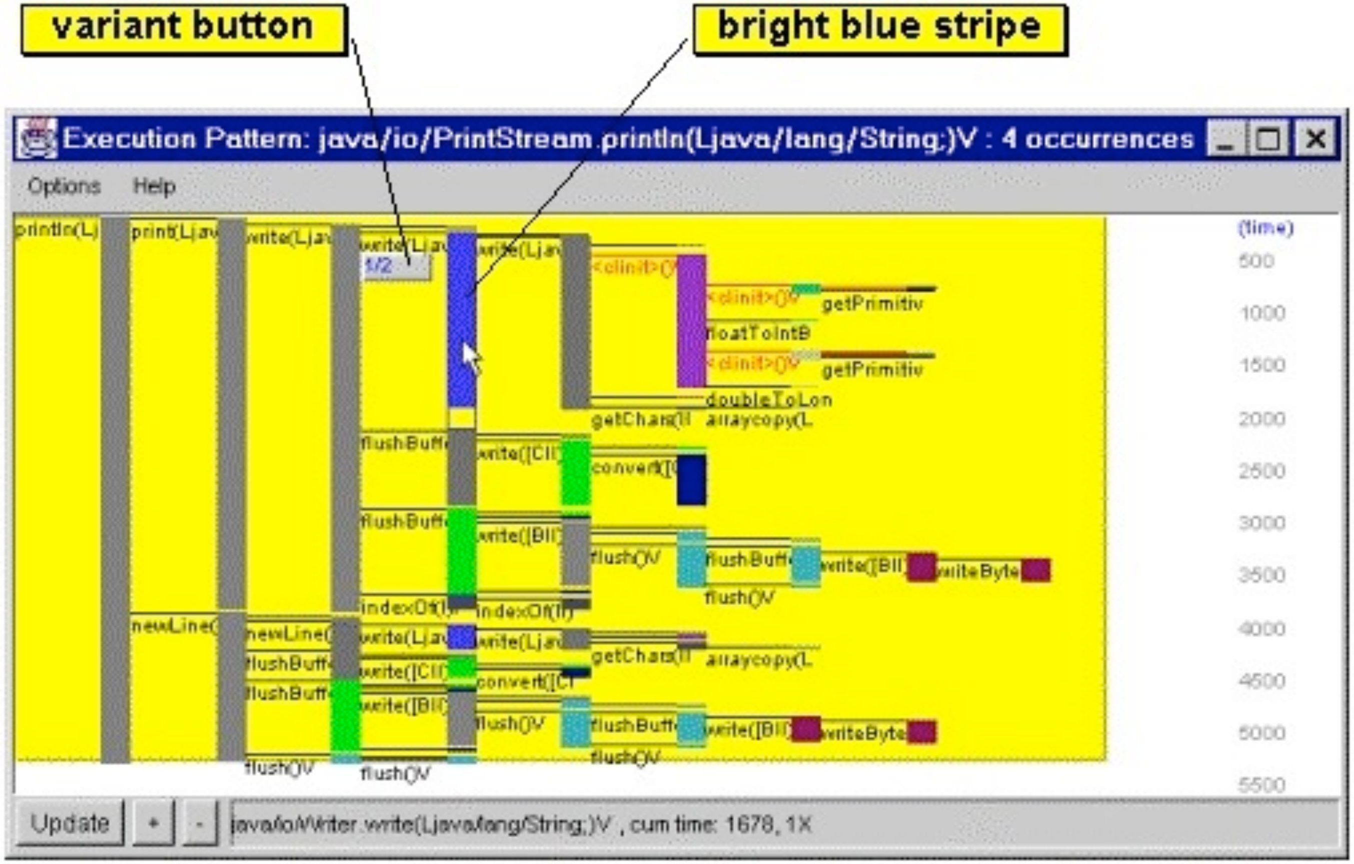
Dynamic Code Visualization

Visualization of dynamic behavior 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



Example 2: Inter-class call matrix

- Simple
- Scales quite well
- Reproducible

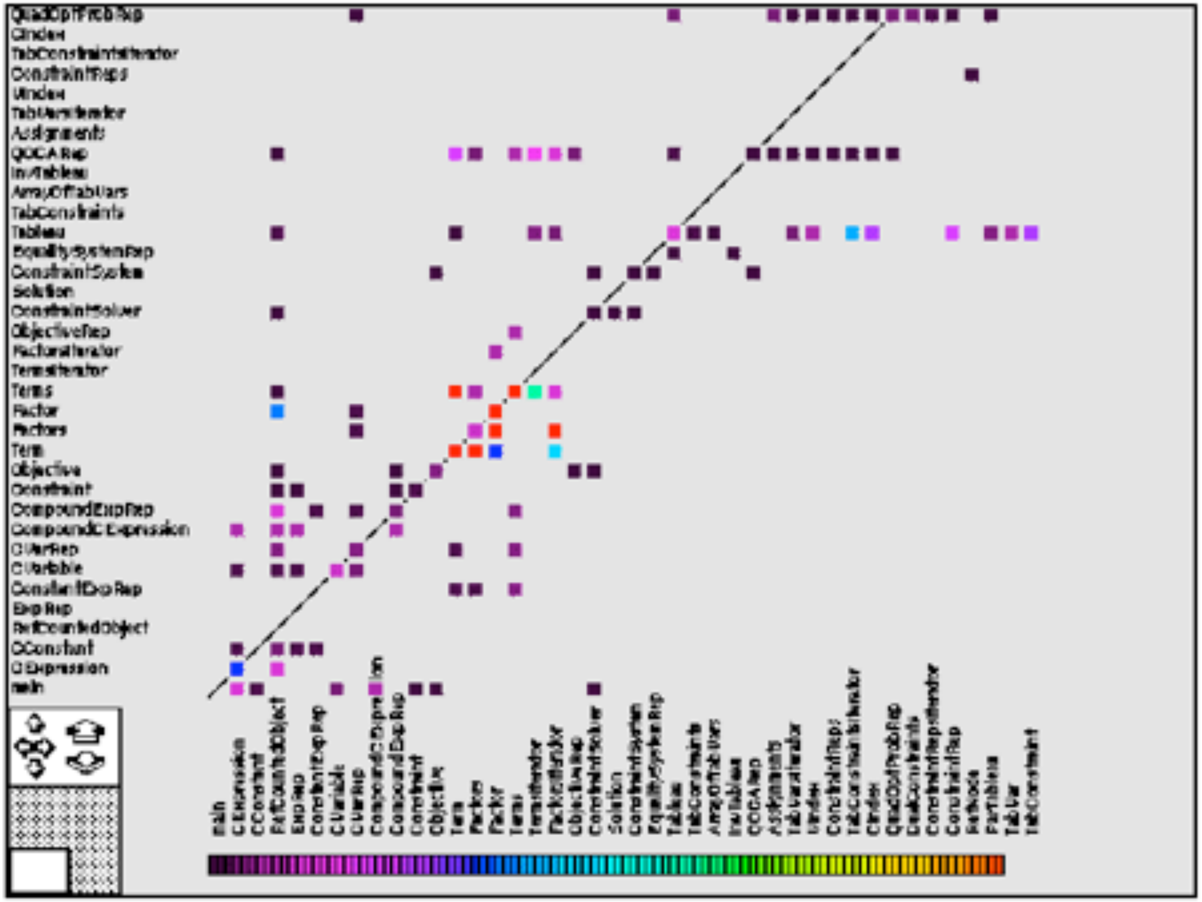


Figure 6: Inter-class call matrix

Dynamic SV: Evaluation

Code instrumentation problem

- Logging, Extended VMs, Method Wrapping

Scalability problem

- Traces quickly become very big

Completeness problem

- Scenario driven

Pros:

- Good for fine-tuning, problem detection

Cons:

- Tool support crucial
- Lack of abstraction without tool support

III. Software Quality Assessment

Visualization and Metrics

Why is visualization important at all?

Is it actually useful?

- No, visualization is only a means, not the end…
- Yes, visualization is only a means, not the end!!!

The question is: "What is the end?"

- We want to understand systems…

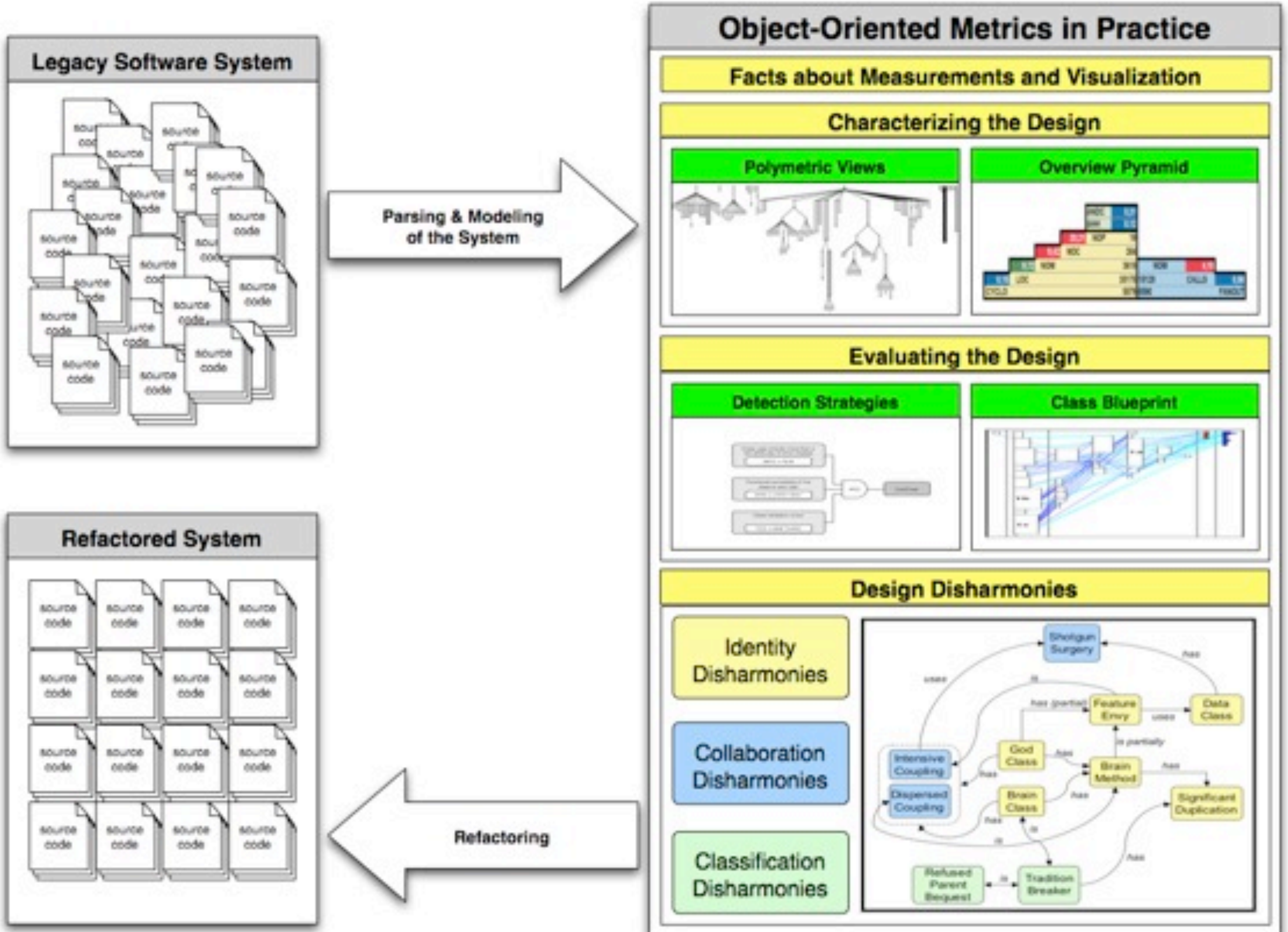
Question 2: "Why are visualizations not used more?"

- The "context" does not permit heavy-weight approaches

This is where reality kicks in, i.e., what is actually useful in practice?

- Lightweight approaches!

OO Metrics in a Nutshell



Metrics

What is a metric?

The mapping of a particular characteristic of a measured entity to a numerical value

Why is it useful to measure?

To keep control of…complexity

Advantages

Ability to quantify aspects of quality

Possibility to automate the “measurements” of systems

Drawbacks

Numbers are just numbers: don’t trust them

Metrics capture only fine-grained symptoms, not causes of design problems

Hard for developers to deal with them

Inflation of measurements

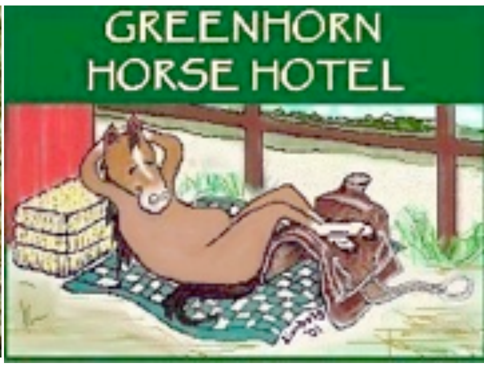
What is interesting for a developer/designer?

Understanding the Code

- Code outsourcing
- New Hires

Evaluating & Improving the Code

- Portable Design
- Flexible Design



Understanding the Code

"Yesterday I met a system…"

- How many lines of code? --> 35'000 LOC
- How many functions/methods? --> 3'600 NOM
- How many classes? --> 380 NOC
- etc…

Is it "normal" to have a system of…

- 380 classes with 3'600 methods?
- 3600 methods with 35'000 lines of code?

What is "normal"? What about coupling or cohesion?

- We need means of comparison: proportions are important
- Collect more relevant numbers: the more the better…or not?

How can we characterize the design of a system?

Characterizing the Design of a System

How do you describe a system?

- Lines of code? Classes? Methods? Megabytes? Files?

Characterizing a System with few metrics is difficult because of

- Unbalanced Characterization
  - How “object-oriented” is a 500-class/25 kLOC system?
- Misused Metrics
  - What can I say about a 100 kLOC system?
- Uncorrelated Metrics
  - 100-class/20kLOC vs. 100-class/1MLOC
- Missing Reference Points
  - What is “normal”?

How do we characterize design?

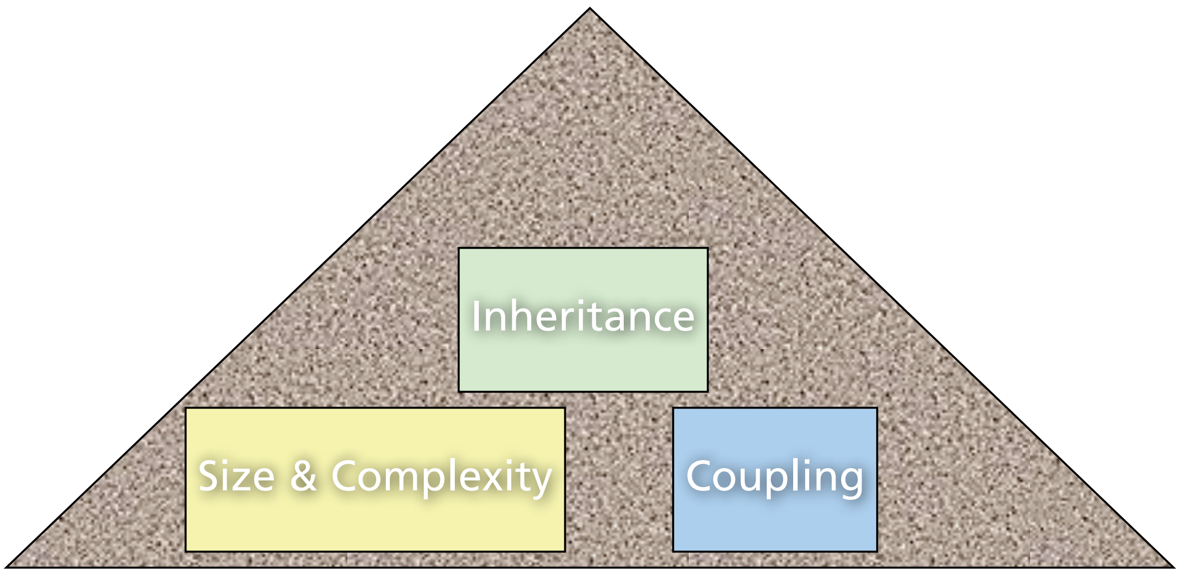
- The Overview Pyramid
- Polymetric Views

The Metrics Pyramid

The Overview Pyramid

A metrics-based means to both describe and characterize the structure of an object-oriented system by quantifying its complexity, coupling and usage of inheritance

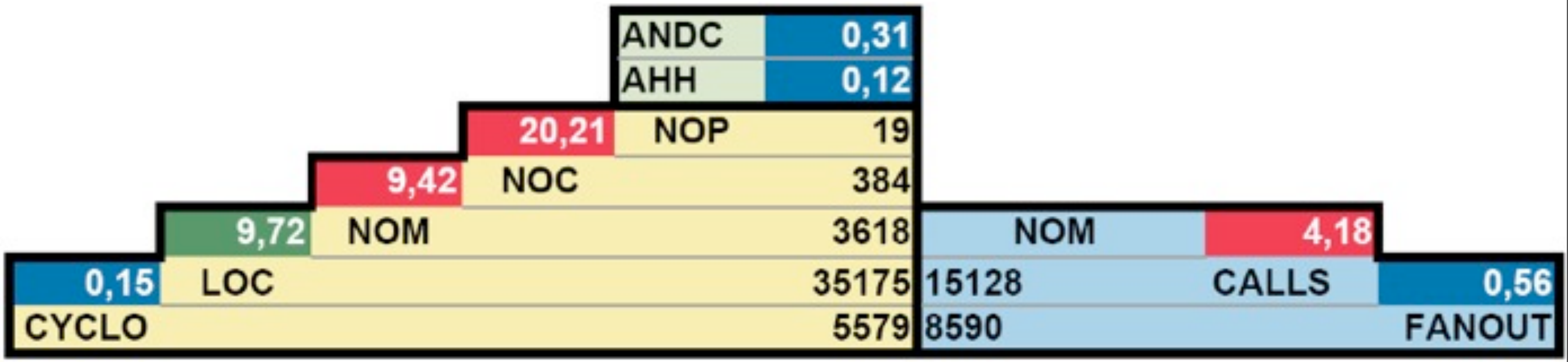
Measuring these 3 aspects at system level provides a comprehensive characterization of an entire system



The Overview Pyramid in Detail

The left side: System Size & Complexity

- Direct metrics: NOP, NOC, NOM, LOC, CYCLO
- Derived metrics: NOC/P, NOM/C, LOC/M, CYCLO/LOC



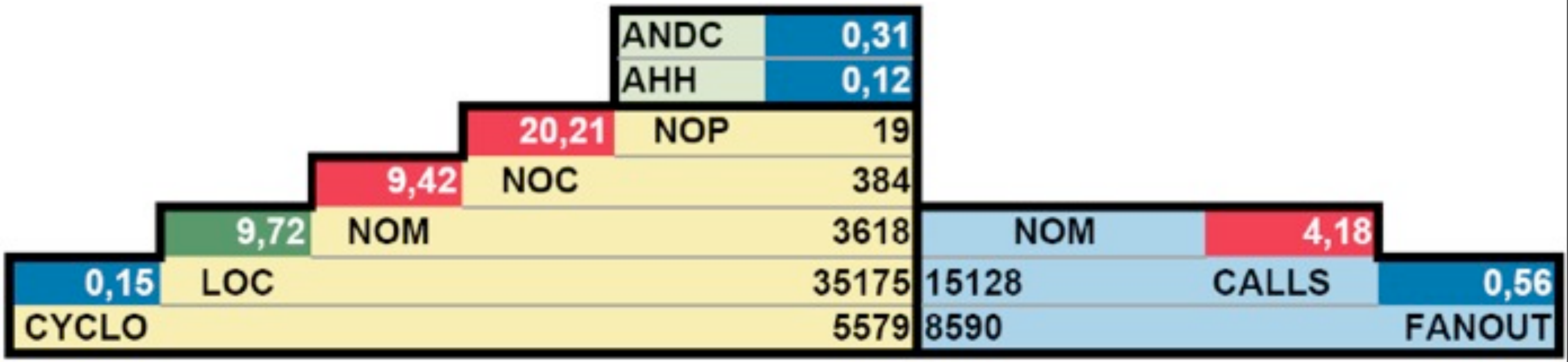
The Overview Pyramid in Detail

The left side: System Size & Complexity

- Direct metrics: NOP, NOC, NOM, LOC, CYCLO
- Derived metrics: NOC/P, NOM/C, LOC/M, CYCLO/LOC

The right side: System Coupling

- Direct metrics: CALLS, FANOUT
- Derived metrics: CALLS/M, FANOUT/CALL



The Overview Pyramid in Detail

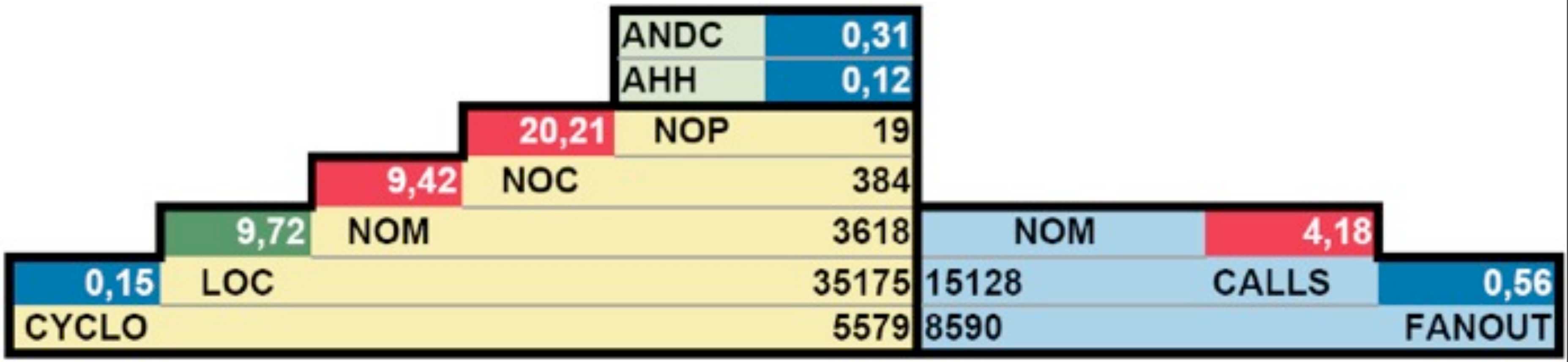
The left side: System Size & Complexity

- Direct metrics: NOP, NOC, NOM, LOC, CYCLO
- Derived metrics: NOC/P, NOM/C, LOC/M, CYCLO/LOC

The right side: System Coupling

- Direct metrics: CALLS, FANOUT
- Derived metrics: CALLS/M, FANOUT/CALL

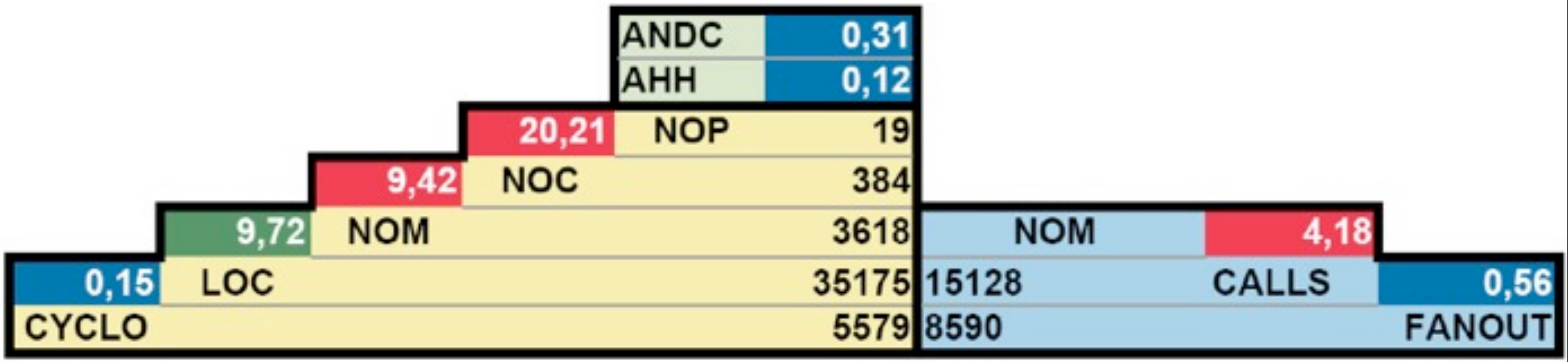
-



Interpreting the Overview Pyramid

The pyramid characterizes a system in terms of size&complexity, coupling, and inheritance; based on 8 computed proportions:

- They are independent of the size of the system!
- This enables an objective assessment…
  - Wait a second…objective? Where is the reference point?



Putting things in a real-world context

We measured 80+ systems written in Java and C++

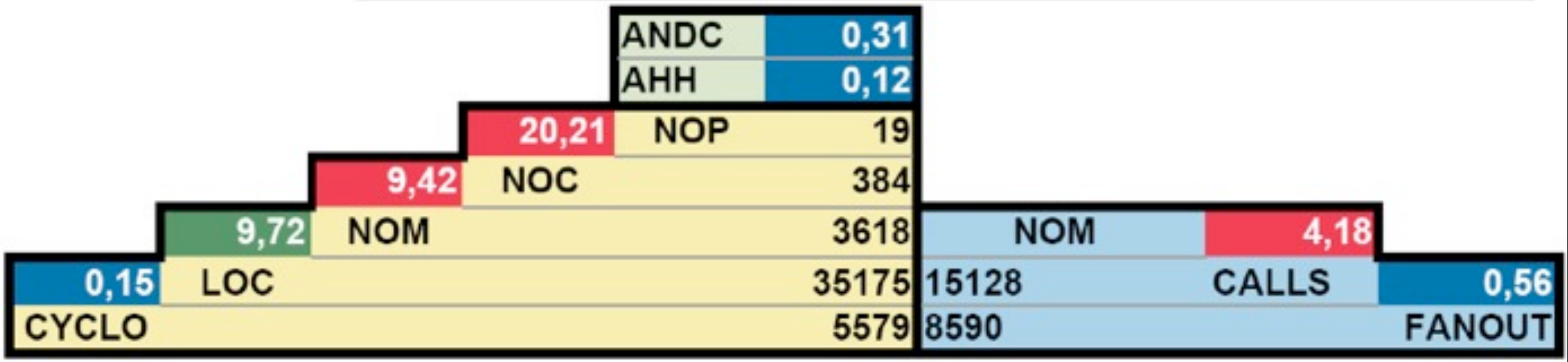
Based on the obtained measurements we can now statistically assess the design of a system

High

Average

Low

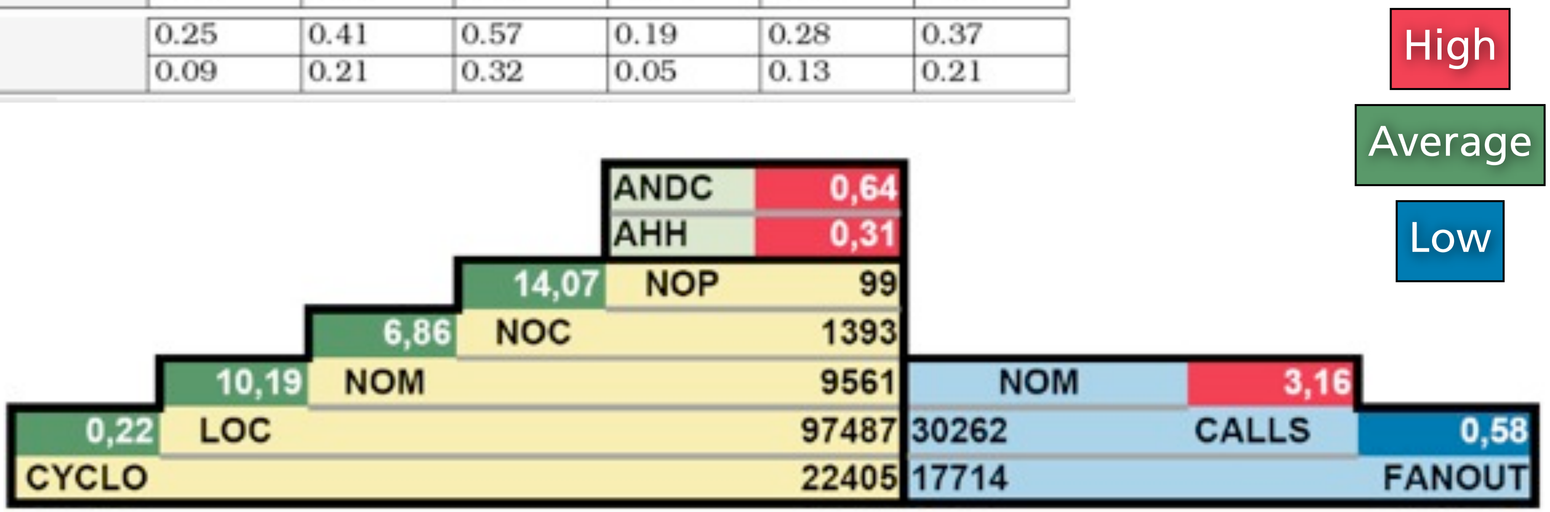
	Java			C++		
Metric	Low	Average	High	Low	Average	High
CYCLO/Line of code	0.16	0.20	0.24	0.20	0.25	0.30
LOC/Operation	7	10	13	5	10	16
NOM/Class	4	7	10	4	9	15
NOC /Package	6	17	26	3	19	35
CALLS/Operation	2.01	2.62	3.2	1.17	1.58	2
FANOUT /Call	0.56	0.62	0.68	0.20	0.34	0.48
ANDC	0.25	0.41	0.57	0.19	0.28	0.37
AHH	0.09	0.21	0.32	0.05	0.13	0.21



Overview Pyramid Example: ArgoUML

Metric	Value	Remarks
No. of Lines of Code	223,068	including comments
No. of Source Files	1,209	*.java files
No. of Packages	99	–
No. of Classes	1,393	including 140 inner classes
No. of Methods	9,561	including accessor methods
No. of Attributes	3,358	all variables including static and local variables

	Java			C++		
Metric	Low	Average	High	Low	Average	High
CYCLO/Line of code	0.16	0.20	0.24	0.20	0.25	0.30
LOC/Operation	7	10	13	5	10	16
NOM/Class	4	7	10	4	9	15
NOC /Package	6	17	26	3	19	35
CALLS/Operation	2.01	2.62	3.2	1.17	1.58	2
FANOUT /Call	0.56	0.62	0.68	0.20	0.34	0.48
ANDC	0.25	0.41	0.57	0.19	0.28	0.37
AHH	0.09	0.21	0.32	0.05	0.13	0.21



See(k)ing to understand

The Overview Pyramid allows us to characterize the design of a system

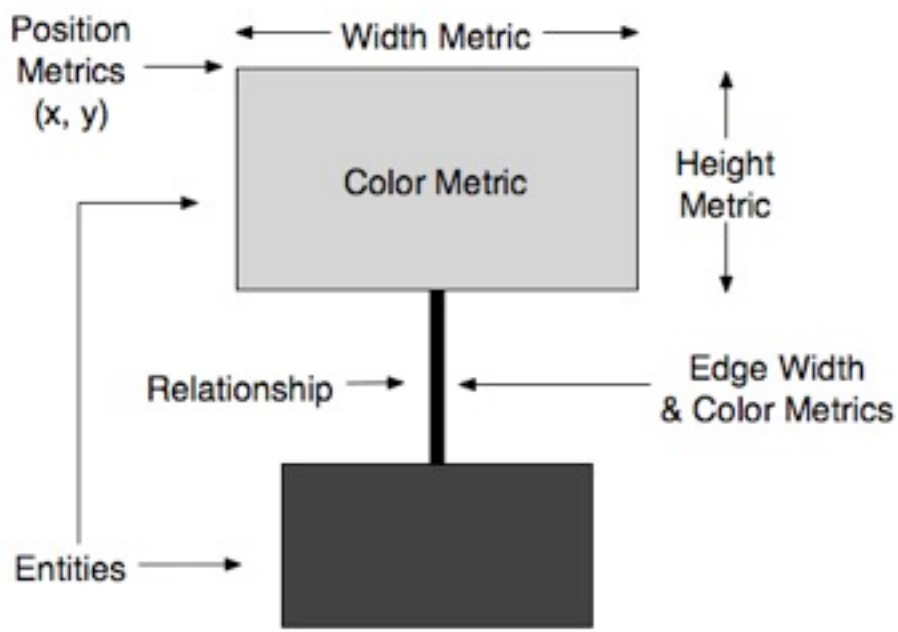
But…we need to see what we are talking about

Polymetric Views

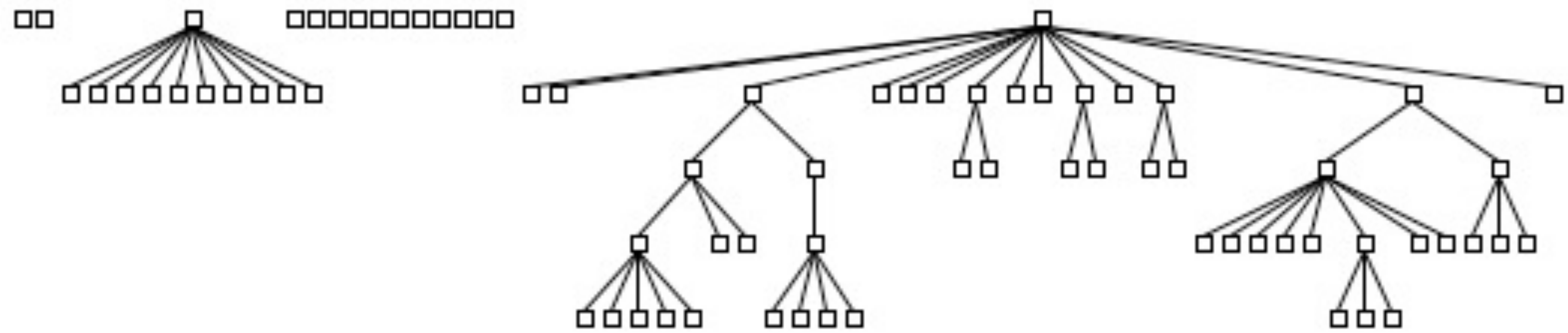
Polymetric Views

Metrics-enriched visualizations of software entities and their relationships; useful for

- Rendering numbers in a simple, yet effective and highly condensed way
- Visually characterizing a system in its own context

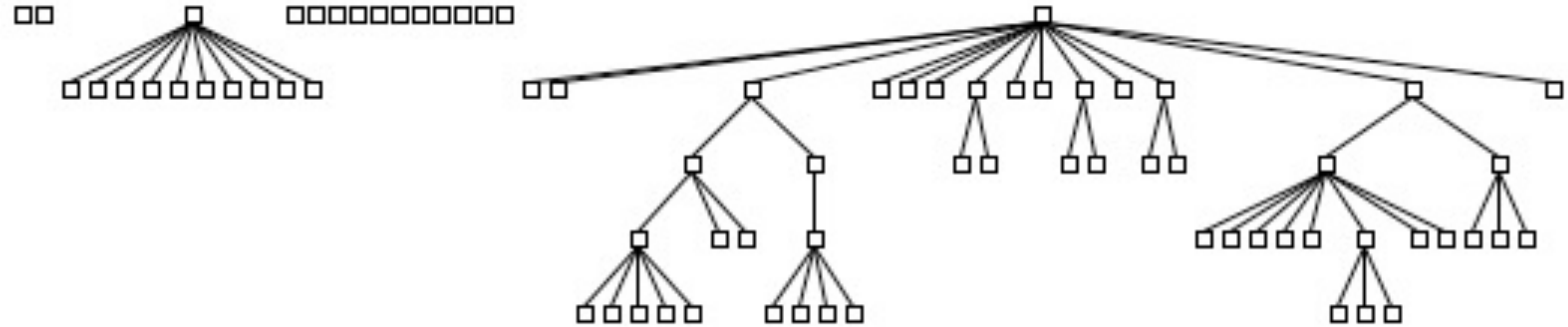


The Polymetric View - Example



Nodes = Classes
Edges = Inheritance Relationships

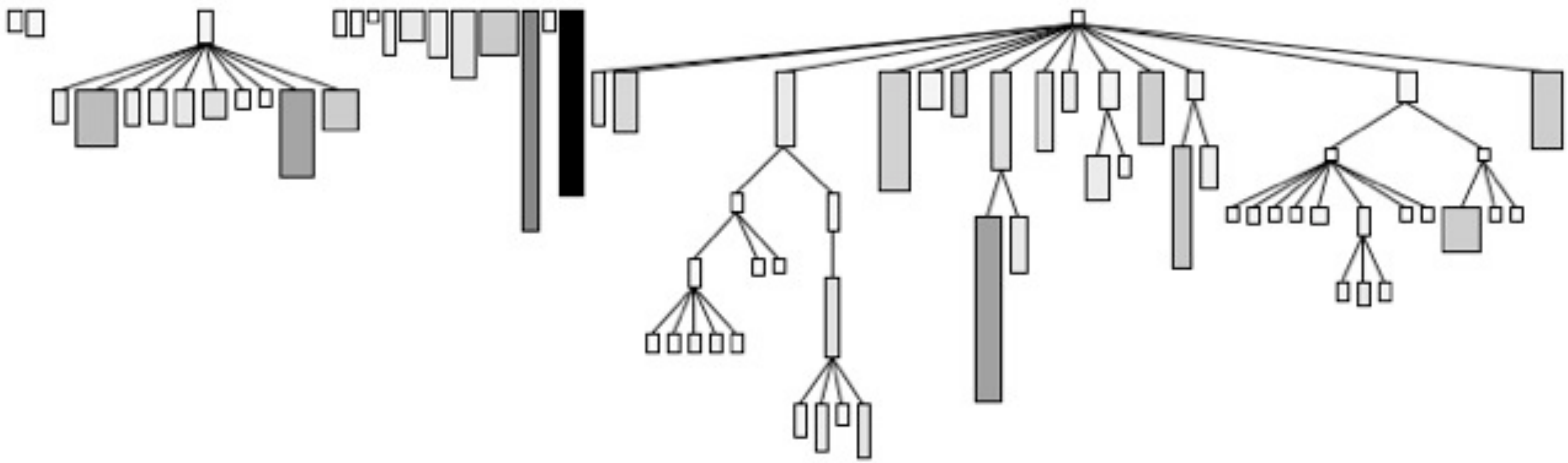
The Polymetric View - Example



Nodes =	Classes
Edges =	Inheritance Relationships

Width =	Number of Attributes
Height =	Number of Methods
Color =	Number of Lines of Code

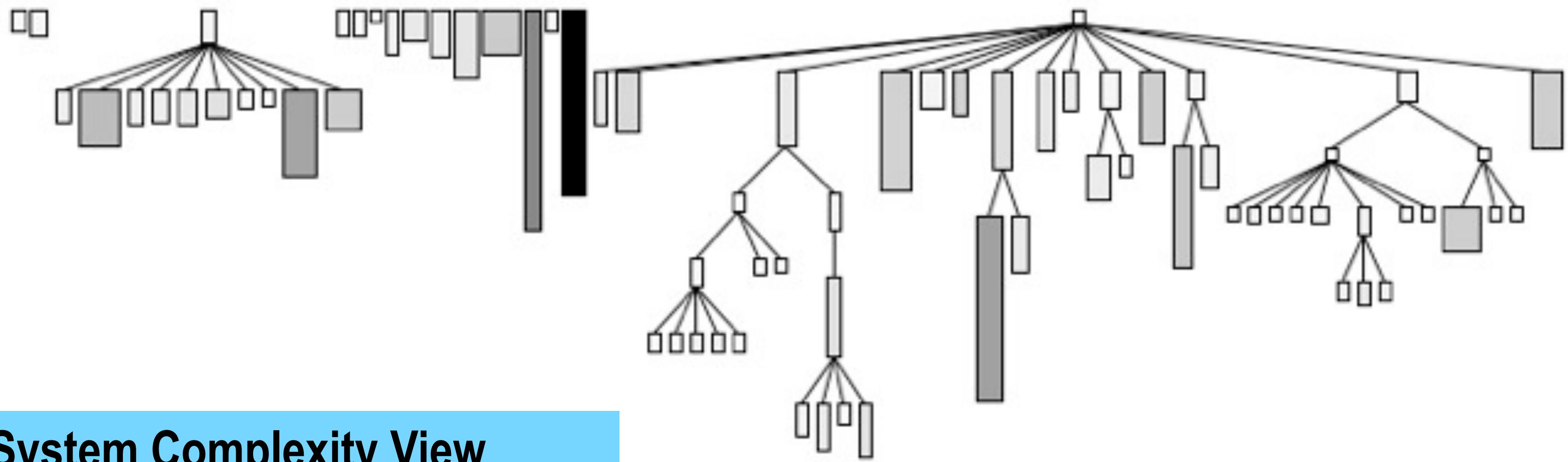
The Polymetric View - Example



Nodes = Classes
Edges = Inheritance Relationships

Width = Number of Attributes
Height = Number of Methods
Color = Number of Lines of Code

The Polymetric View - Example

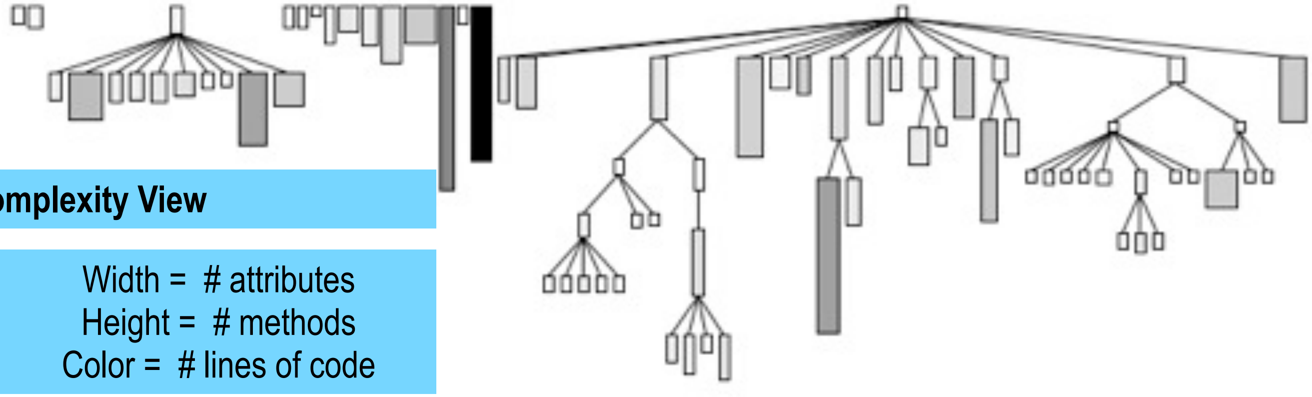


System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

Width = Number of Attributes
Height = Number of Methods
Color = Number of Lines of Code

The Polymetric View - Example (II)

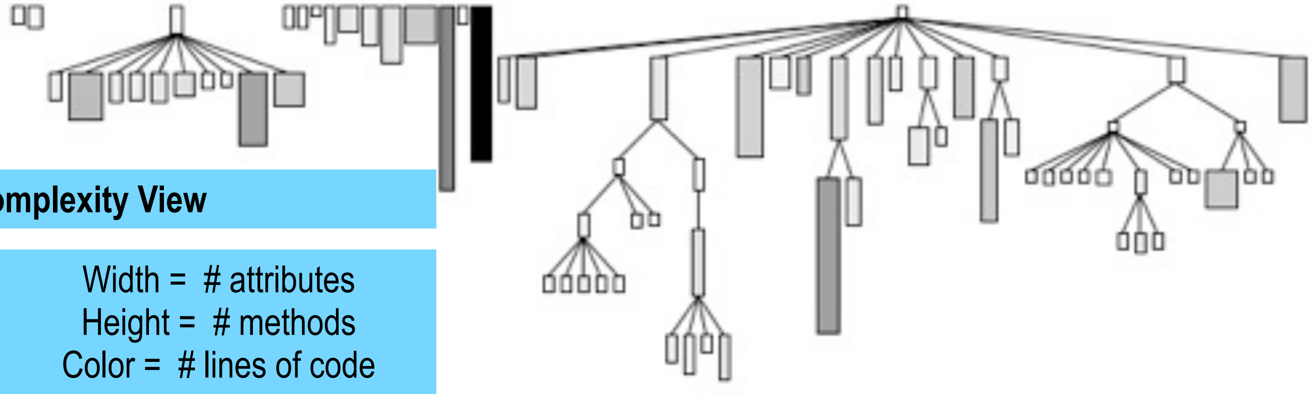


System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

Width = # attributes
Height = # methods
Color = # lines of code

The Polymetric View - Example (II)

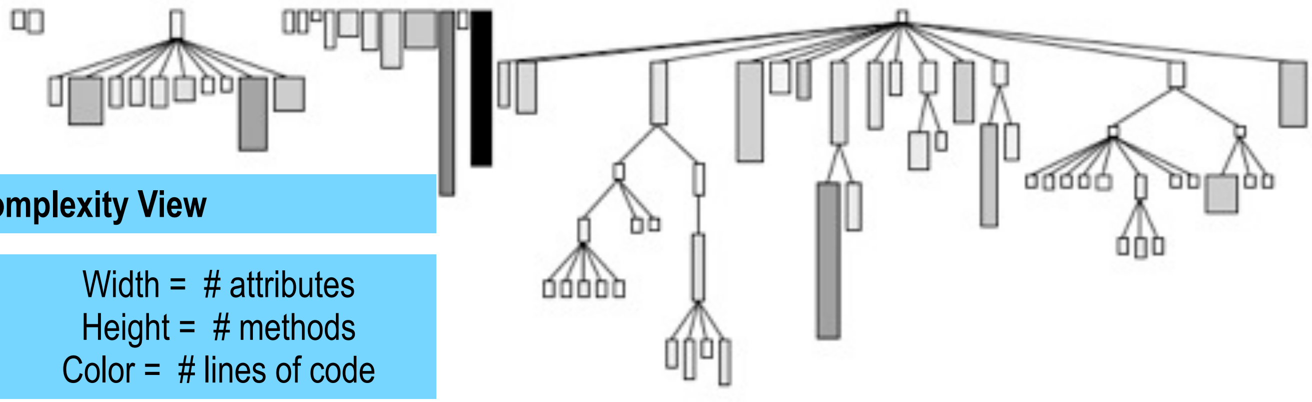


System Complexity View

Nodes = Classes	Width = # attributes
Edges = Inheritance Relationships	Height = # methods
	Color = # lines of code

Reverse engineering goals

The Polymetric View - Example (II)



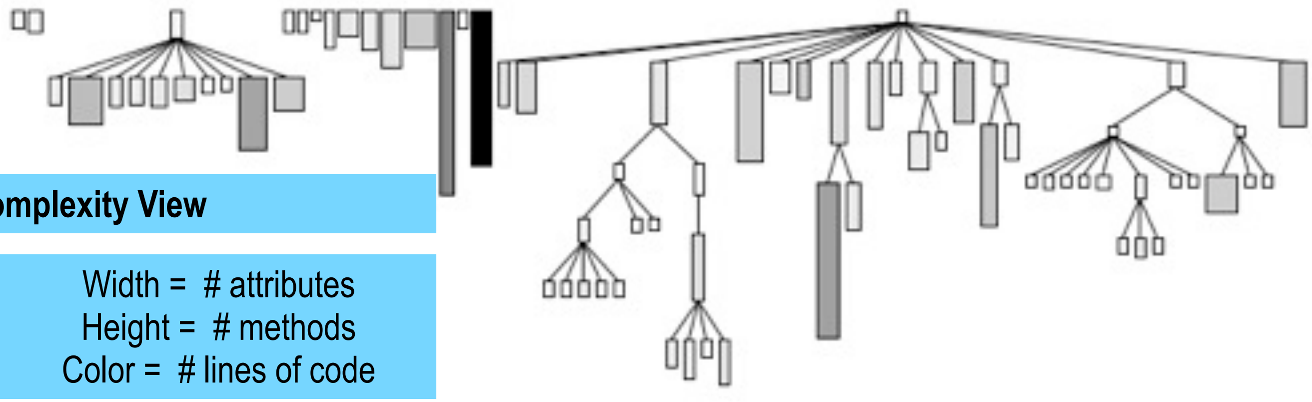
System Complexity View

Nodes = Classes
Edges = Inheritance Relationships
	Width = # attributes
	Height = # methods
	Color = # lines of code

Reverse engineering goals

- Get an impression (build a first raw mental model) of the system, know the size, structure, and complexity of the system in terms of classes and inheritance hierarchies
- Locate important (domain model) hierarchies, see if there are any deep, nested hierarchies
- Locate large classes (standalone, within inheritance hierarchy), locate stateful classes and classes with behavior

The Polymetric View - Example (II)



System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

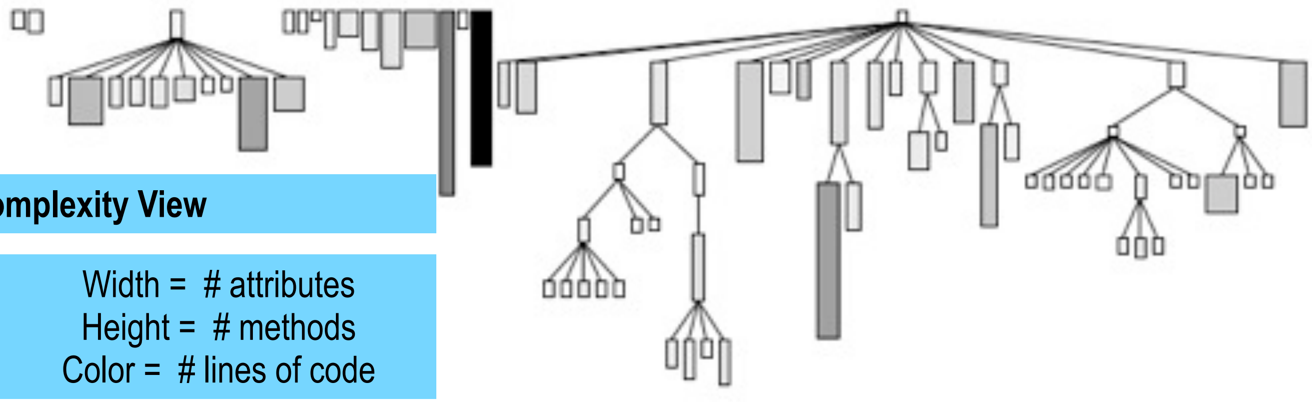
Width = # attributes
Height = # methods
Color = # lines of code

Reverse engineering goals

- Get an impression (build a first raw mental model) of the system, know the size, structure, and complexity of the system in terms of classes and inheritance hierarchies
- Locate important (domain model) hierarchies, see if there are any deep, nested hierarchies
- Locate large classes (standalone, within inheritance hierarchy), locate stateful classes and classes with behavior

View-supported tasks

The Polymetric View - Example (II)



System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

Width = # attributes
Height = # methods
Color = # lines of code

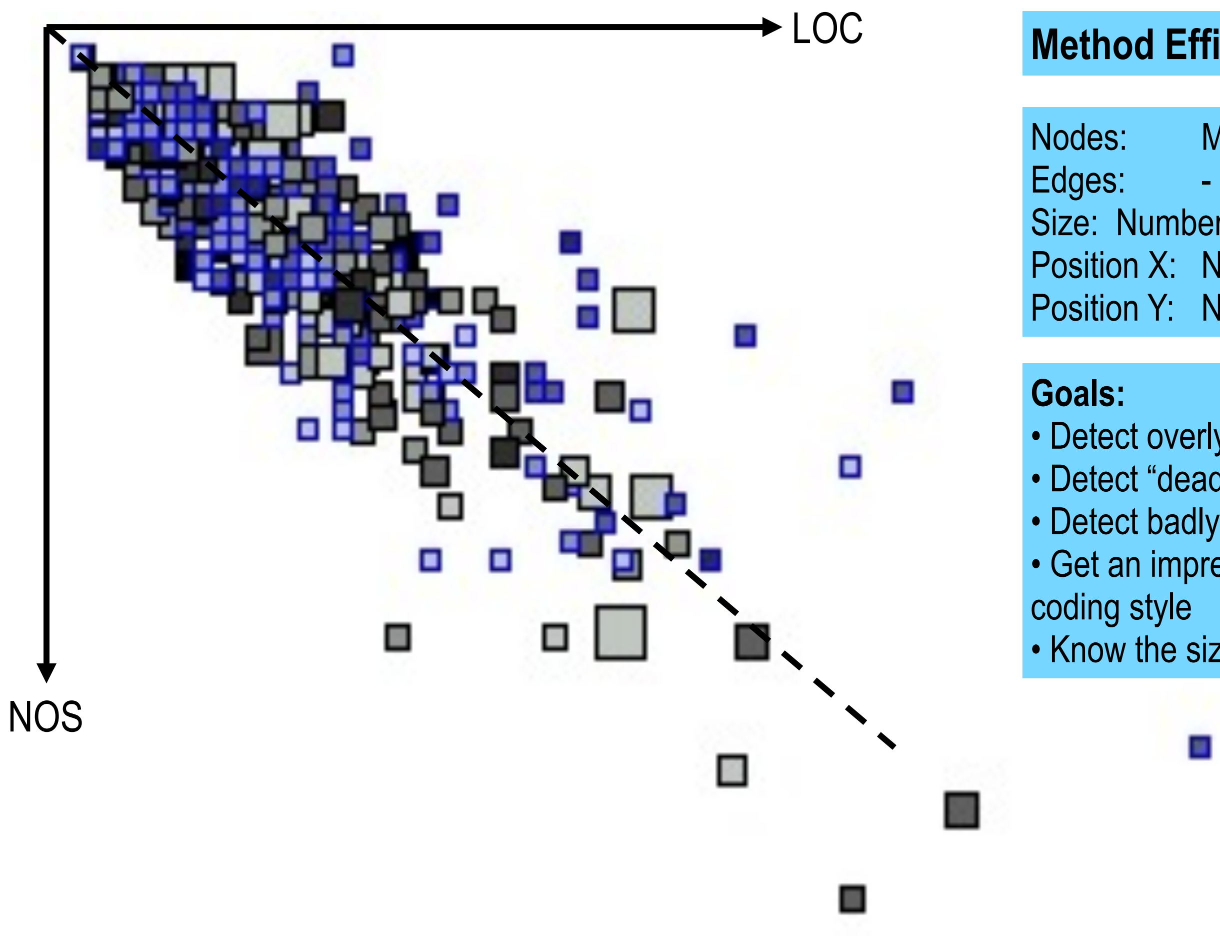
Reverse engineering goals

- Get an impression (build a first raw mental model) of the system, know the size, structure, and complexity of the system in terms of classes and inheritance hierarchies
- Locate important (domain model) hierarchies, see if there are any deep, nested hierarchies
- Locate large classes (standalone, within inheritance hierarchy), locate stateful classes and classes with behavior

View-supported tasks

- Count the classes, look at the displayed nodes, count the hierarchies
- Search for node hierarchies, look at the size and shape of hierarchies, examine the structure of hierarchies
- Search big nodes, note their position, look for tall nodes, look for wide nodes, look for dark nodes, compare their size and shape, "read" their name => opportunistic code reading

Coarse-grained Polymetric Views - Example

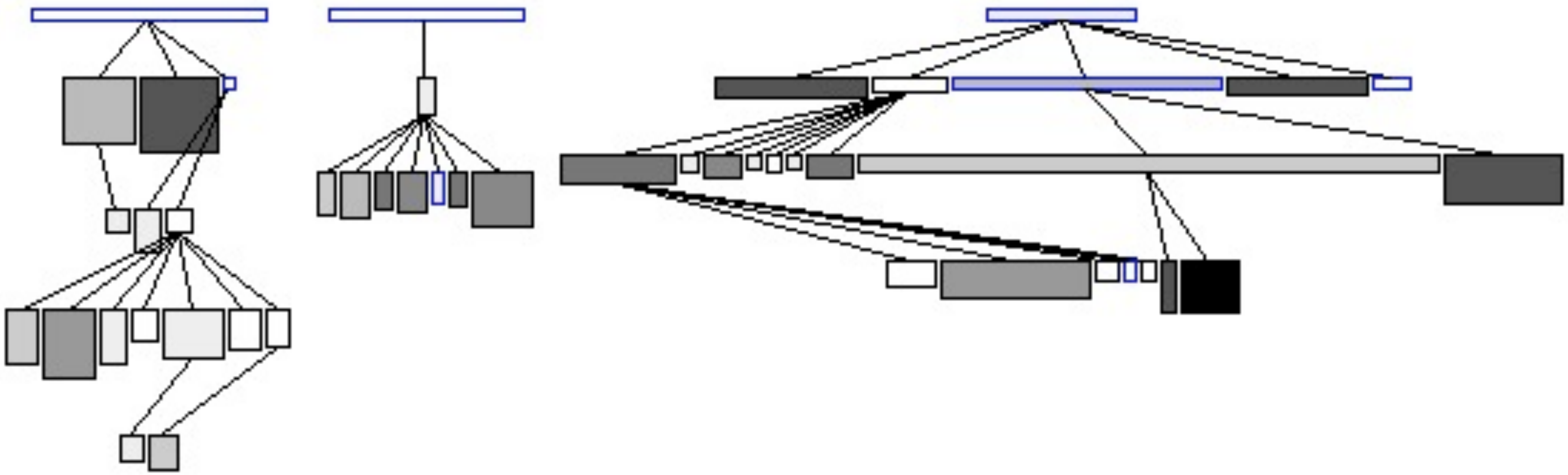


Method Efficiency Correlation View

Nodes: Methods
Edges: -
Size: Number of method parameters
Position X: Number of lines of code
Position Y: Number of statements

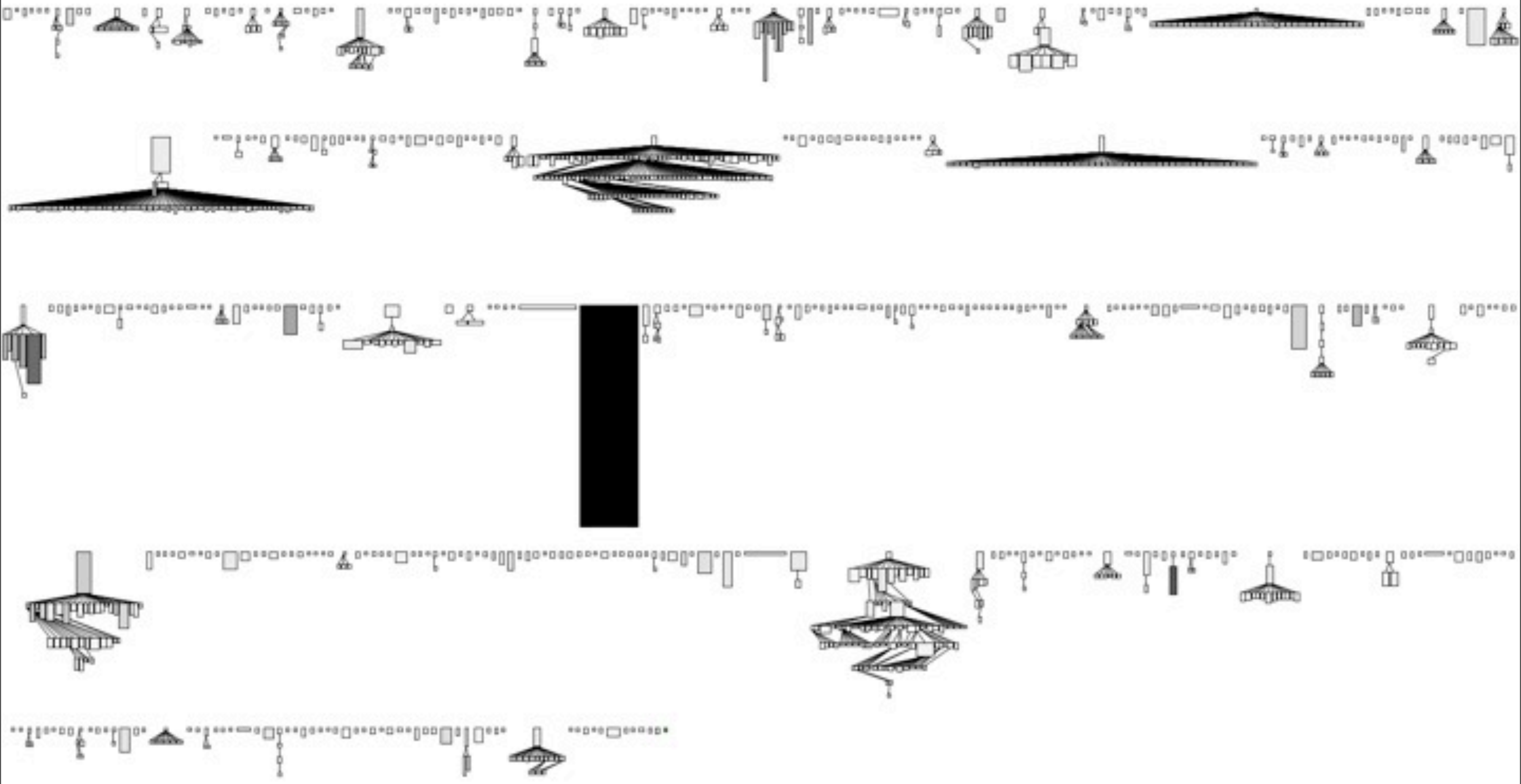
Goals:
- Detect overly long methods
- Detect "dead" code
- Detect badly formatted methods
- Get an impression of the system in terms of coding style
- Know the size of the system in # methods

Inheritance Classification View

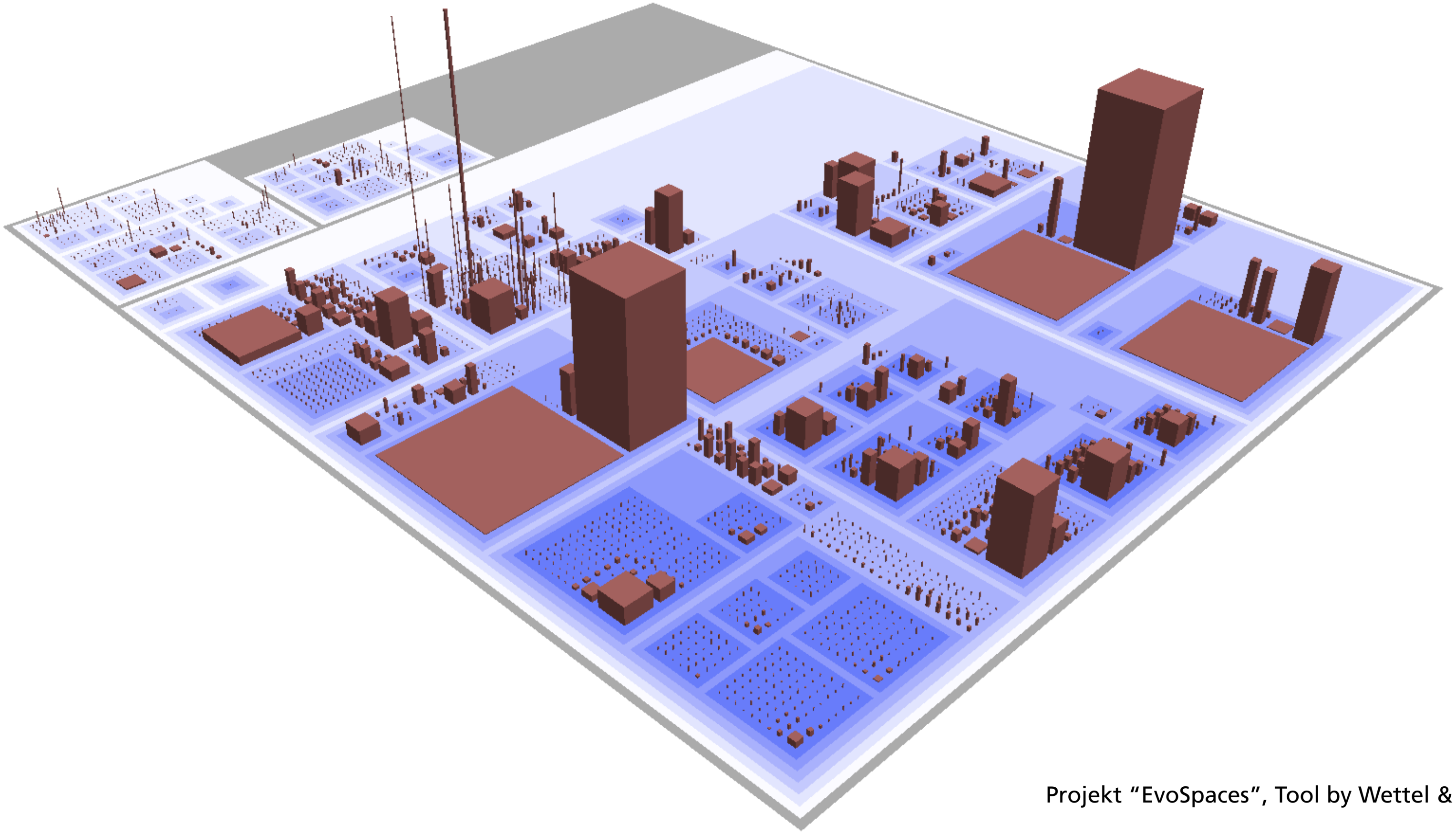


Boxes:	Classes
Edges:	Inheritance
Width:	Number of Methods Added
Height:	Number of Methods Overridden
Color:	Number of Method Extended

Polymetric View Example: ArgoUML

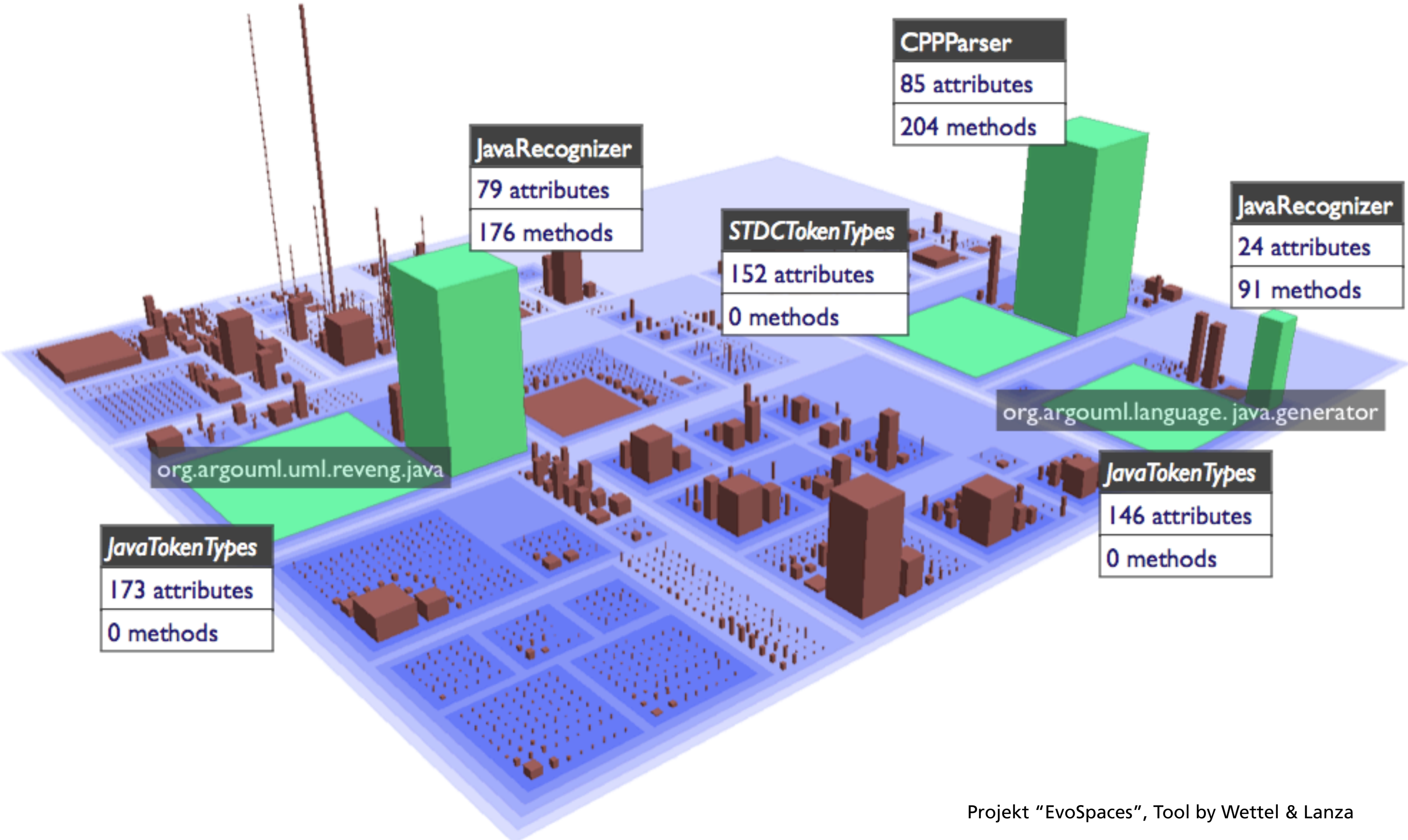


Software Architecture Exploration



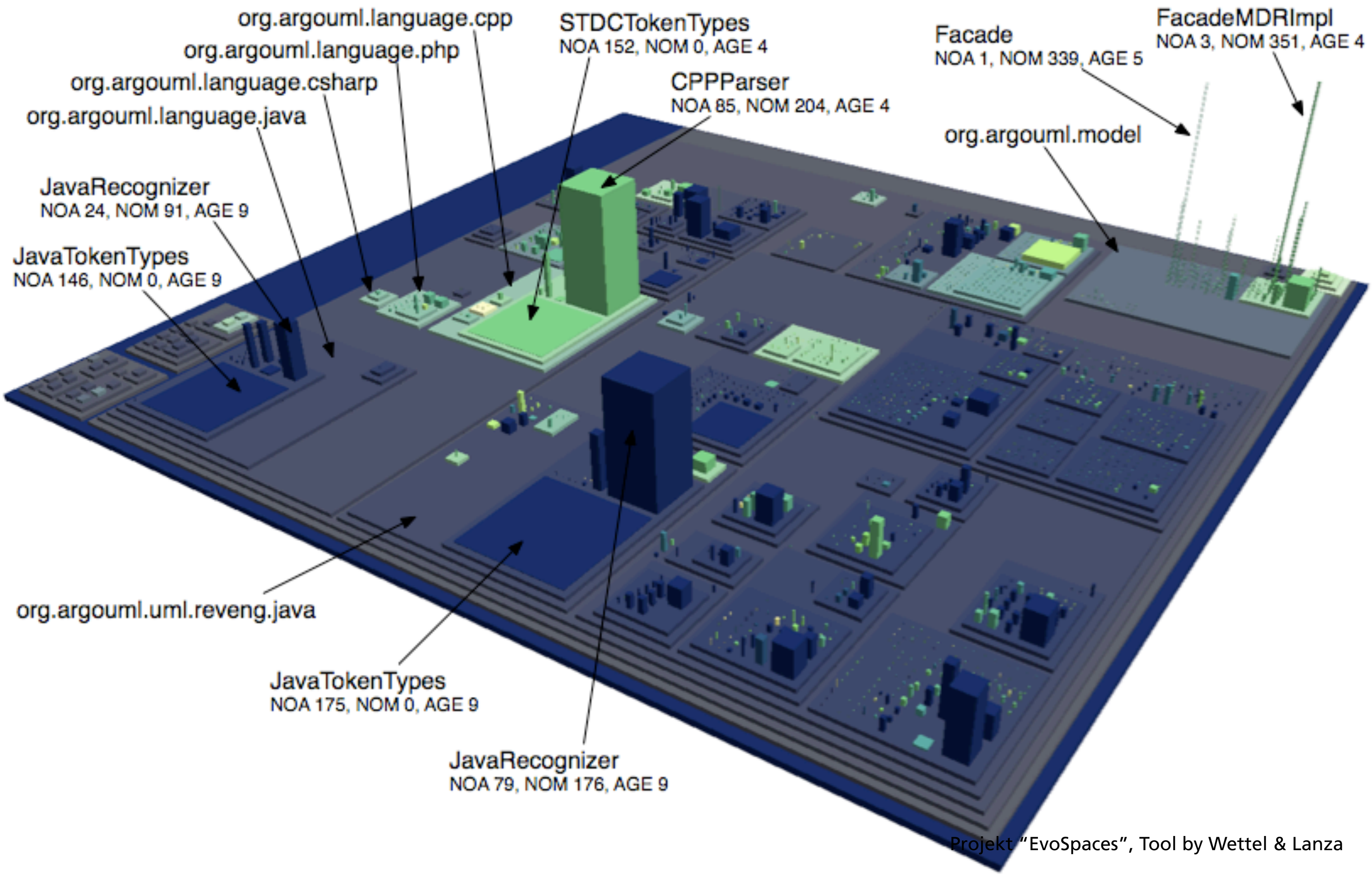
Projekt "EvoSpaces", Tool by Wettel & Lanza

ArgoUML City



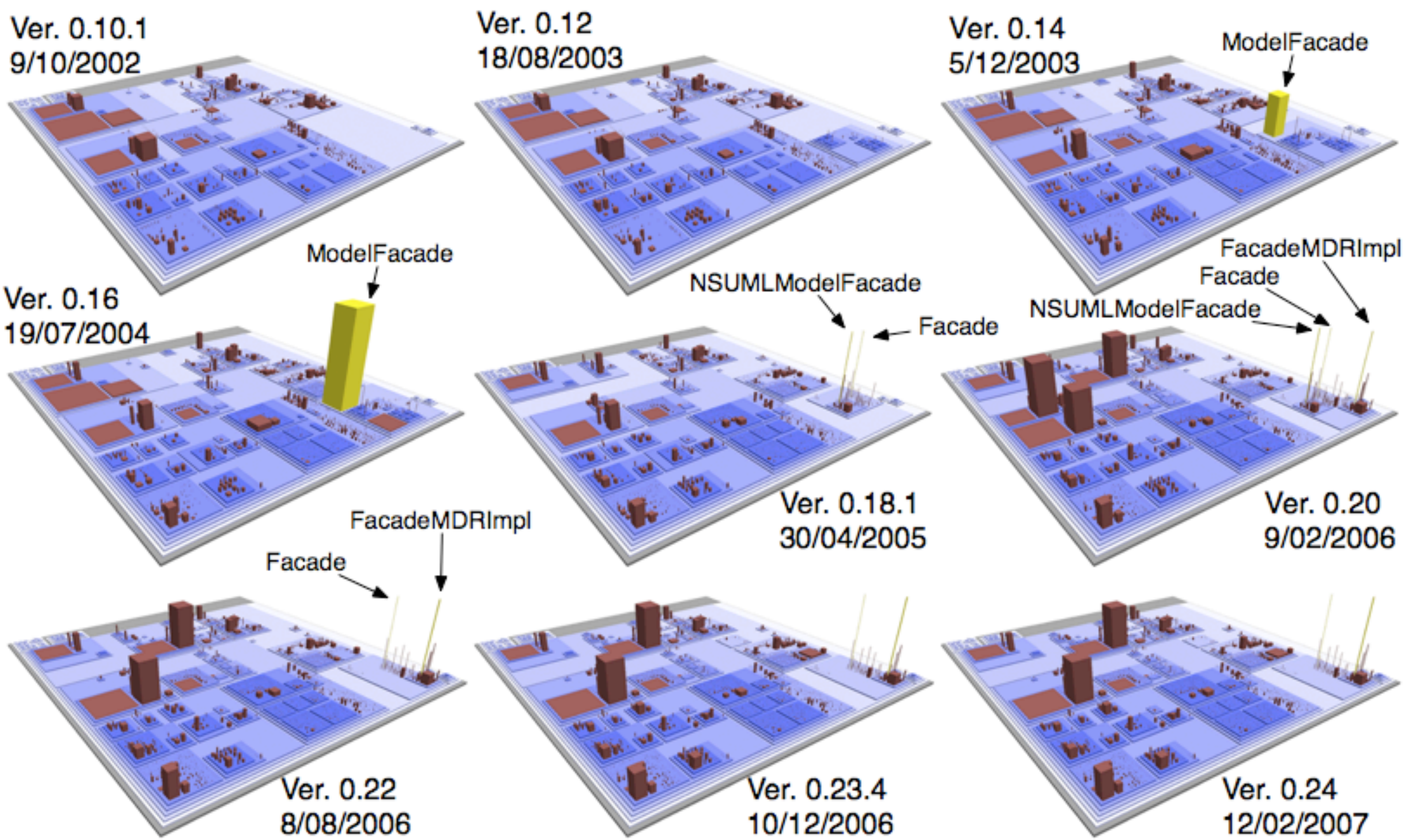
Projekt “EvoSpaces”, Tool by Wettel & Lanza

The age of a City

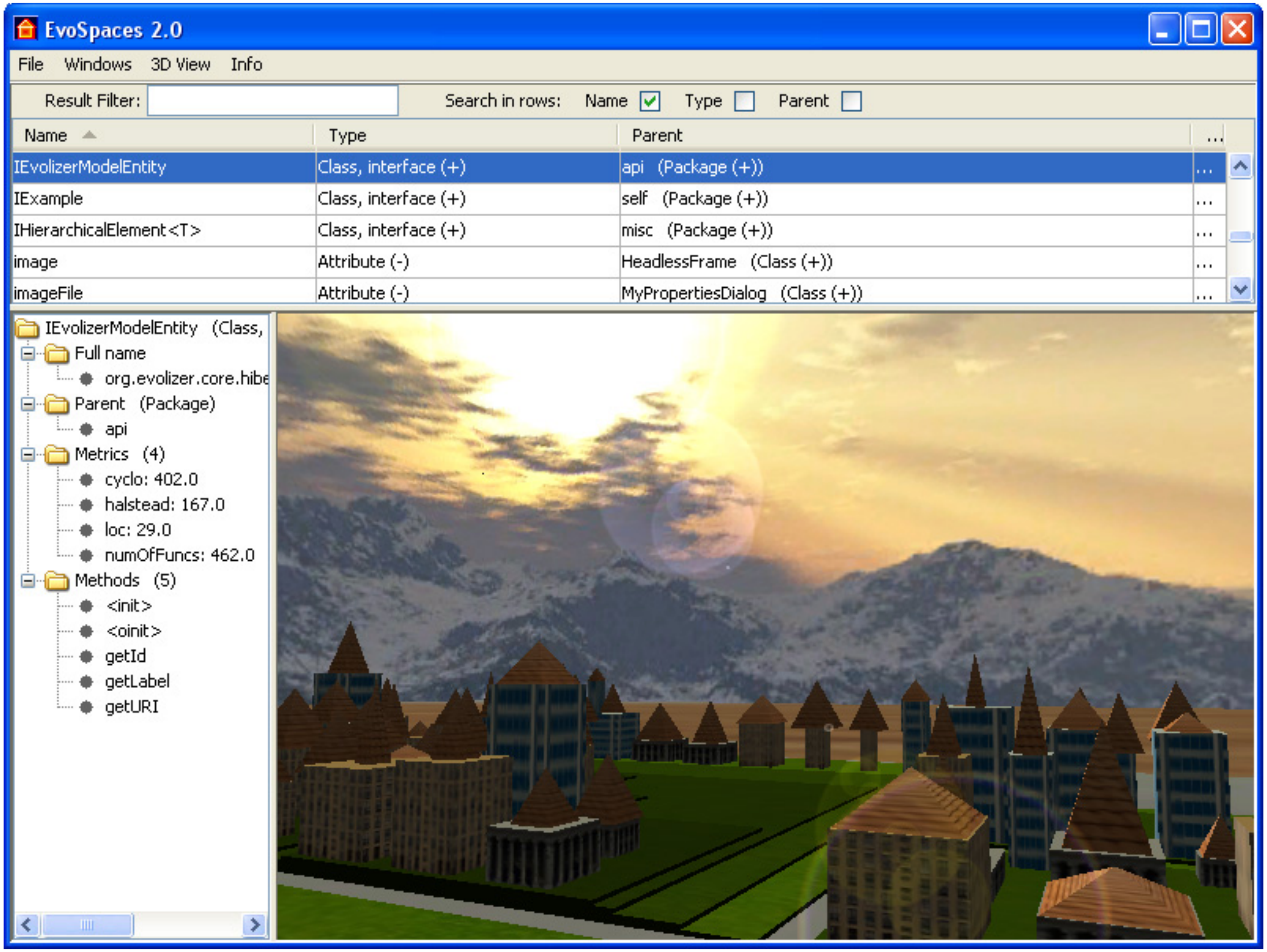


Projekt "EvoSpaces", Tool by Wettel & Lanza

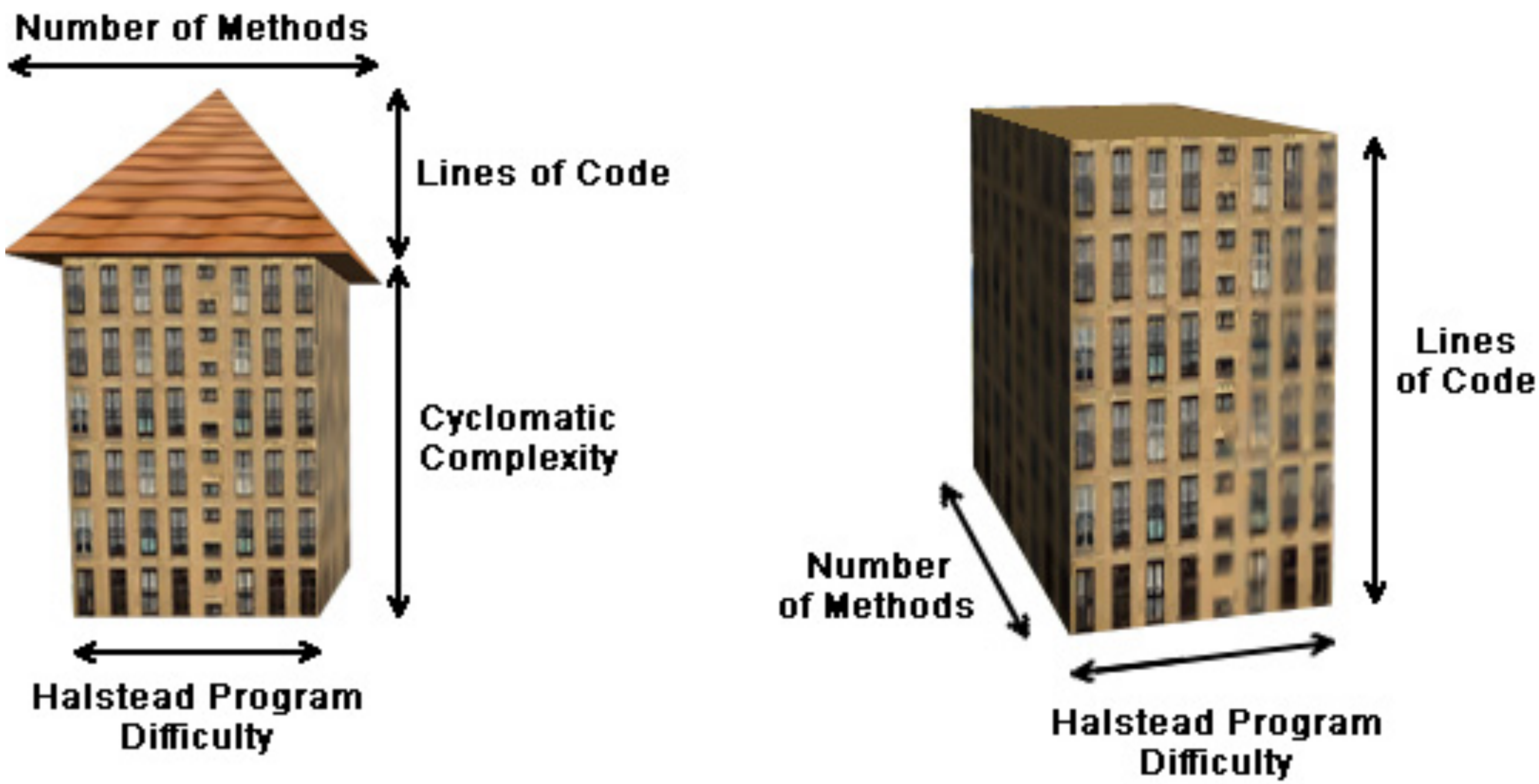
Evolution of a City



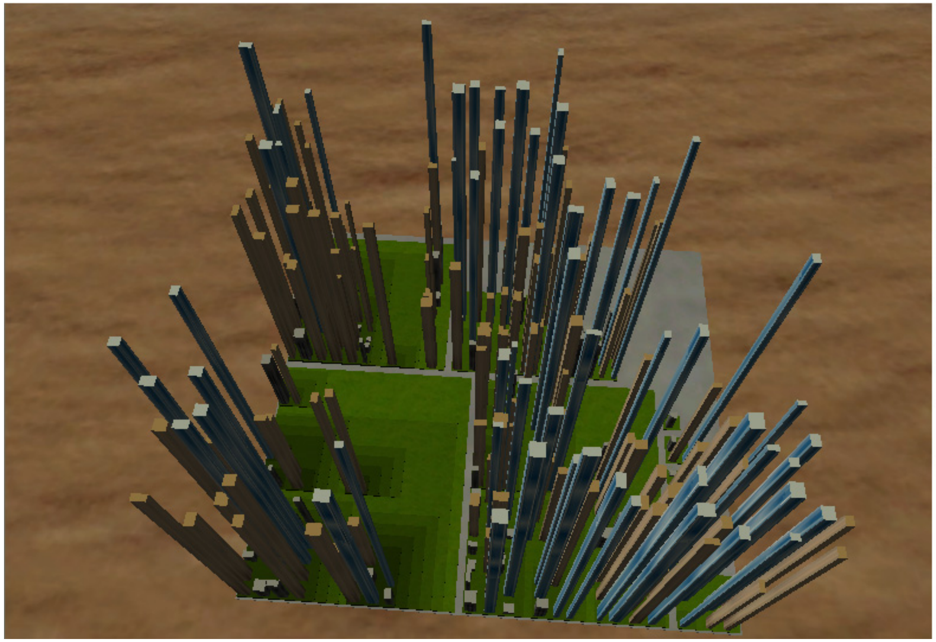
EvoSpaces Tool

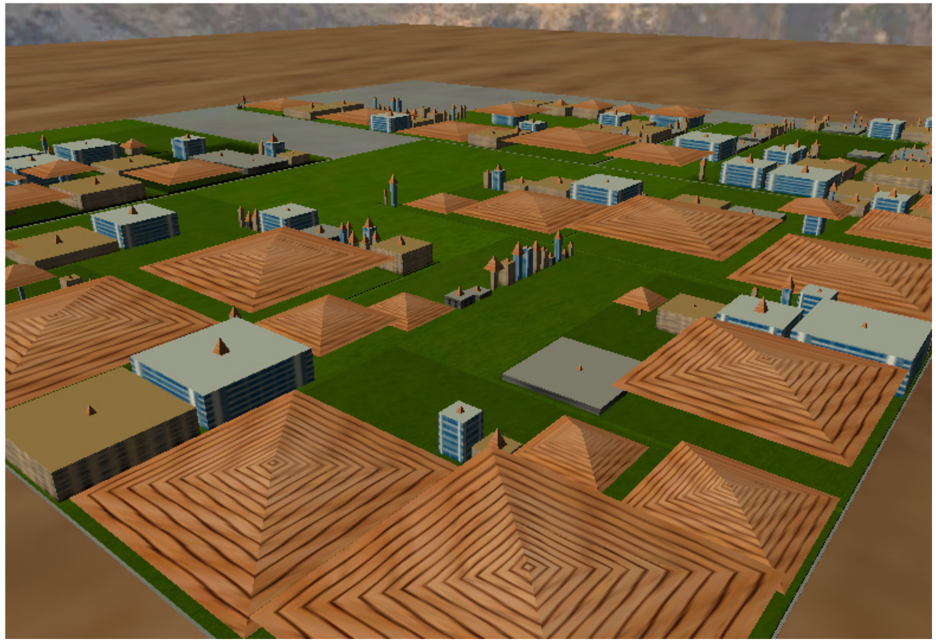


EvoSpaces: a closer look



Metric look forms a City





Reflections on Visualization

Visualizations are useless…

…as pictures: Polymetric views are navigable & interactive

…if not accessible: Polymetric views are implemented in…

CodeCrawler, Mondrian, Sotograph, Jsee, etc.

It will take some time and a lot of work for them to be accepted - time will tell

“Everything must change to remain the same”
[Giuseppe Lanza Tomasi di Lampedusa, “Il Gattopardo”]

Evaluating the Design of a System

What entities do we measure in object-oriented design?

It depends…on the language

What metrics do we use?

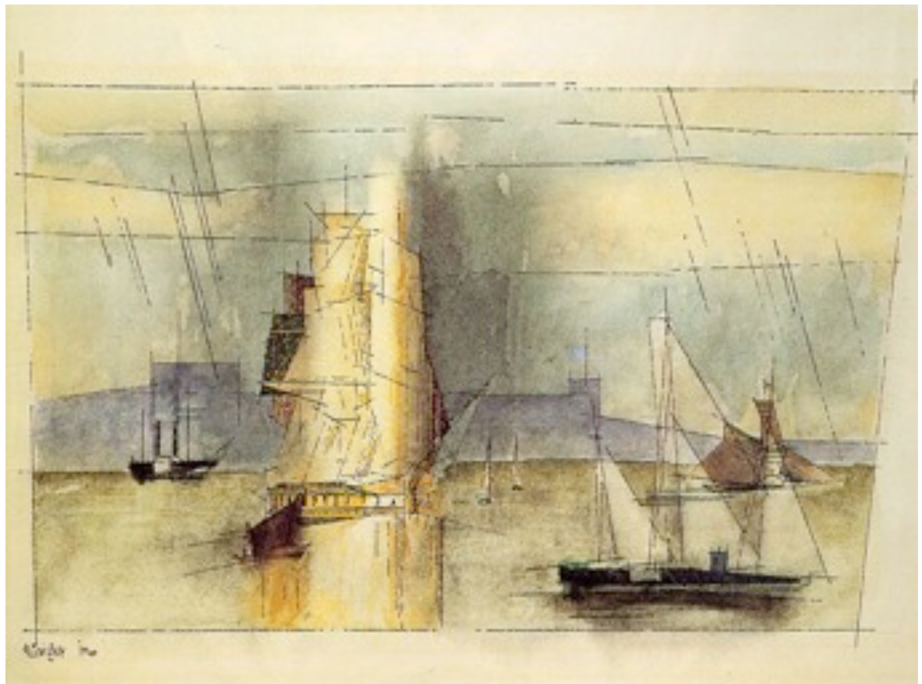
It depends…on our measurement goals

What can we do with the information obtained?

It depends…on our objectives

Simple metrics are not enough to understand and evaluate design

Can you understand the beauty of a painting by measuring its frame?



Design Heuristics

Professional Context

There has been excellent work in Software Design

- Design Patterns
- Design Heuristics
- Refactorings
- Quality Models

What is good design?

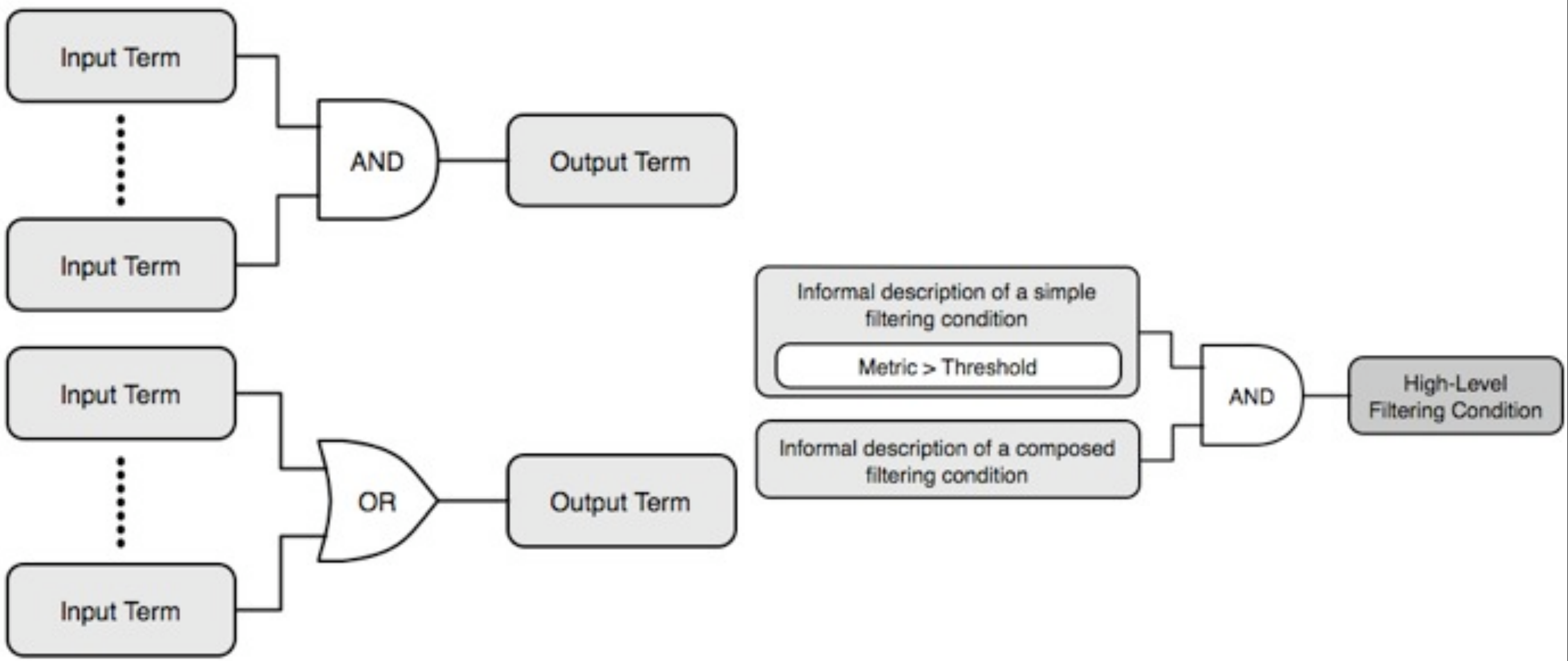
What is bad design?

How do we detect design?

- Detection Strategies
- The Class Blueprint

Detection Strategies

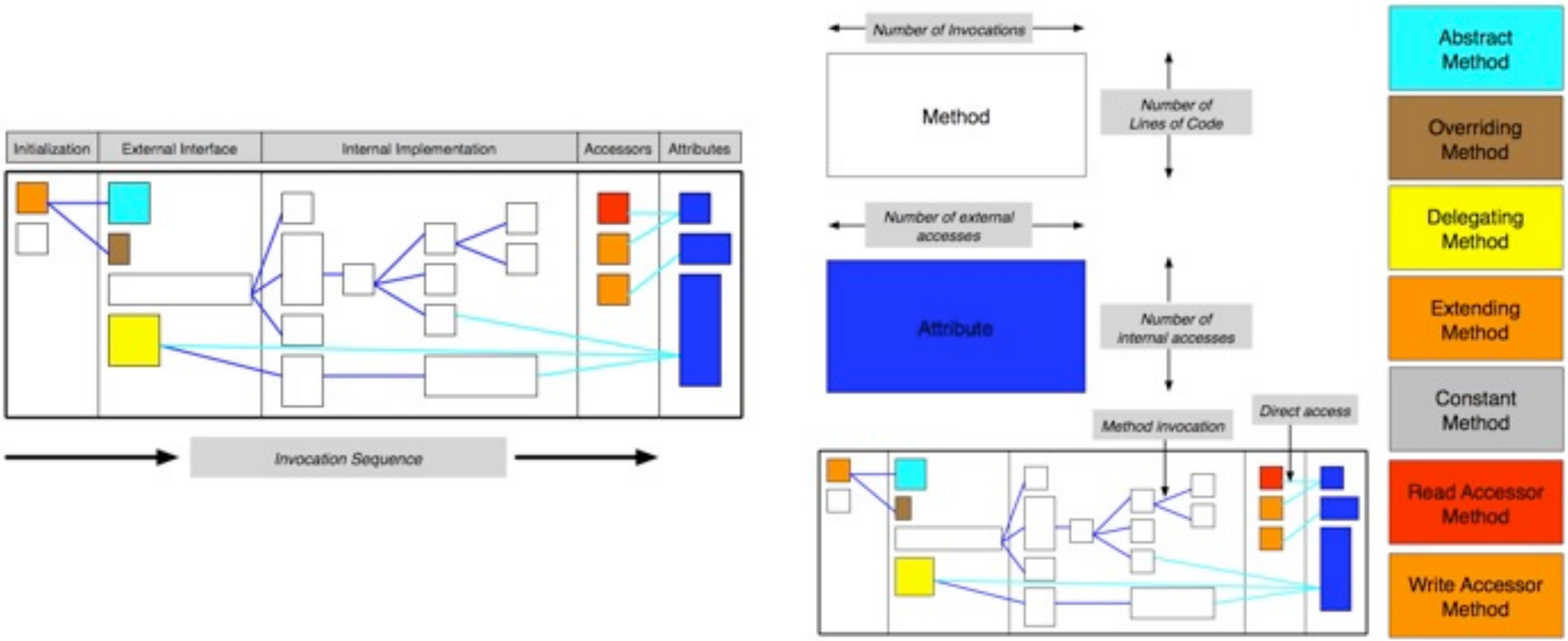
A detection strategy is a metrics-based predicate to identify candidate software artifacts that conform to (or violate) a particular design rule



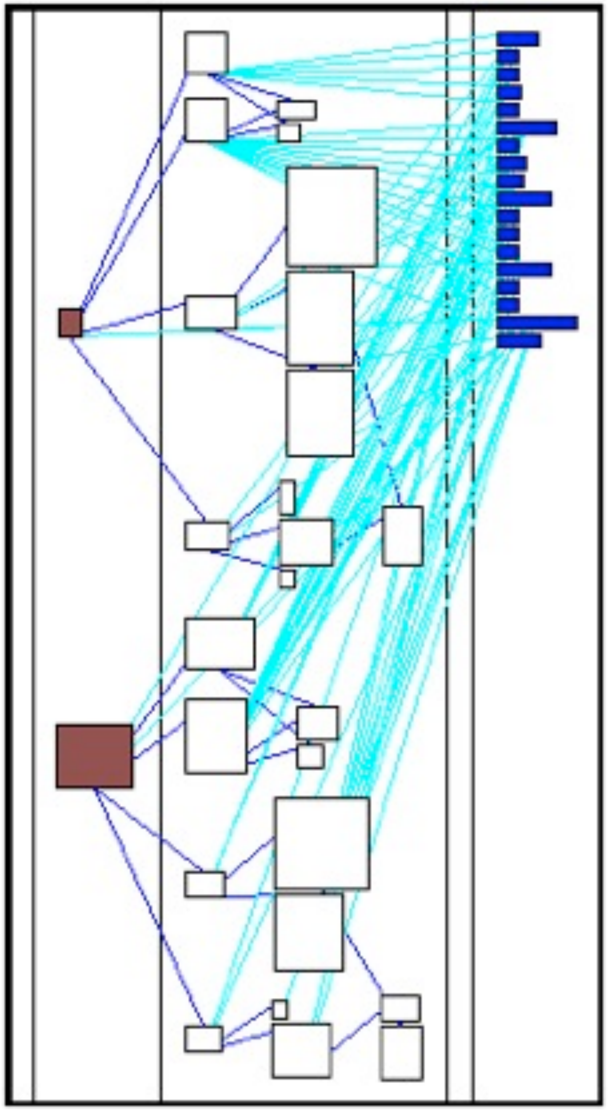
The Class Blueprint

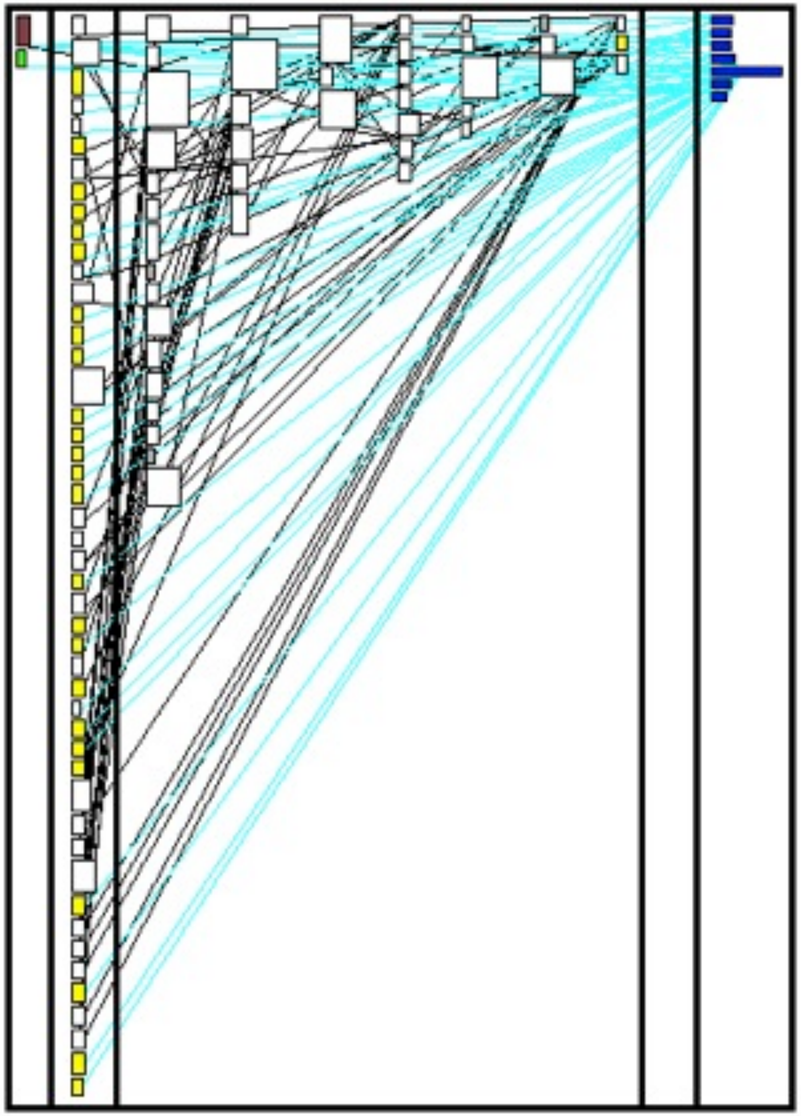
A semantically rich visualization of the internal structure of classes and class hierarchies

- Useful for inspecting source code, and detecting visual anomalies which point to design disharmonies

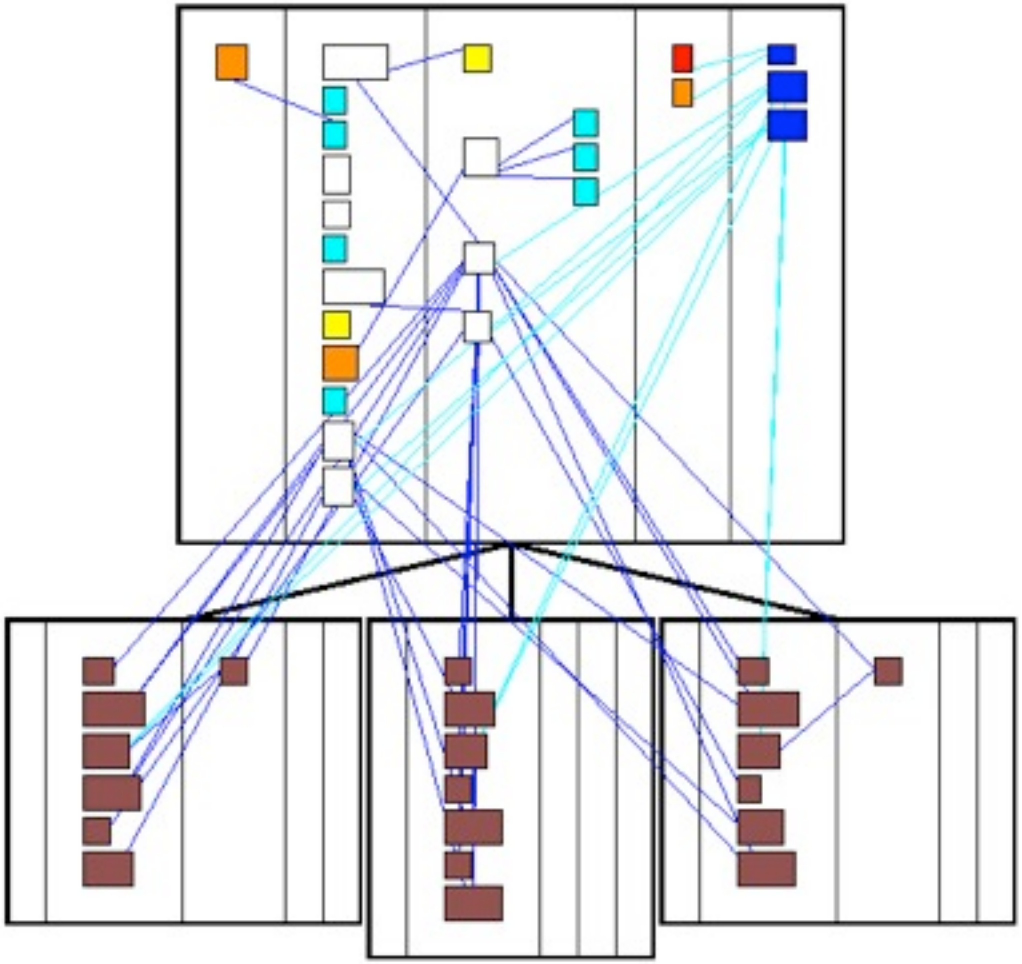


The Class Blueprint: Seeing Code & Design





The Class Blueprint - What do we see?



Nice! …but, what about the practice?

In practice the key question is where to start

We have devised a methodology to characterize, evaluate and improve the design of object-oriented systems

It is based on:

- The Overview Pyramid
- The System Complexity View
- Detection Strategies
- Class Blueprints

Design Harmony

Software is a human artifact

There are several ways to implement things

The point is to find the appropriate way!

Appropriate to what?

- Identity Harmony
  - How do I define myself?
- Collaboration Harmony
  - How do I interact with others?
- Classification Harmony
  - How do I define myself with respect to my ancestors and descendants?

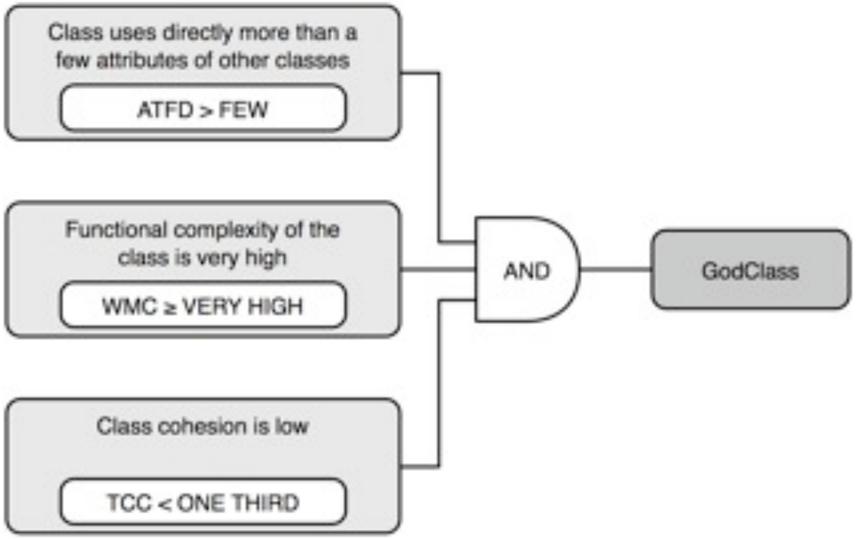
Let's see some examples

Identity Disharmony: God Class

An aggregation of different abstractions which (mis)uses other classes to perform its functionality

- The "other" classes are usually dumb data holders
- Difficult to cure: only do it if it hampers evolution

Detection: Find large and complex classes on which many other classes depend

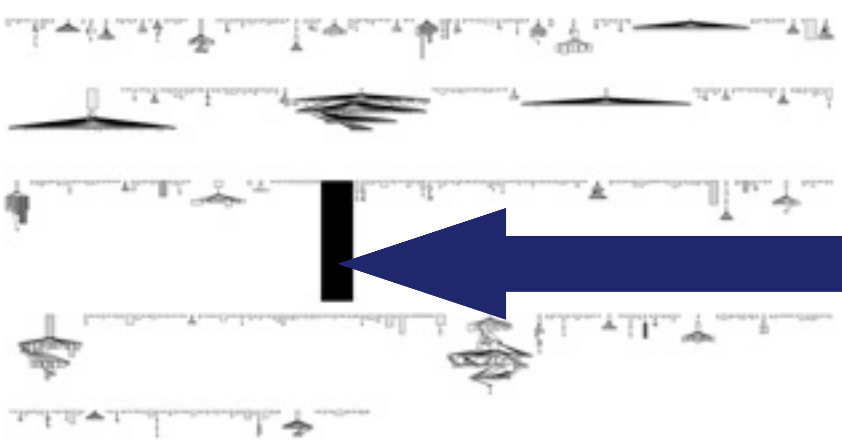


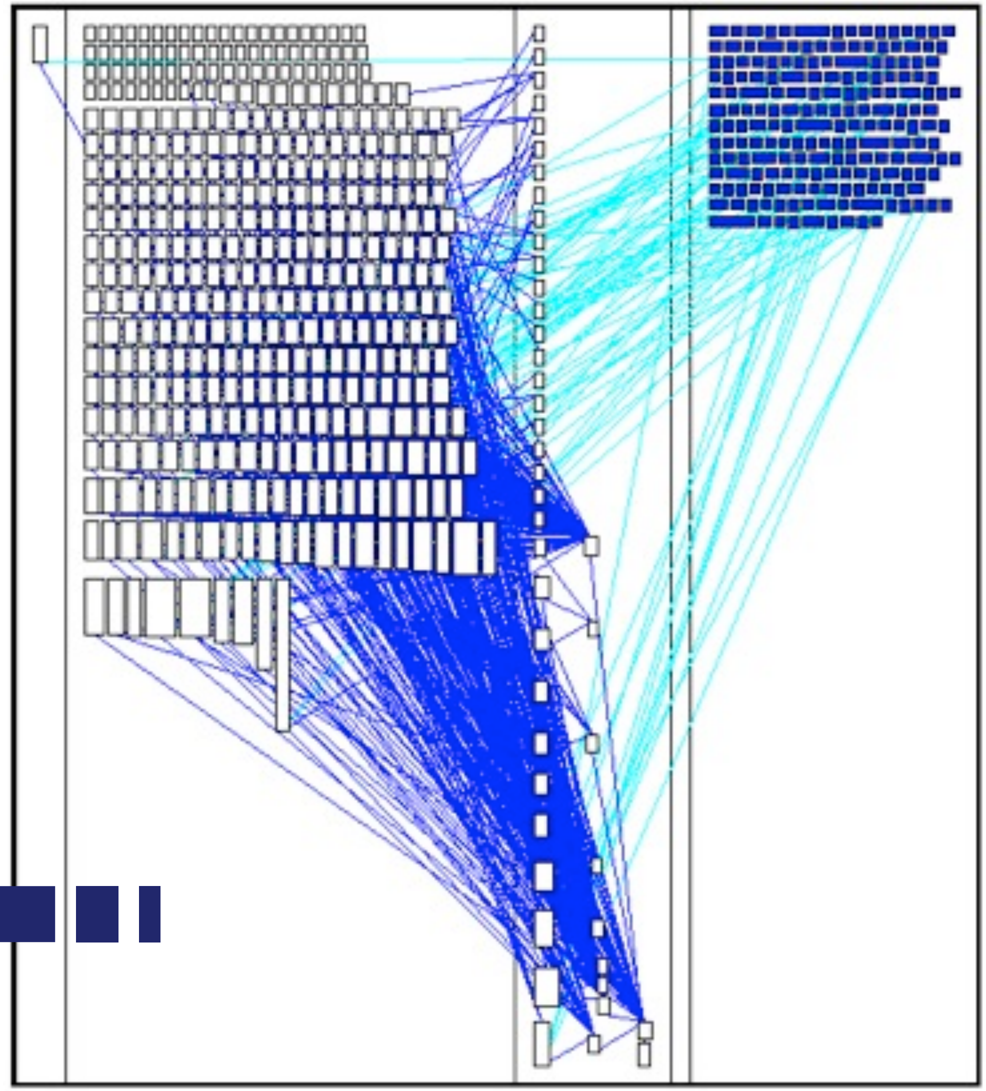
Oh my God…it's the ModelFacade

ModelFacade: The Black Hole

- 453 methods
- 114 attributes
- 3500 lines of code

Coupled to hundreds of ArgoUML classes



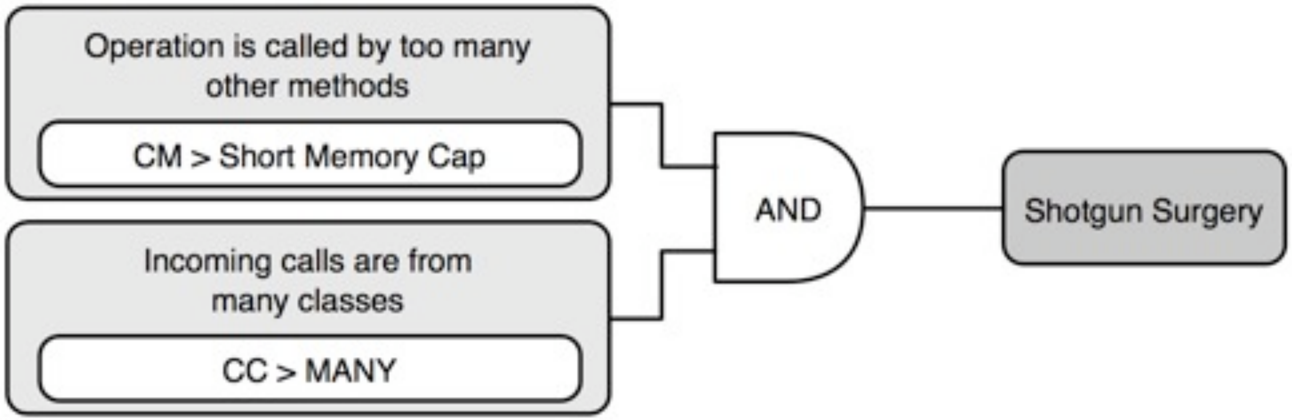


Collaboration Disharmony: Shotgun Surgery

A change in a method may imply changes in many places

Detection: Find the classes in which a change would significantly affect many other places in the system

- We have to consider both the strength and the dispersion of the coupling
- We focus on incoming coupling

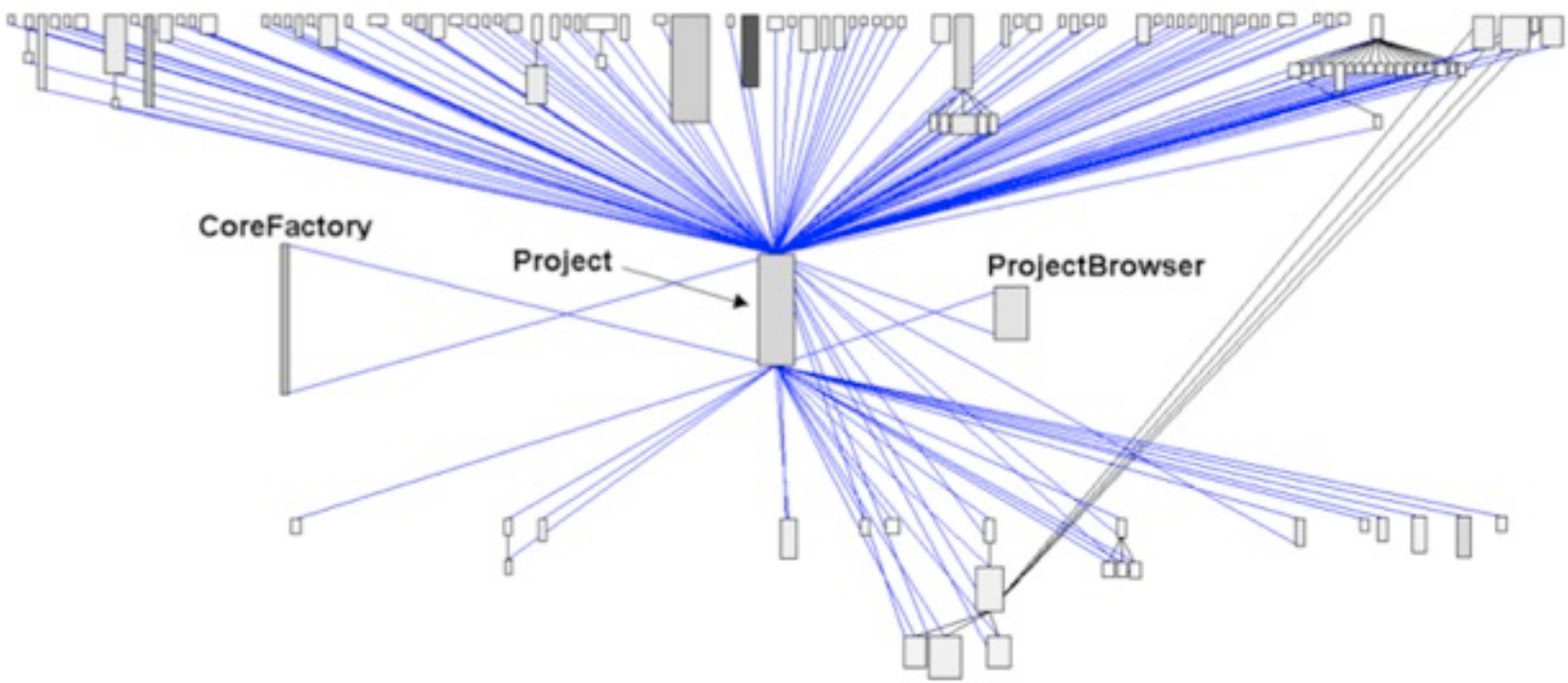


I shot…the Project…

Project has several methods affected by SS

- Coupled with 131 classes (ModelFacade not shown here)
- Cyclic Dependencies with CoreFactory & ProjectBrowser

Changing Project may lead to problems

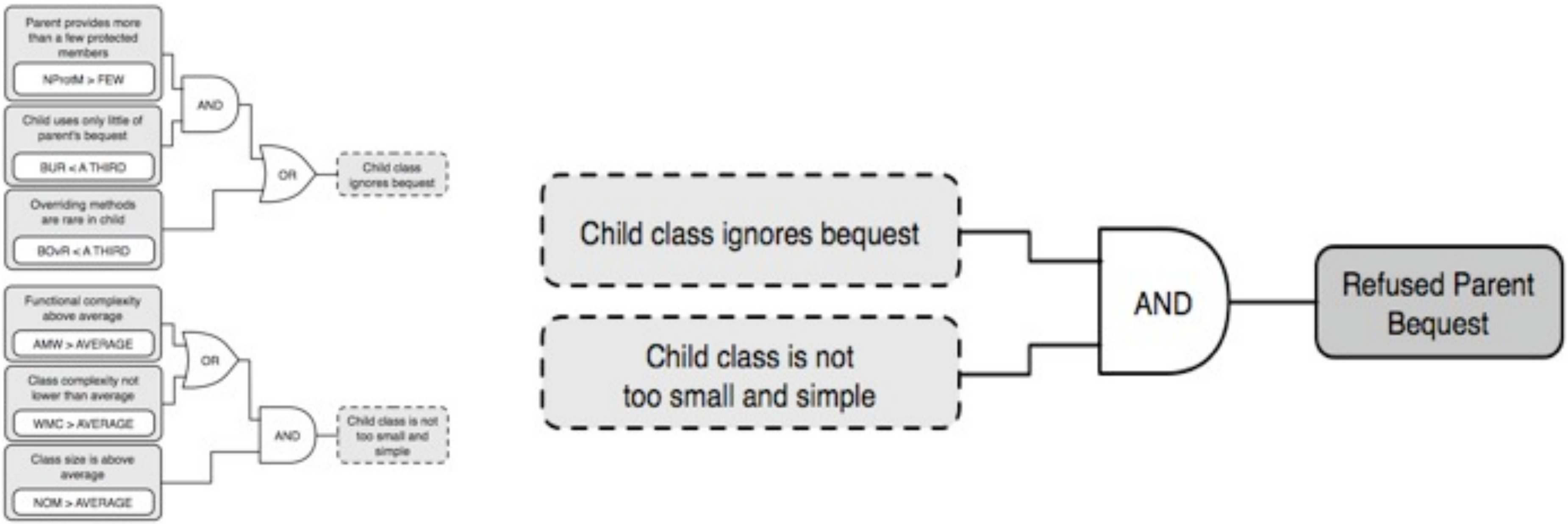


Classification Disharmony

The primary goal of inheritance: code reuse

- When you add a subclass you should look at what is "already there": add/extend-abstract-change cycle

Detection: Find fairly complex classes with low usage of inheritance-specific members of the superclass(es)

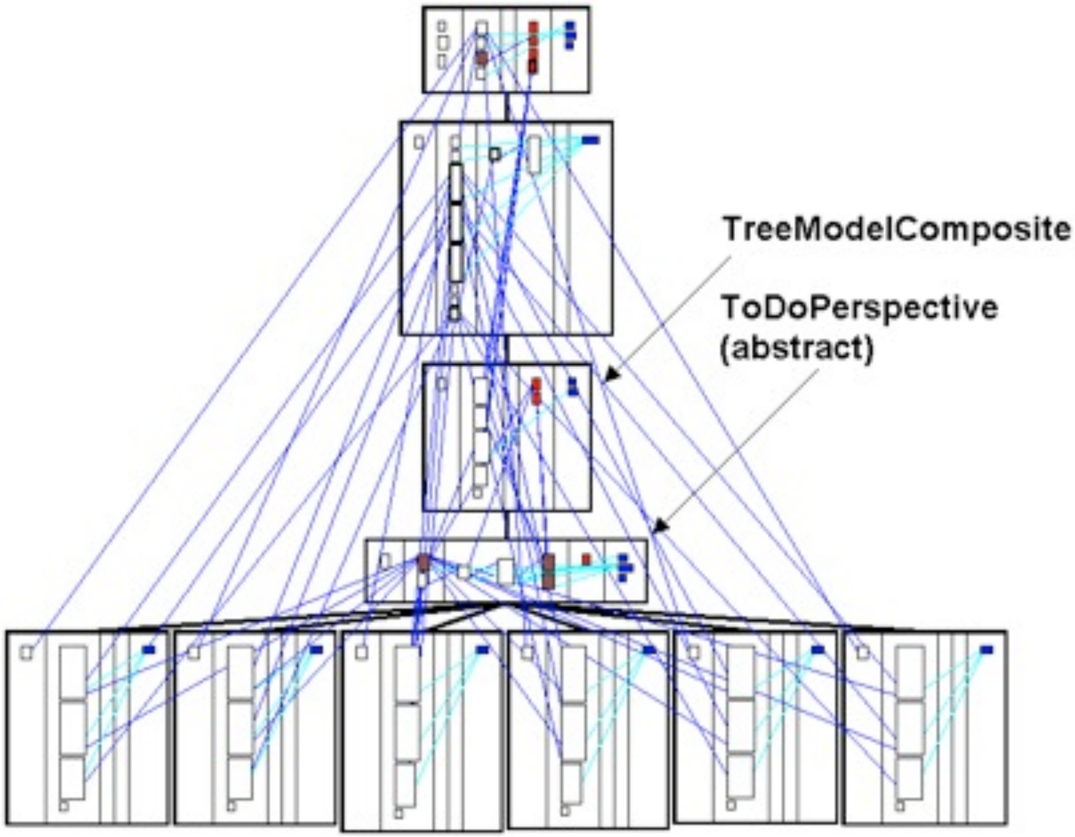


Kids never listen: The PerspectiveSupport Hierarchy

"Pipeline"-Inheritance with funky usage of abstract classes

Suspicious regularity in the leaf classes: duplicated code

TreeModelComposite ignores what is the superclasses

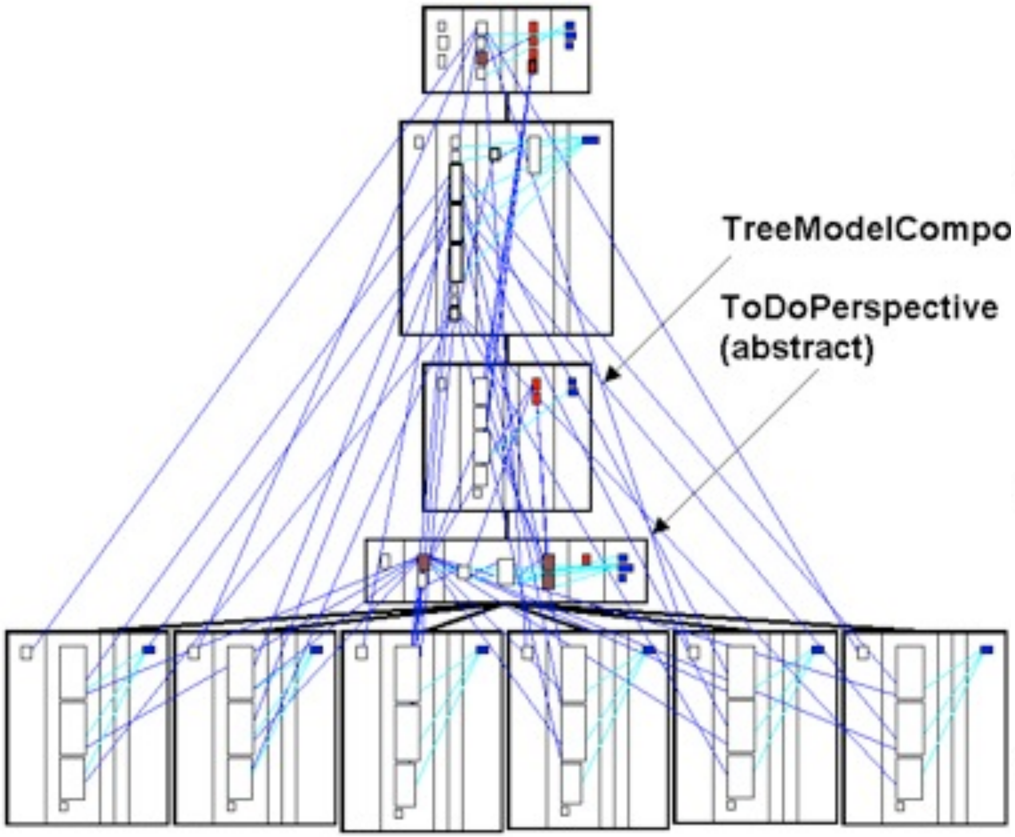


Kids never listen: The PerspectiveSupport Hierarchy

"Pipeline"-Inheritance with funky usage of abstract classes

Suspicious regularity in the leaf classes: duplicated code

TreeModelComposite ignores what is the superclasses



Recovering from a Design Disharmony

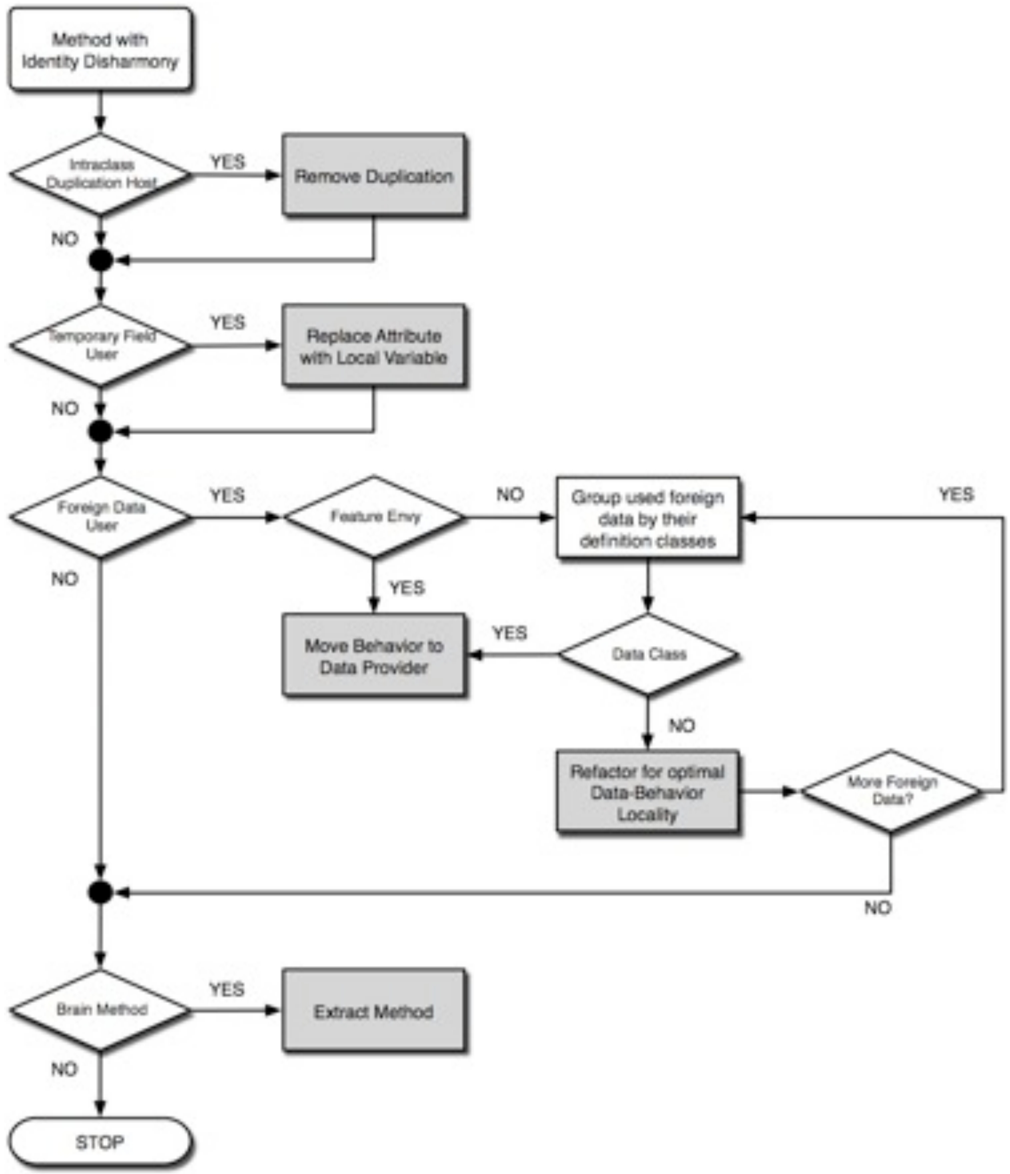
Misery loves company:

- The Design Disharmonies do not exist alone, they are correlated

Where to start?

How to start?

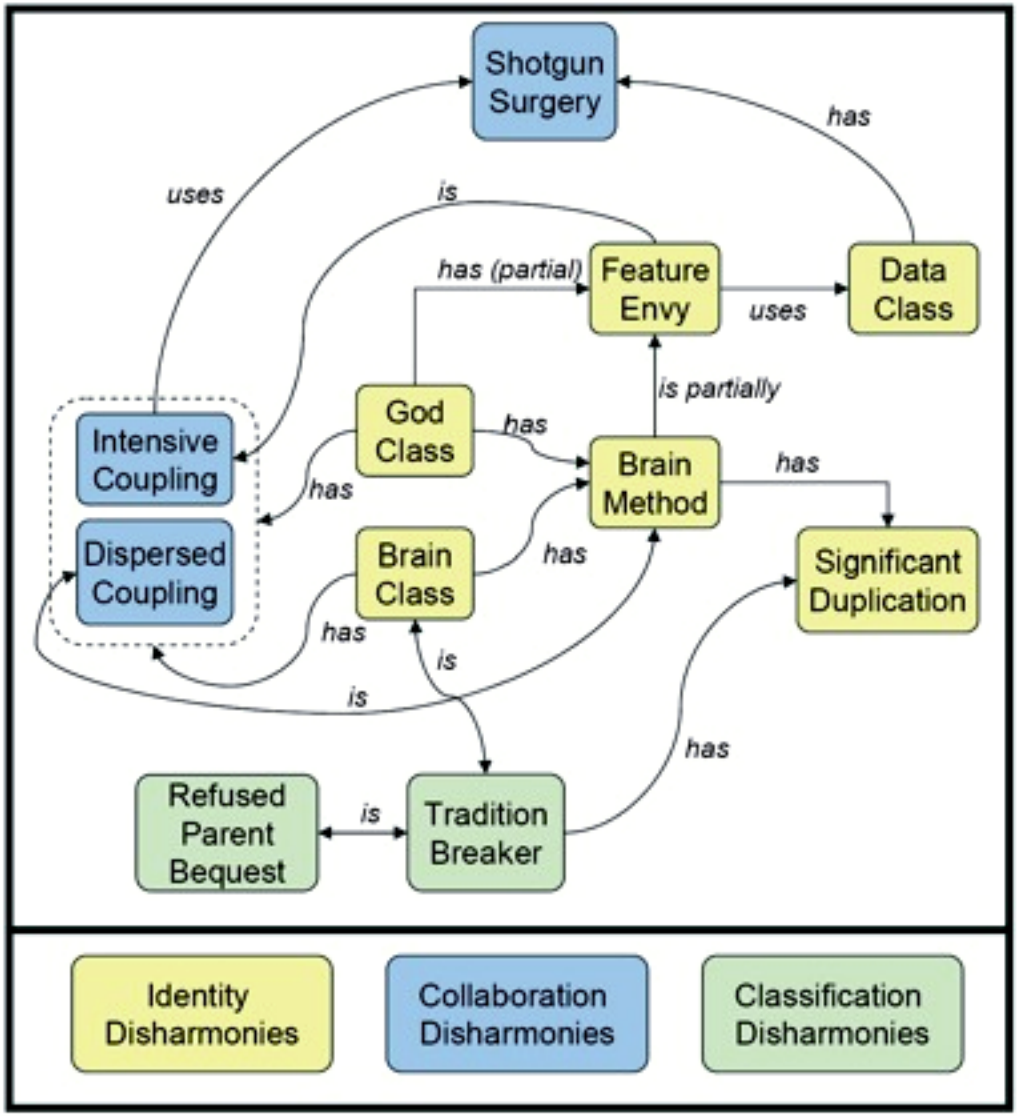
Recovering can be a lengthy process and must be evaluated in terms of effort/benefit



A Catalogue of Design Disharmonies

For each Design Disharmony, we provide

- Description
- Context
- Impact
- Detection Strategy
- Examples
- Refactoring



Tools

"A fool with a tool is still a fool", but…

Better a fool with a tool than just a fool…

Everything presented is based on extensive tooling

- Moose
- CodeCrawler
- iPlasma
- Free and open source - take it or leave it

(Parts of) these tools are now making it into industry

- The Disharmonies are now part of "Borland Together"

Software Visualization: Conclusions

Software Visualization is very useful when used correctly

An integrated approach is needed, just having nice pictures is not enough

Most tools still at prototype level

In general: only people that know what they see can react on that: SV is for expert/advanced developers

The future of software development is coming…and SV is part of it