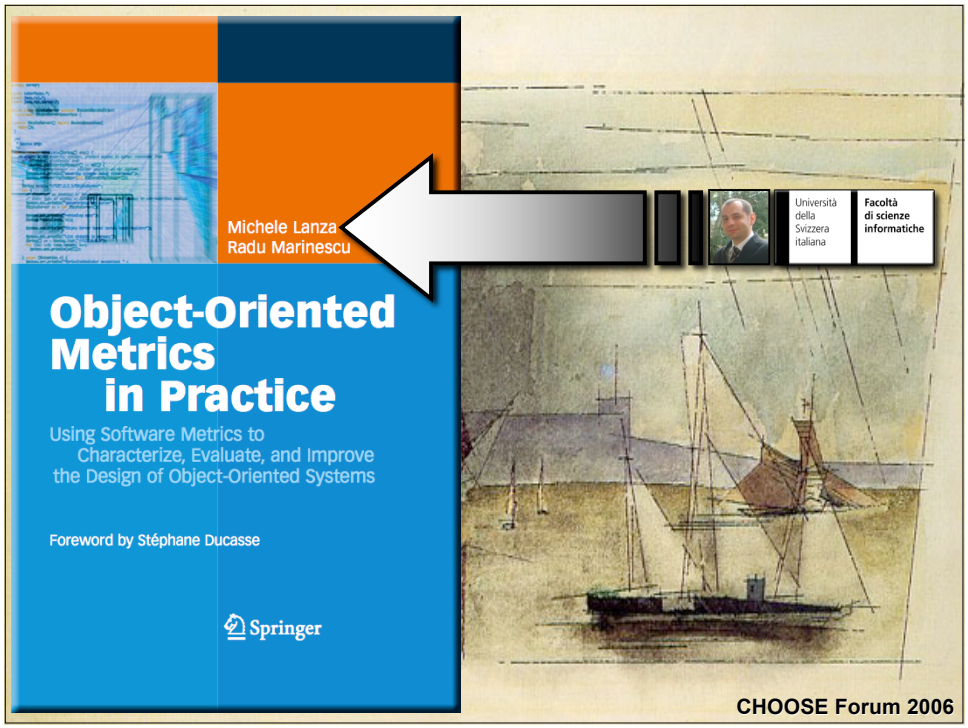
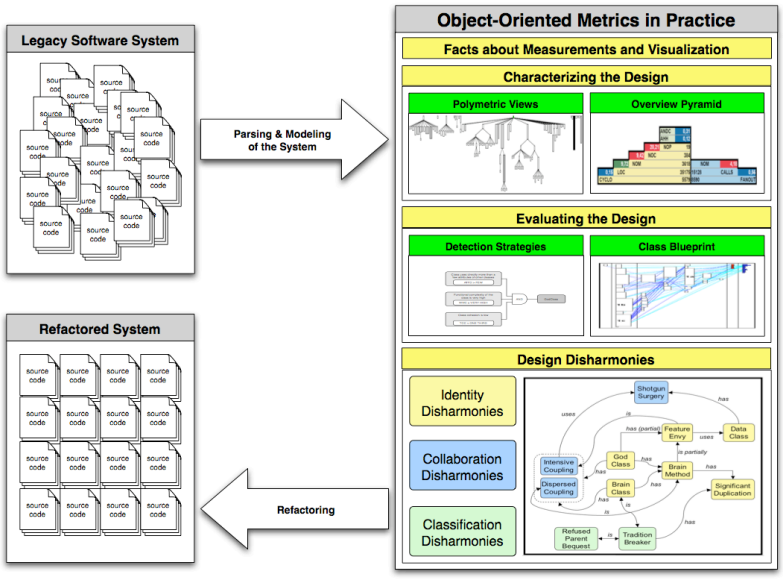
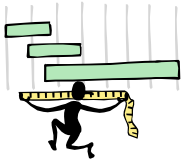


OOMiP in the (Nut)Shell



Metrics

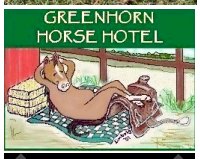
- What is a metric?
  - The mapping of a particular characteristic of a measured entity to a numerical value
- Why is it useful to measure?
  - To keep control of…complexity
- Advantages
  - Ability to quantify aspects of quality
  - Possibility to automate the “measurements” of systems
- Drawbacks
  - 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



Use Metrics - but with higher-level mechanisms

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

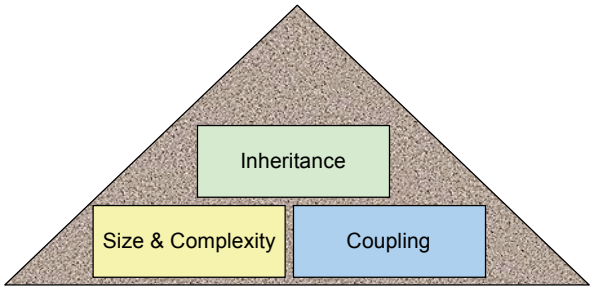
- "Yesterday I met a system…"
  - How many lines of code?
    - 35'000 LOC
  - How many functions/methods?
    - 3'600 NOM
  - How many classes?
    - 380 NOC
  - Etc…
- Is it "normal" to have a system of…
  - 380 classes with 3600 methods?
  - 3600 methods with 35'000 lines of code?
- What is "normal"? What about coupling? What about 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? Number of Classes? Number of Methods? Megabytes? Files?
- Characterizing a System with one or two 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"?

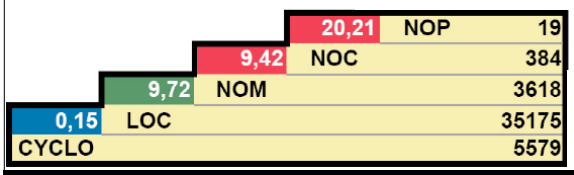
The Overview Pyramid

- 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



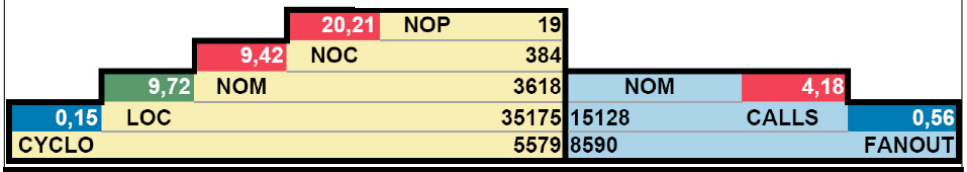
The Overview Pyramid in Detail

- The left side: System Size & Complexity
  - Direct metrics: NOP, NOC, NOM, LOC, CYCLO
  - Derived metrics: NOC/P, NOM/C, LOC/M, CYCLO/LOC



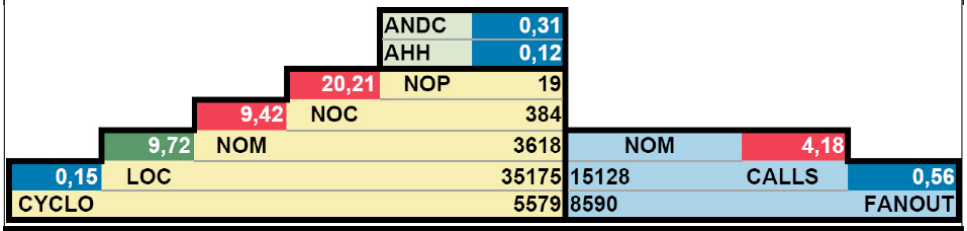
The Overview Pyramid in Detail

- 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



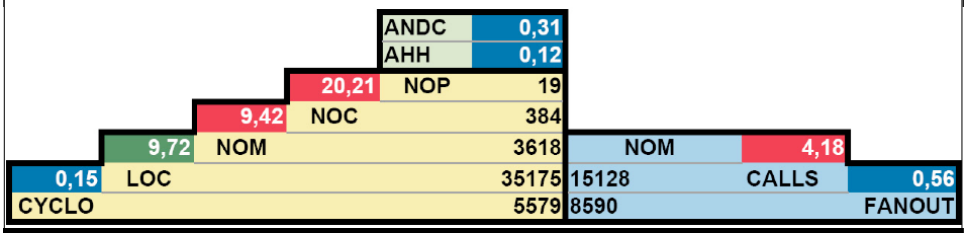
The Overview Pyramid in Detail

- 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
- The top: System Inheritance
  - Direct metrics: ANDC, AHH



Interpreting the Overview Pyramid

- The pyramid characterizes a system from the viewpoints of size&complexity, coupling, and inheritance; based on the 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?

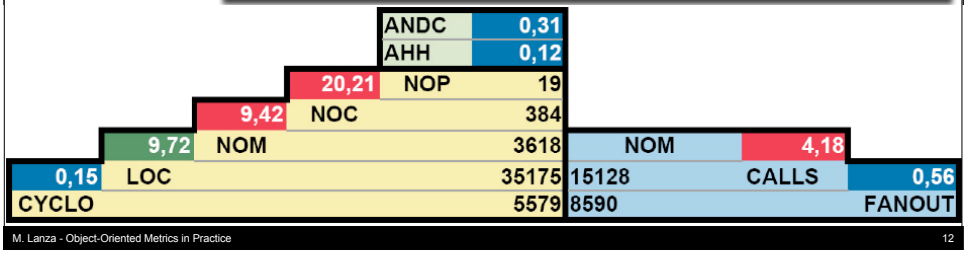


Putting things in a real-world context

- We have measured dozens of systems written in Java and C++
- Based on the obtained measurements we can now statistically assess the design of a system

High
Average
Low

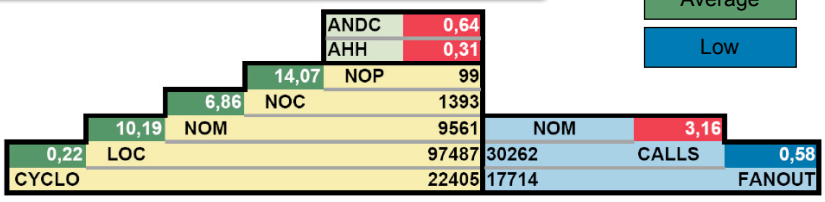
	Java			C++		
Metric	Low	Average	High	Low	Average	High
CYCLO/Line of code	0.16	0.20	0.24	0.20	0.25	0.30
LOC/Operation	7	10	13	5	10	16
NOM/Class	4	7	10	4	9	15
NOC /Package	6	17	26	3	19	35
CALLS/Operation	2.01	2.62	3.2	1.17	1.58	2
FANOUT /Call	0.56	0.62	0.68	0.20	0.34	0.48
ANDC	0.25	0.41	0.57	0.19	0.28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

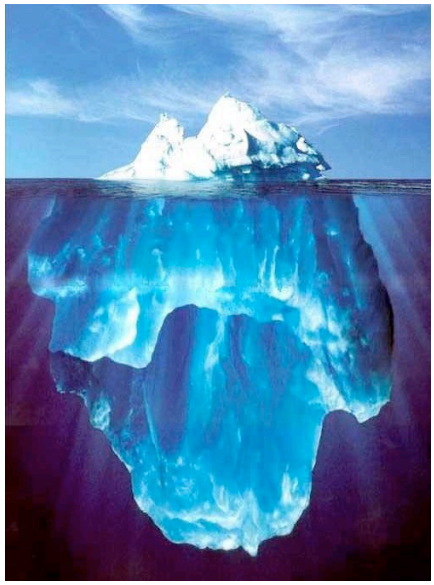


See(k)ing to understand

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

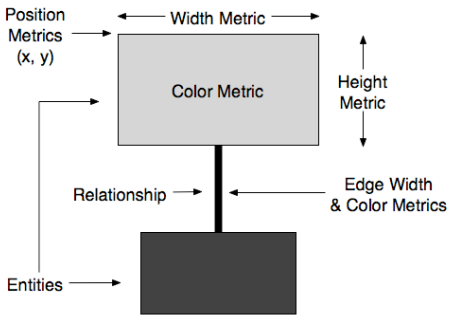
See(k)ing to understand

- The Overview Pyramid allows us to characterize the design of a system
- But…we need to see what we are talking about

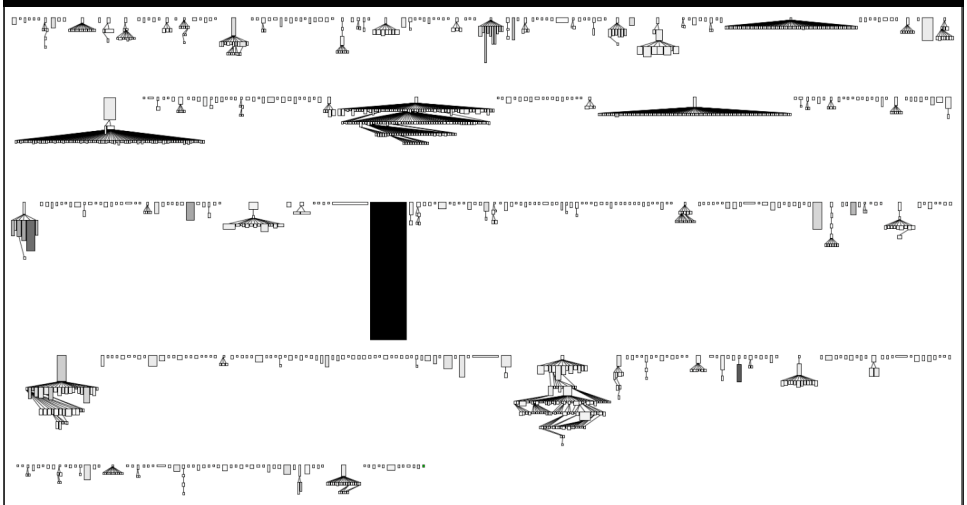


Polymetric Views

- 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



Polymetric View Example: ArgoUML

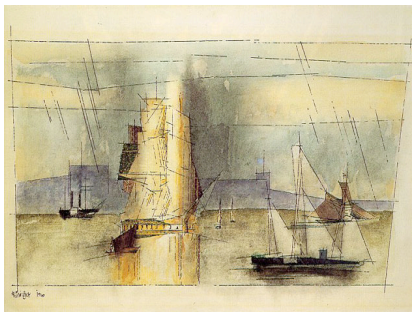


Reflections on Visualization

- 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”]

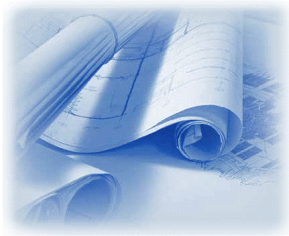
Evaluating the Design of a System

- 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?



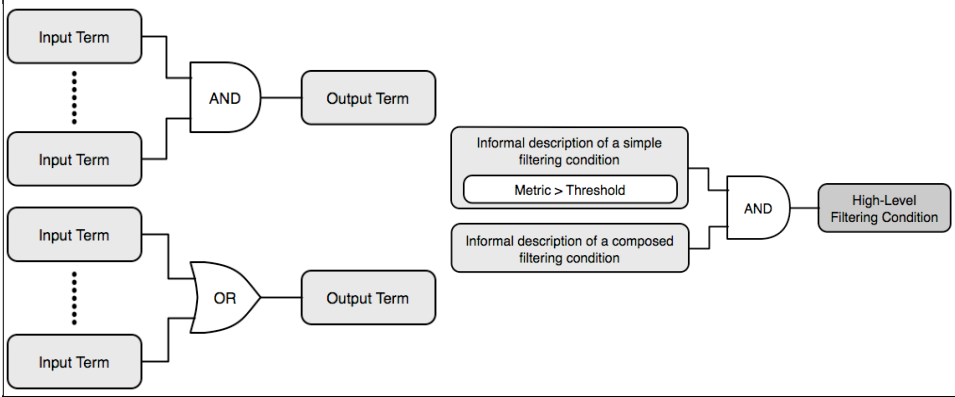
Professional Context

- 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 it?
  - Detection Strategies
  - The Class Blueprint



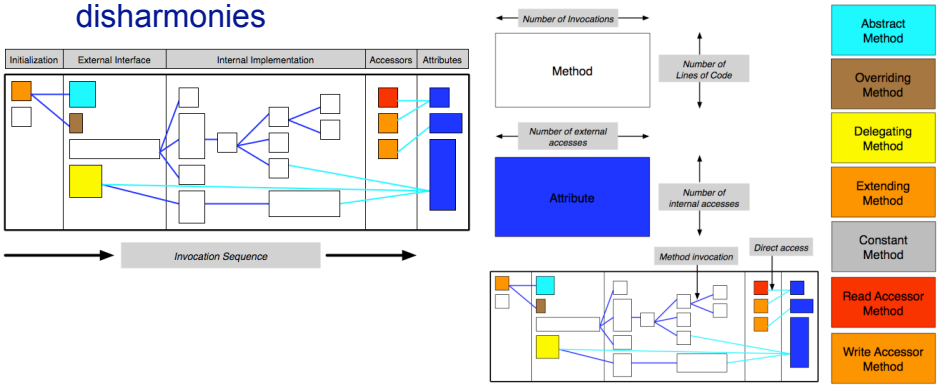
Detection Strategies

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

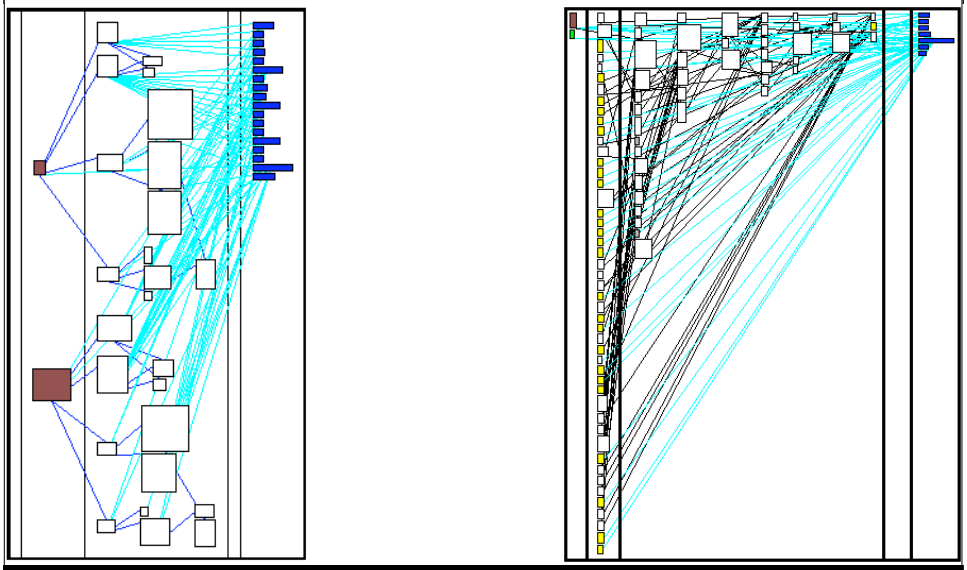


The Class Blueprint

- 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



Nice! …but, what about the practice?

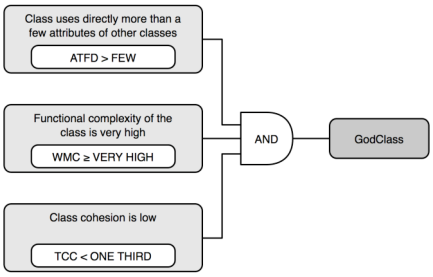
- In practice the key question is ***where to start***
- We have devised a methodology to ***characterize***, ***evaluate*** and ***improve*** the design of object-oriented systems
- It is based on:
  - The Overview Pyramid
  - The System Complexity View
  - Detection Strategies
  - Class Blueprints

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

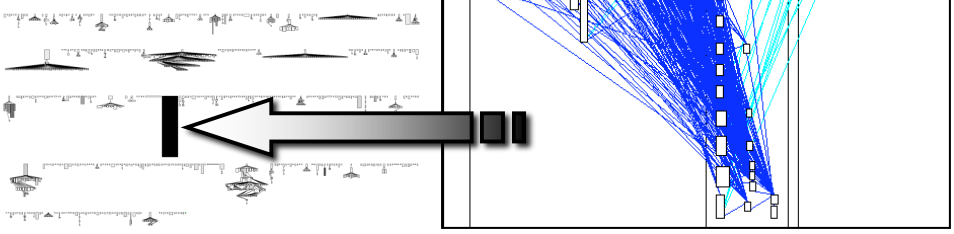
Identity Disharmony: God Class

- A God class is 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



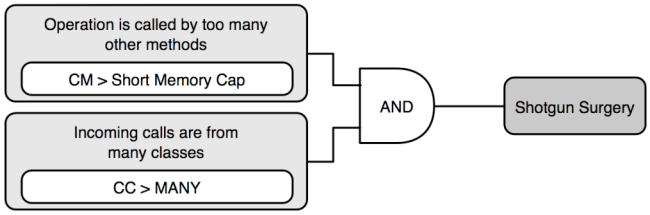
Oh my God…it's the ModelFacade

- ModelFacade: The Black Hole
  - 453 methods
  - 114 attributes
  - 3500 lines of code
- Coupled to hundreds of ArgoUML classes



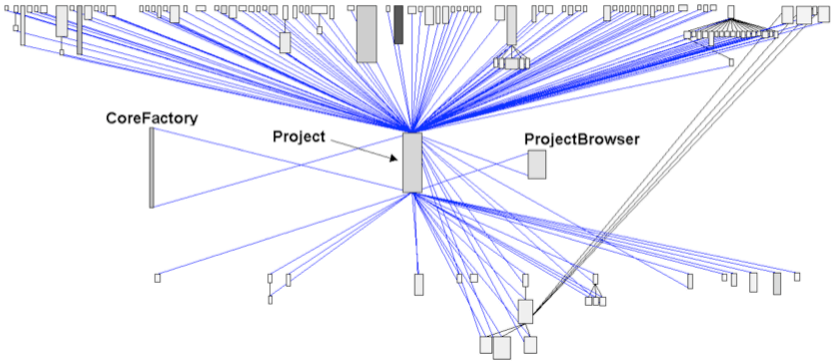
Collaboration Disharmony: Shotgun Surgery

- A change in a method may imply many 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



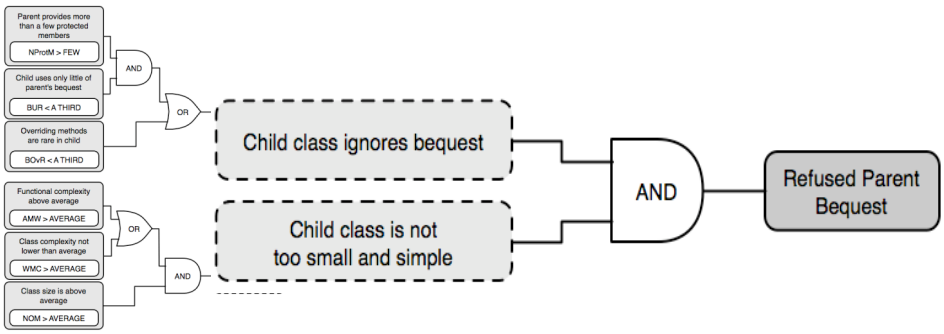
I shot…the Project…

- 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



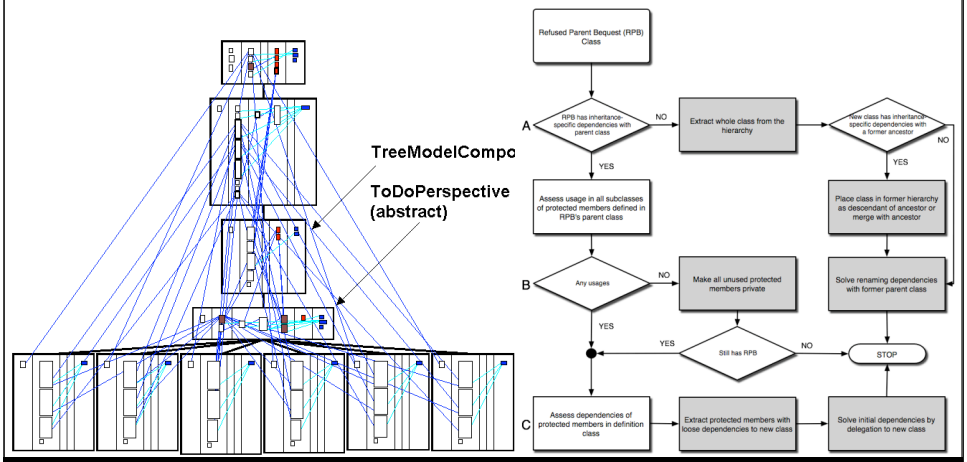
Classification Disharmony: Refused Parent Bequest

- 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)



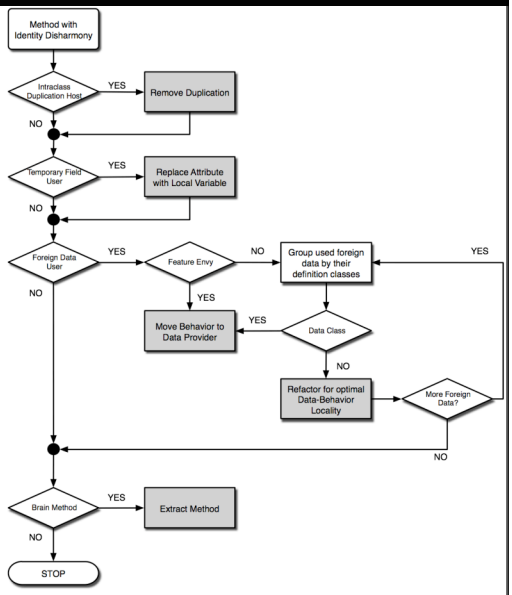
Kids never listen: The PerspectiveSupport Hierarchy

- "Pipeline"-Inheritance with funky usage of abstract classes
- Suspicious regularity in the leaf classes: duplicated code
- TreeModelComposite ignores what is the superclasses



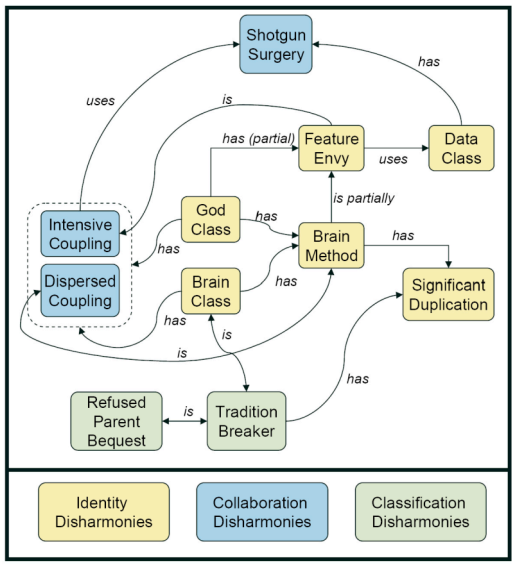
Recovering from a Design Disharmony

- 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

The End - Questions?

- References
  - M. Lanza, R. Marinescu; “Object-Oriented Metrics in Practice”; Springer 2006; ISBN: 3-540-24429-8
  - M. Lanza, S. Ducasse; “Polymetric Views - A Lightweight Visual Approach to Reverse Engineering”; In IEEE Transactions on Software Engineering, Vol. 29, No. 9, pp. 782 - 795, IEEE CS Press 2003.
  - M. Lanza, S. Ducasse; “A Categorization of Classes based on the Visualization of their Internal Structure: The Class Blueprint”; In Proceedings of OOPSLA 2001, pp. 300 - 311, ACM Press 2001.
  - R. Marinescu; “Detection Strategies: Metrics-Based Rules for Detecting Design Flaws”; In Proceedings of ICSM 2004, pp. 350 - 359; IEEE CS Press.