UNIVERSITÄT BERN

Introduction to Software Engineering

12. Software Metrics

Roadmap

- What are metrics? Why do we need them?
- Metrics for cost estimation
- Metrics for software quality evaluation
- Object-Oriented metrics in practice

Sources

> *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.

> *Software Metrics: A Rigorous & Practical Approach*, Norman E. Fenton, Shari l. Pfleeger, Thompson Computer Press, 1996.

Roadmap

> **What are metrics? Why do we need them?**
> Metrics for cost estimation
> Metrics for software quality evaluation
> Object-Oriented metrics in practice

Why Metrics?

When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science.

— Lord Kelvin

Why Measure Software?

Estimate cost and effort	measure correlation between specifications and final product
Improve productivity	measure value and cost of software
Improve software quality	measure usability, efficiency, maintainability ...
Improve reliability	measure mean time to failure, etc.
Evaluate methods and tools	measure productivity, quality, reliability ...

“You cannot control what you cannot measure” — De Marco, 1982
“What is not measurable, make measurable” — Galileo

What are Software Metrics?

Software metrics

> Any type of measurement which relates to a software system, process or related documentation

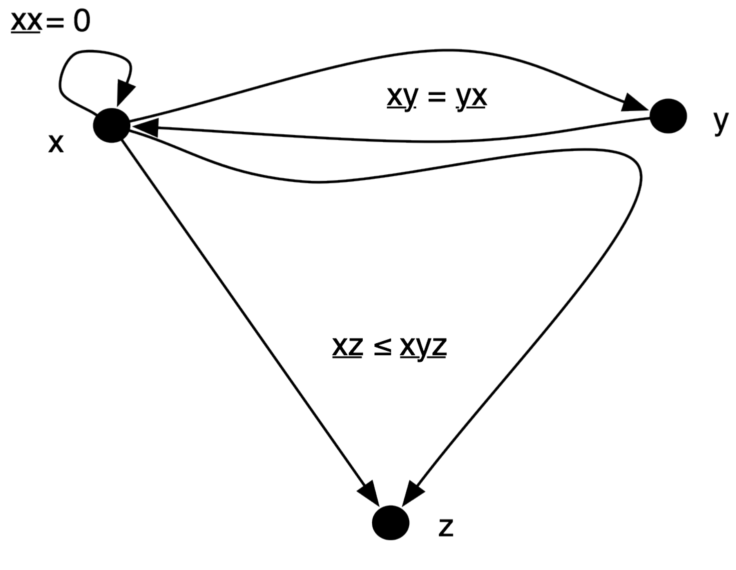
- Lines of code in a program
- the Fog index (calculates readability of a piece of documentation)

  $0.4 \times$ (# words / # sentences) + (% words ≥ 3 syllables)
- number of person-days required to implement a use-case

(Measures vs Metrics)

Mathematically, a metric is a function m measuring the *distance between two objects* such that:

1. $\forall$ x, m(x,x) = 0
2. $\forall$ x, y, m(x,y) = m(y,x)
3. $\forall$ x, y, z, m(x,z) $\leq$ m(x,y) + m(y,z)



So, technically "software metrics" is an abuse of terminology, and we should instead talk about "software measures".

Direct and Indirect Measures

Direct Measures

> *Measured* directly in terms of the observed attribute (usually by counting)
> — Length of source-code, Duration of process, Number of defects discovered

Indirect Measures

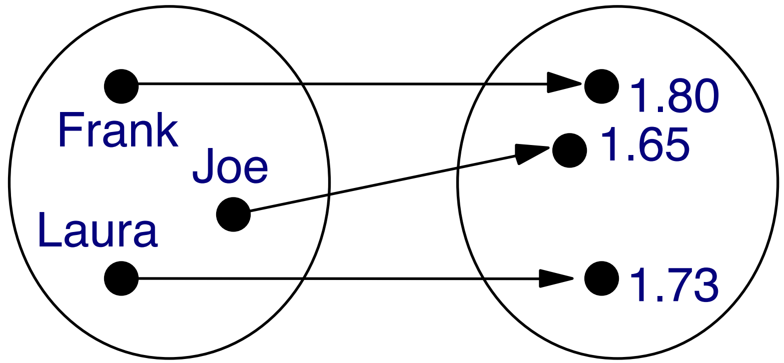
> *Calculated* from other direct and indirect measures
> — Module Defect Density = Number of defects discovered / Length of source
> — Temperature (usually derived from the length of a liquid column)

Measurement Mapping

Measure & Measurement

A measure is a function mapping an *attribute* of a real world entity (= the domain) onto a *symbol* in a set with known mathematical relations (= the range).

A measurement is the symbol assigned to the real world attribute by the measure.



Example: measure mapping height attribute of person on a number representing height in meters.

Purpose: *Manipulate symbol(s) in the range to draw conclusions about attribute(s) in the domain*

Preciseness

To be precise, the definition of the measure must specify:

- ***domain:*** do we measure people's height or width?
- ***range:*** do we measure height in centimetres or inches?
- ***mapping rules:*** do we allow shoes to be worn?

Possible Problems

Example: Compare productivity in lines of code per time unit.

Do we use the same units to compare?	What is a “line of code”? What is the “time unit”?
Is the context the same?	Were programmers familiar with the language?
Is “code size” really what we want to produce?	What about code quality?
How do we want to interpret results?	Average productivity of a programmer? Programmer X is twice as productive as Y?
What do we want to do with the results?	Do you reward “productive” programmers? Do you compare productivity of software processes?

GQM

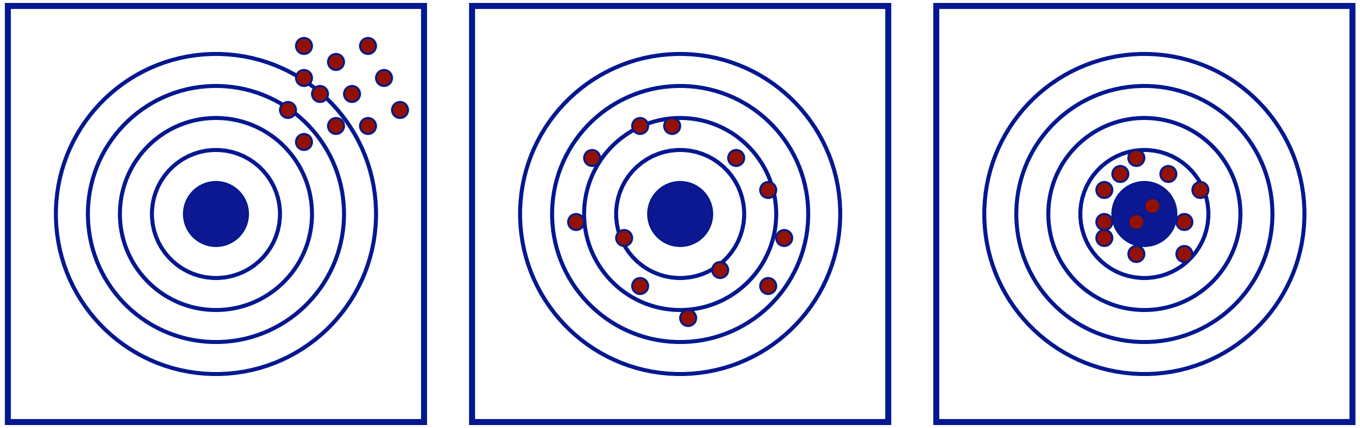
Goal — Question — Metrics approach

[Basili et al. 1984]

> Define *Goal*
  — e.g., “How effective is the coding standard XYZ?”
> Break down into *Questions*
  — “Who is using XYZ?”
  — “What is productivity/quality with/without XYZ?”
> Pick suitable *Metrics*
  — Proportion of developers using XYZ
  — Their experience with XYZ ...
  — Resulting code size, complexity, robustness ...

Validity and reliability

> A good metric is both valid *(measures what it is intended to measure)* and reliable *(yields consistent results)*



Reliable but not valid

Valid but not reliable

Valid and reliable

See: Stephen H. Kan, *Metrics and Models in Software Quality Engineering*, Addison Wesley, 2002. Ch. 3.4

Some Desirable Properties of Metrics

- Valid and reliable (consistent)
- Objective, precise
- Intuitive
- Robust (failure-tolerant)
- Automatable and economical (practical)
- …

Caveat: Attempts to define formally desirable properties have been heavily disputed …

See: 2.Brian Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity, Prentice-Hall, 1996, Ch. 2.6*

Weyuker's Axioms for Software Complexity Measures

1. At least two programs have different measures
2. Only finitely many programs have the same complexity
3. There exist multiple programs with the same complexity
4. There exist equivalent programs with different complexity
5. The complexity of P is less than or equal to an extension of P
6. …

Weyuker. *Evaluating Software Complexity Measures.* IEEE TSE 14(9) p. 1357—1365, 1988

But, also useless metrics satisfy these properties!

"in practice, almost any complexity measure can be successfully used in software development"

Cherniavsky, Smith. *On Weyuker's Axioms for Software Complexity Measures.* IEEE TSE 17 p. 636—638, June 1991

Roadmap

- > What are metrics? Why do we need them?
- > **Metrics for cost estimation**
- > Metrics for software quality evaluation
- > Object-Oriented metrics in practice

Cost estimation objectives

Cost estimation and planning/scheduling are closely related activities

Goals

> To establish a budget for a software project
> To provide a means of controlling project costs
> To monitor progress against the budget
  — comparing planned with estimated costs
> To establish a cost database for future estimation

Estimation techniques

Expert judgement	cheap, but risky!
Estimation by analogy	limited applicability
Parkinson's Law	unlimited risk!
Pricing to win	i.e., you do what you can with the money
Top-down estimation	may miss low-level problems
Bottom-up estimation	may underestimate integration costs
Algorithmic cost modelling	requires correlation data

Each method has strengths and weaknesses!

Estimation should be based on several methods

Algorithmic cost modelling

- Cost is estimated as a *mathematical function of product, project and process attributes* whose values are estimated by project managers
- The function is derived from a study of *historical costing data*
- Most commonly used product attribute for cost estimation is *LOC* (code size)
- Most models are basically similar but with different attribute values

Measurement-based estimation

A. Measure

Develop a *system model* and measure its size

B. Estimate

Determine the effort with respect to an *empirical database* of measurements from *similar projects*

C. Interpret

Adapt the effort with respect to a specific *Development Project Plan*

Lines of code

Lines of Code as a measure of system size?

> Easy to measure; but *not well-defined* for modern languages
  — *What's a line of code?*

> A *poor indicator of productivity*
  — Ignores software reuse, code duplication, benefits of redesign
  — The lower level the language, the more productive the programmer!
  — The more verbose the programmer, the higher the productivity!

Function points

Function Points (Albrecht, 1979)

- Based on a combination of program characteristics:
  - external inputs and outputs
  - user interactions
  - external interfaces
  - files used by the system
- A weight is associated with each of these
- The function point count is computed by multiplying each raw count by the weight and summing all values
- Function point count modified by complexity of the project

Function points

Good points, bad points

> Can be measured already after design
> FPs can be used to estimate LOC depending on the average number of LOC per FP for a given language
> LOC can vary wildly in relation to FP
> FPs are very subjective — depend on the estimator. They cannot be counted automatically

Programmer productivity

A measure of the rate at which individual engineers involved in software development produce software and associated documentation

Productivity metrics

> Size-related measures based on some output from the software process. This may be lines of delivered source code, object code instructions, etc.

> Function-related measures based on an estimate of the functionality of the delivered software. Function-points are the best known of this type of measure

…

Programmer productivity ...

Productivity estimates

> Real-time embedded systems, 40-160 LOC/P-month
> Systems programs , 150-400 LOC/P-month
> Commercial applications, 200-800 LOC/P-month

Quality and productivity

> *All metrics based on volume/unit time are flawed because they do not take quality into account*
  — Productivity may generally be increased at the cost of quality
  — It is not clear how productivity/quality metrics are related

The COCOMO model

> Developed at TRW, a US defence contractor

> Based on a *cost database* of more than 60 different projects

> Exists in three stages
1. ***Basic*** — Gives a “ball-park” estimate based on product attributes
2. ***Intermediate*** — Modifies basic estimate using project and process attributes
3. ***Advanced*** — Estimates project phases and parts separately

Basic COCOMO Formula

> $\text{Effort} = C \times PM^{S} \times M$
  - Effort is measured in person-months
  - C is a *complexity factor*
  - PM is a product metric (size or functionality, usually KLOC)
  - exponent S is close to 1, but increasing for large projects
  - M is a multiplier based on process, product and development attributes (~ 1)

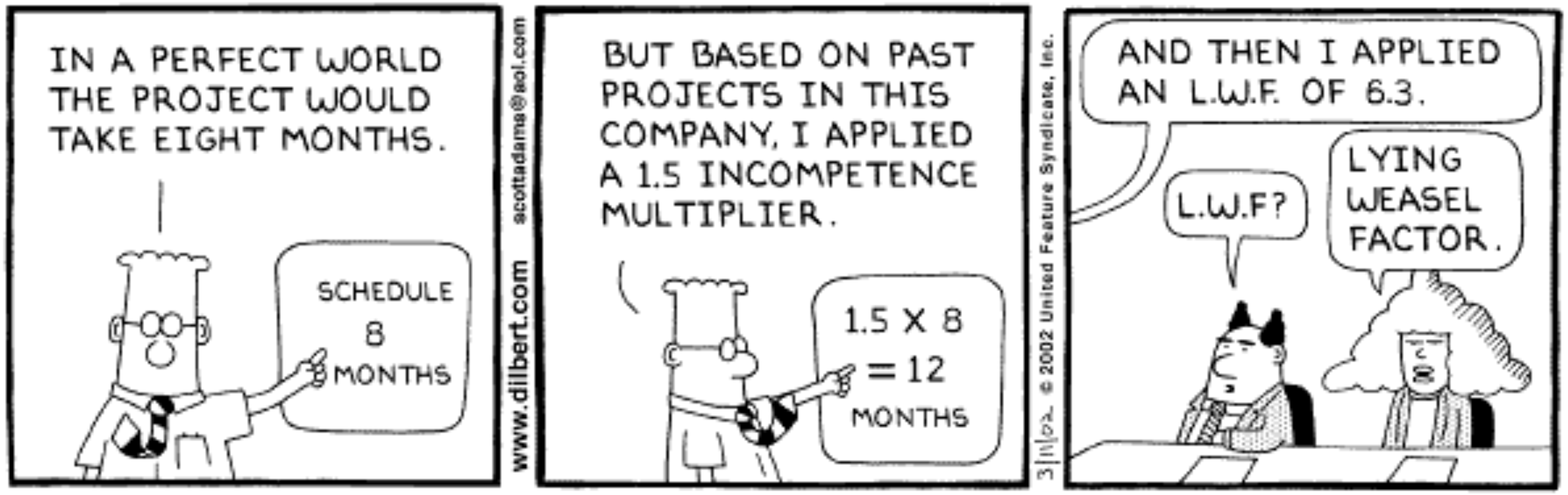
COCOMO Project classes

Organic mode: *small teams*, familiar environment, *well-understood applications*, no difficult non-functional requirements (EASY)	*Effort* $= 2.4\ (KDSI)^{1.05} \times M$
Semi-detached mode: Project team may have *experience mixture*, system may have more *significant non-functional constraints*, organization may have less familiarity with application (HARDER)	*Effort* $= 3\ (KDSI)^{1.12} \times M$
Embedded: *Hardware/software systems*, *tight constraints*, unusual for team to have deep application experience (HARD)	*Effort* $= 3.6\ (KDSI)^{1.2} \times M$

KDSI = Kilo Delivered Source Instructions

COCOMO assumptions and problems

- Implicit productivity estimate
  - — Organic mode = 16 LOC/day
  - — Embedded mode = 4 LOC/day
- Time required is a function of total effort *not team size*
- Not clear how to adapt model to *personnel availability*



http://en.wikipedia.org/wiki/Fair_use

COCOMO assumptions and problems ...

- *Staff required* can’t be computed by dividing the development time by the required schedule
- The number of people working on a project varies depending on the *phase of the project*
- The more people who work on the project, the *more total effort* is usually required (!)
- Very *rapid build-up* of people often correlates with *schedule slippage*

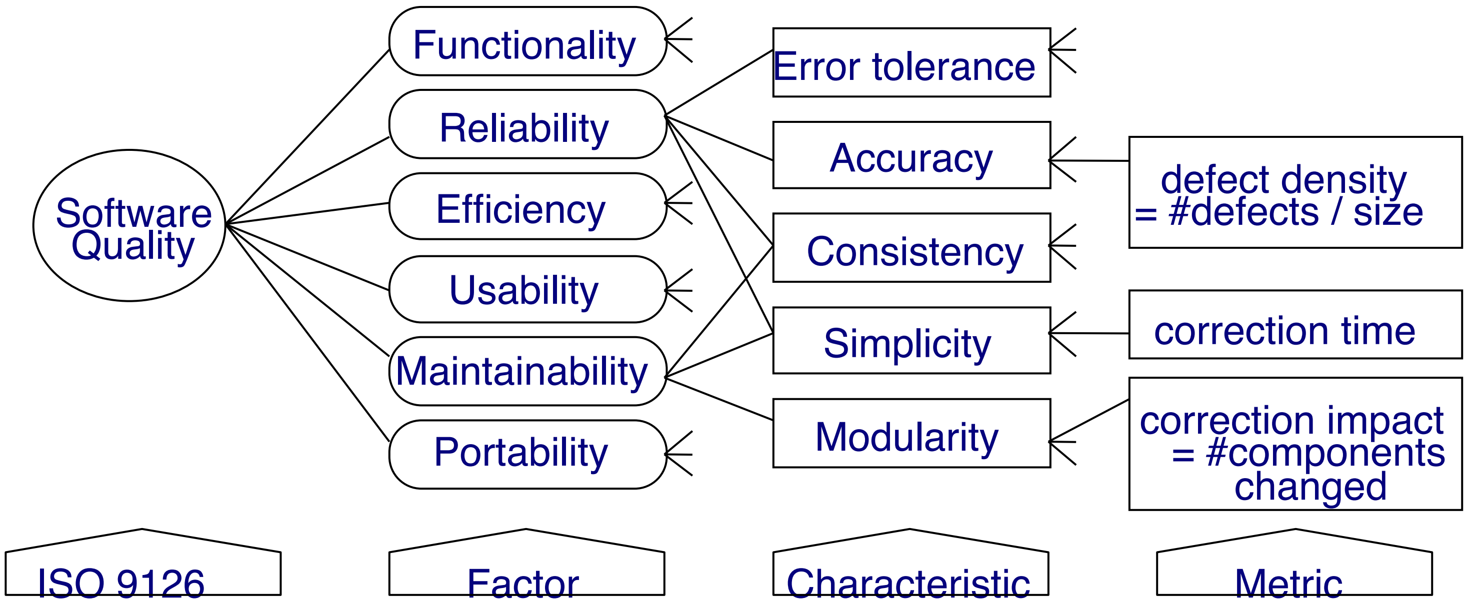
Roadmap

- > What are metrics? Why do we need them?
- > Metrics for cost estimation
- > **Metrics for software quality evaluation**
- > Object-Oriented metrics in practice

Quantitative Quality Model

Quality according to ISO 9126 standard

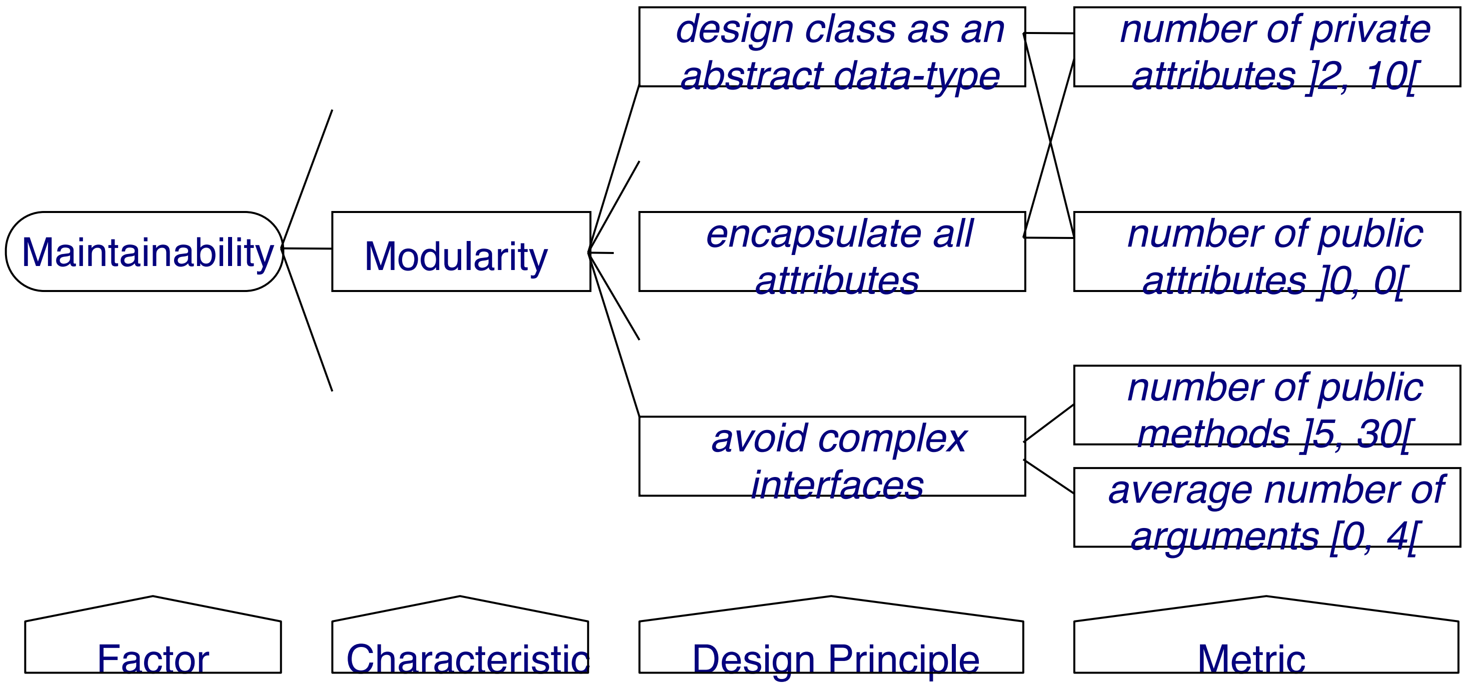
- Divide-and conquer approach via “hierarchical quality model”
- Leaves are simple metrics, measuring basic attributes



“Define your own” Quality Model

Define the quality model with the development team

> Team chooses the characteristics, design principles, metrics ... and the thresholds



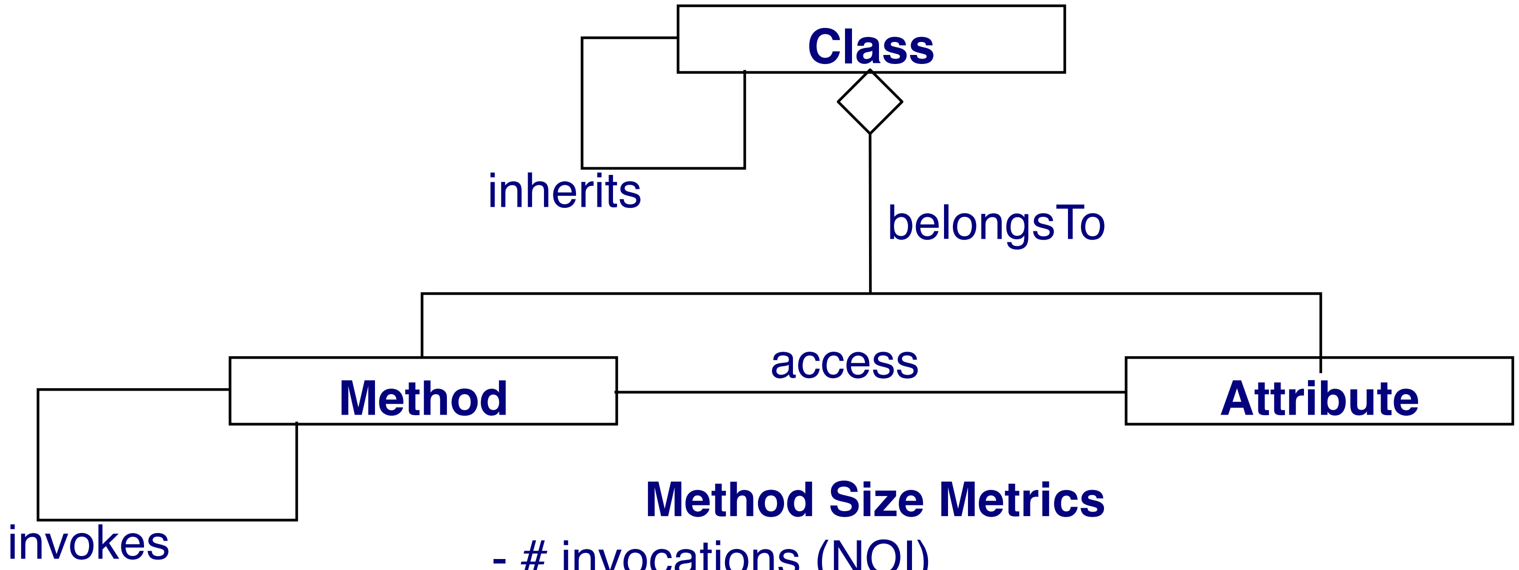
Sample Size (and Inheritance) Metrics

Inheritance Metrics

- hierarchy nesting level (HNL)
- # immediate children (NOC)
- # inherited methods, unmodified (NMI)
- #overridden methods (NMO)

Class Size Metrics

- # methods (NOM)
- # attributes, instance/class (NIA, NCA)
- # S of method size (WMC)



Method Size Metrics

- # invocations (NOI)
- # statements (NOS)
- # lines of code (LOC)
- # arguments (NOA)

Sample Coupling & Cohesion Metrics

The following definitions stem from [Chid91a], later republished as [Chid94a]

Coupling Between Objects (CBO)

CBO = number of other classes to which given class is coupled

Interpret as “number of other classes a class requires to compile”

Lack of Cohesion in Methods (LCOM)

LCOM = number of disjoint sets (= not accessing same attribute) of local methods

Coupling & Cohesion Metrics

Beware!

Researchers disagree whether coupling/cohesion methods are valid

> Classes that are observed to be cohesive may have a high LCOM value
  - — due to accessor methods

> Classes that are not much coupled may have high CBO value
  - — no distinction between data, method or inheritance coupling

Sample Quality Metrics (I)

Productivity (Process Metric)

- functionality / time
- functionality in LOC or FP; time in hours, weeks, months
  - be careful to compare: the same unit does not always represent the same
- Does not take into account the quality of the functionality!

Sample Quality Metrics (II)

Reliability (Product Metric)

- mean time to failure = mean of probability density function PDF
  - for software one must take into account the fact that repairs will influence the rest of the function ⇒ quite complicated formulas
- average time between failures = # failures / time
  - time in execution time or calendar time
  - necessary to calibrate the probability density function
- mean time between failure = MTTF + mean time to repair
  - to know when your system will be available, take into account repair

Sample Quality Metrics (III)

Correctness (Product Metric)

> “a system is correct or not, so one cannot measure correctness”

> defect density = # known defects / product size
  - product size in LOC or FP
  - # known defects is a time based count!

> *do not compare across projects* unless your data collection is sound!

Sample Quality Metrics (IV)

Maintainability (Product Metric)

> #time to repair certain categories of changes
> “mean time to repair” vs. “average time to repair”
>   — similar to “mean time to failure” and “average time between failures”
> beware of the units
>   — “categories of changes” is subjective
>   — time =?
>   problem recognition time + administrative delay time + problem analysis time + change time + testing & reviewing time

Roadmap

- What are metrics? Why do we need them?
- Metrics for cost estimation
- Metrics for software quality evaluation
- **Object-Oriented metrics in practice**

Michele Lanza and Radu Marinescu, *Object-Oriented Metrics in Practice*, Springer-Verlag, 2006

Pattern: Study the Exceptional Entities

Problem

— **How can you quickly gain insight into complex software?**

Solution

— *Measure* software entities and *study the anomalous ones*

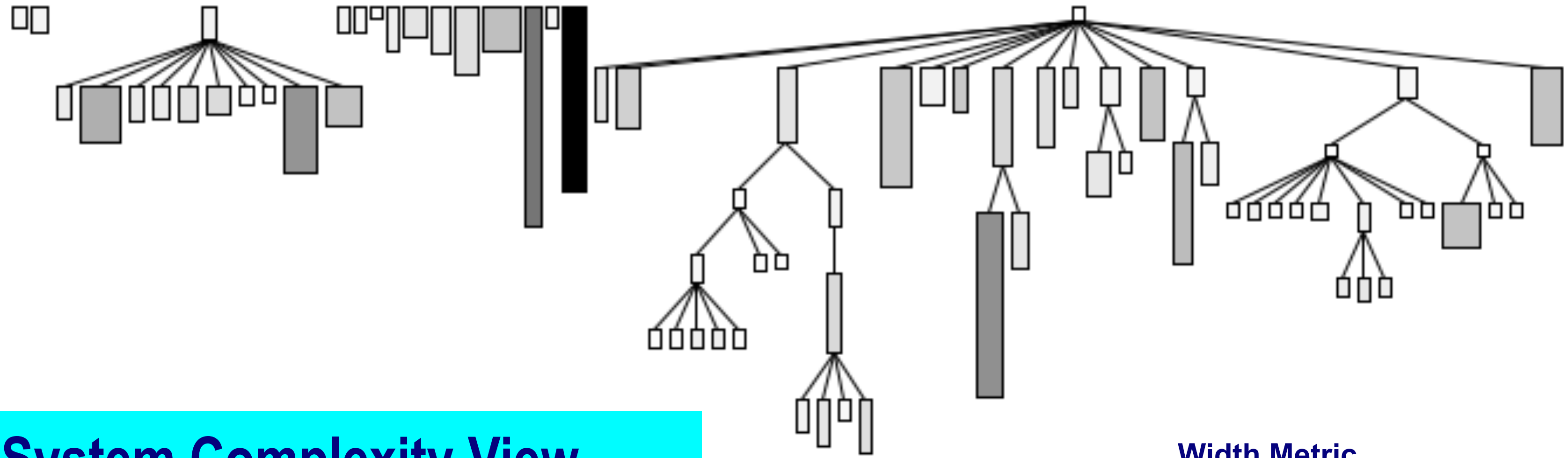
Steps

— Use simple metrics

— Visualize metrics to get an overview

— Browse the code to get insight into the anomalies

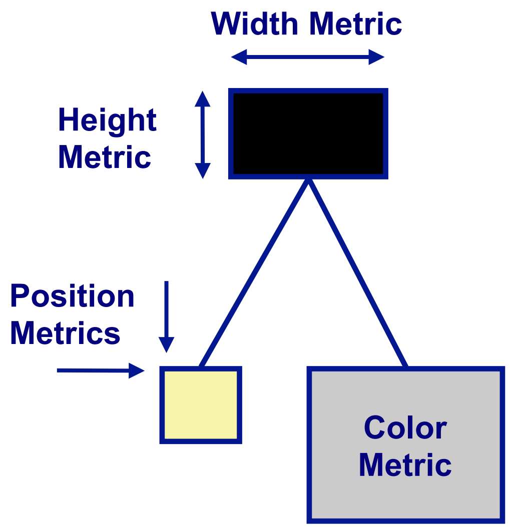
System Complexity View



System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

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



Detection strategy

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

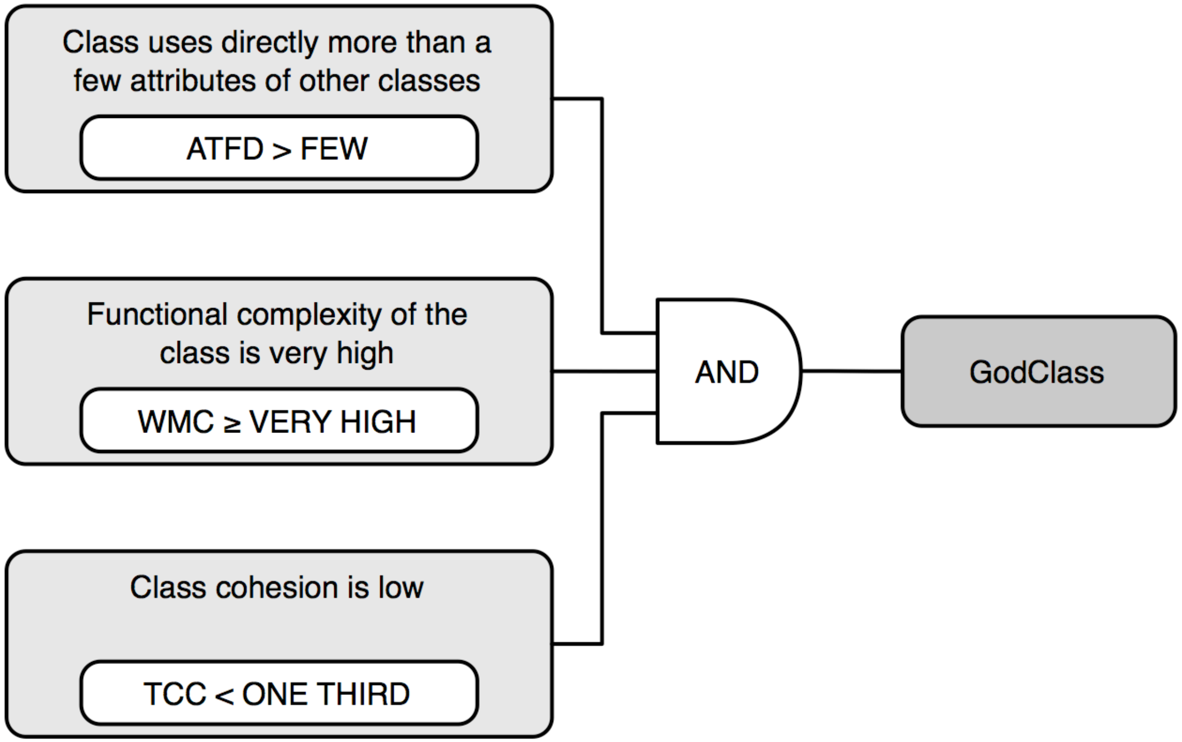
Filters and composition

- A data filter is a predicate used to focus attention on a *subset of interest* of a larger data set
  - Statistical filters
    - *I.e., top and bottom 25% are considered outliers*
  - Other relative thresholds
    - *I.e., other percentages to identify outliers (e.g., top 10%)*
  - Absolute thresholds
    - *I.e., fixed criteria, independent of the data set*

- A useful detection strategy can often be expressed as a *composition* of data filters

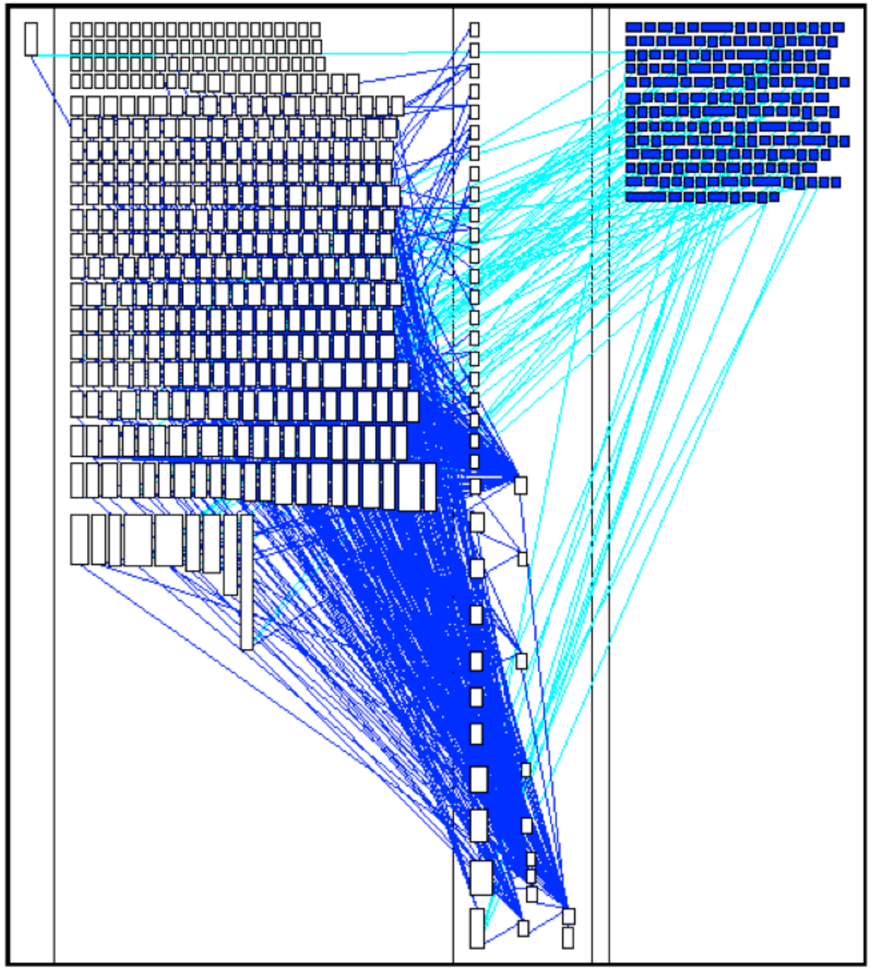
God Class

> A *God Class* centralizes intelligence in the system
- Impacts understandibility
- Increases system fragility



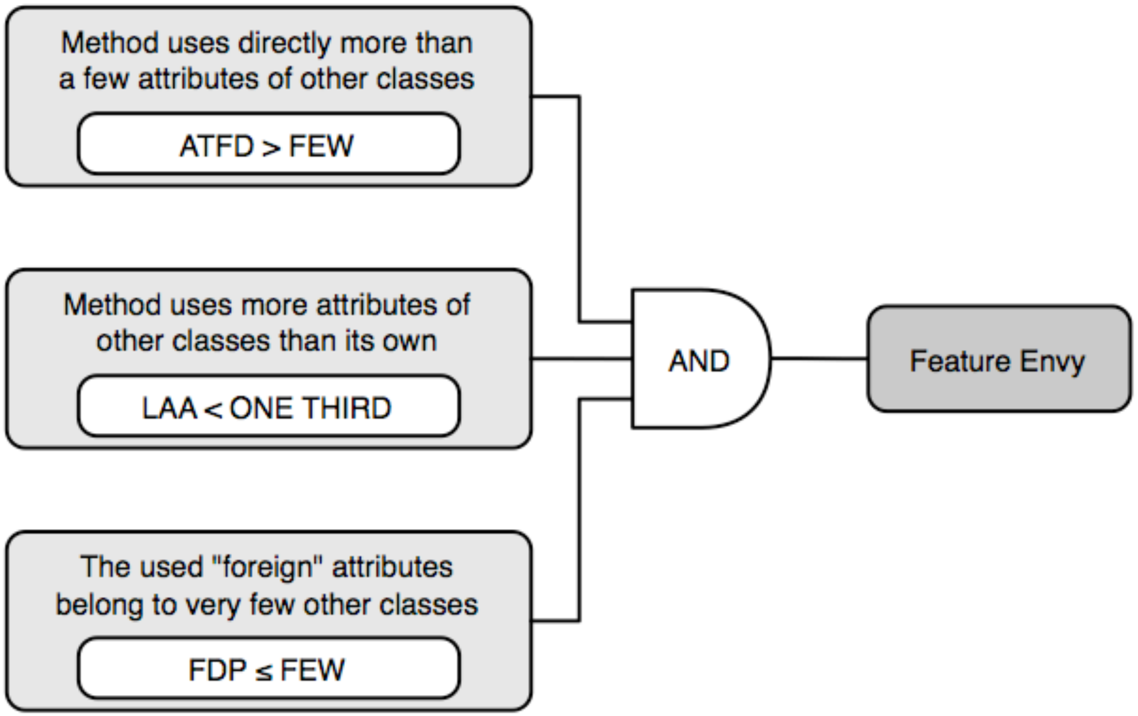
ModelFacade (ArgoUML)

> 453 methods
> 114 attributes
> over 3500 LOC
> all methods and all attributes are static

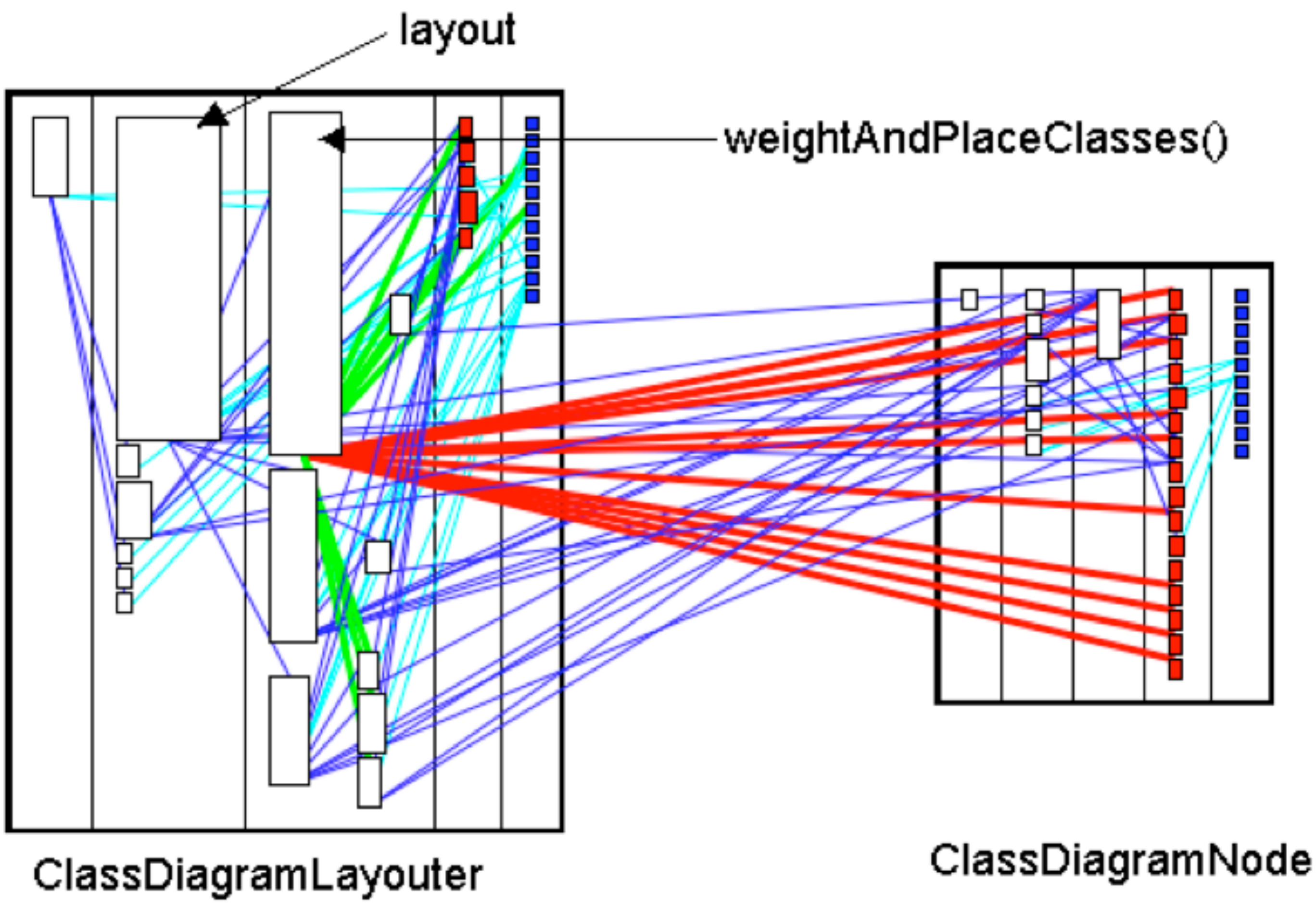


Feature Envy

> Methods that are more interested in data of other classes than their own [Fowler et al. 99]

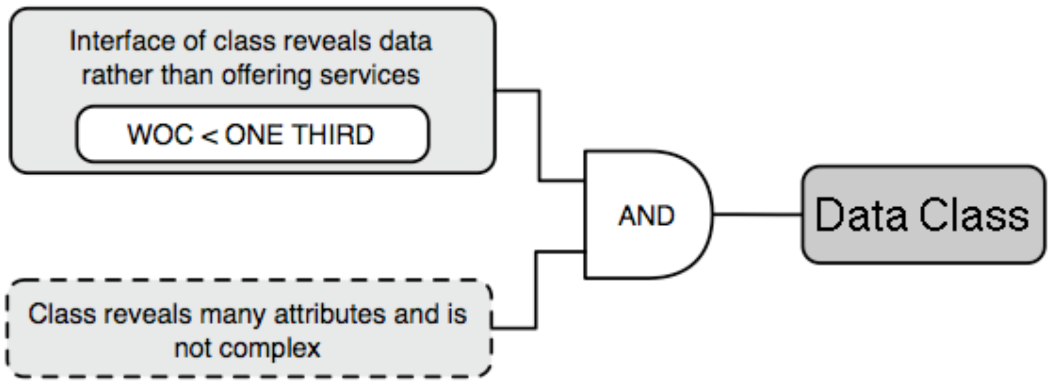


ClassDiagramLayouter

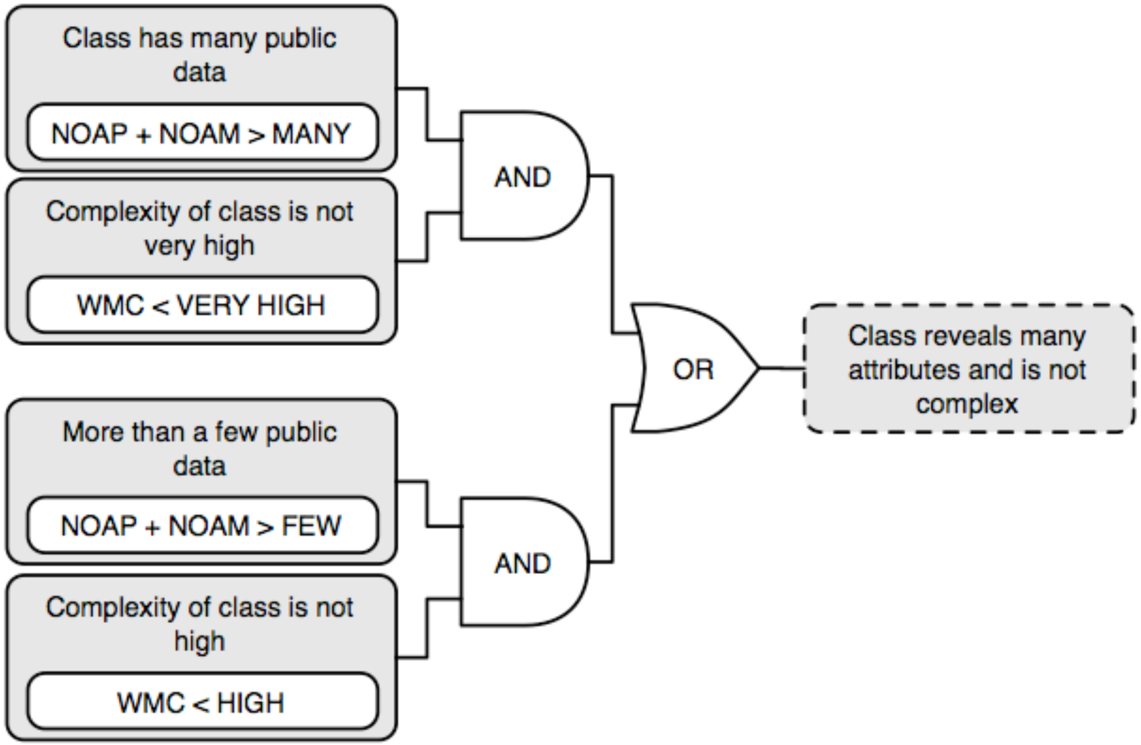


Data Class

> A Data Class provides data to other classes but little or no functionality of its own

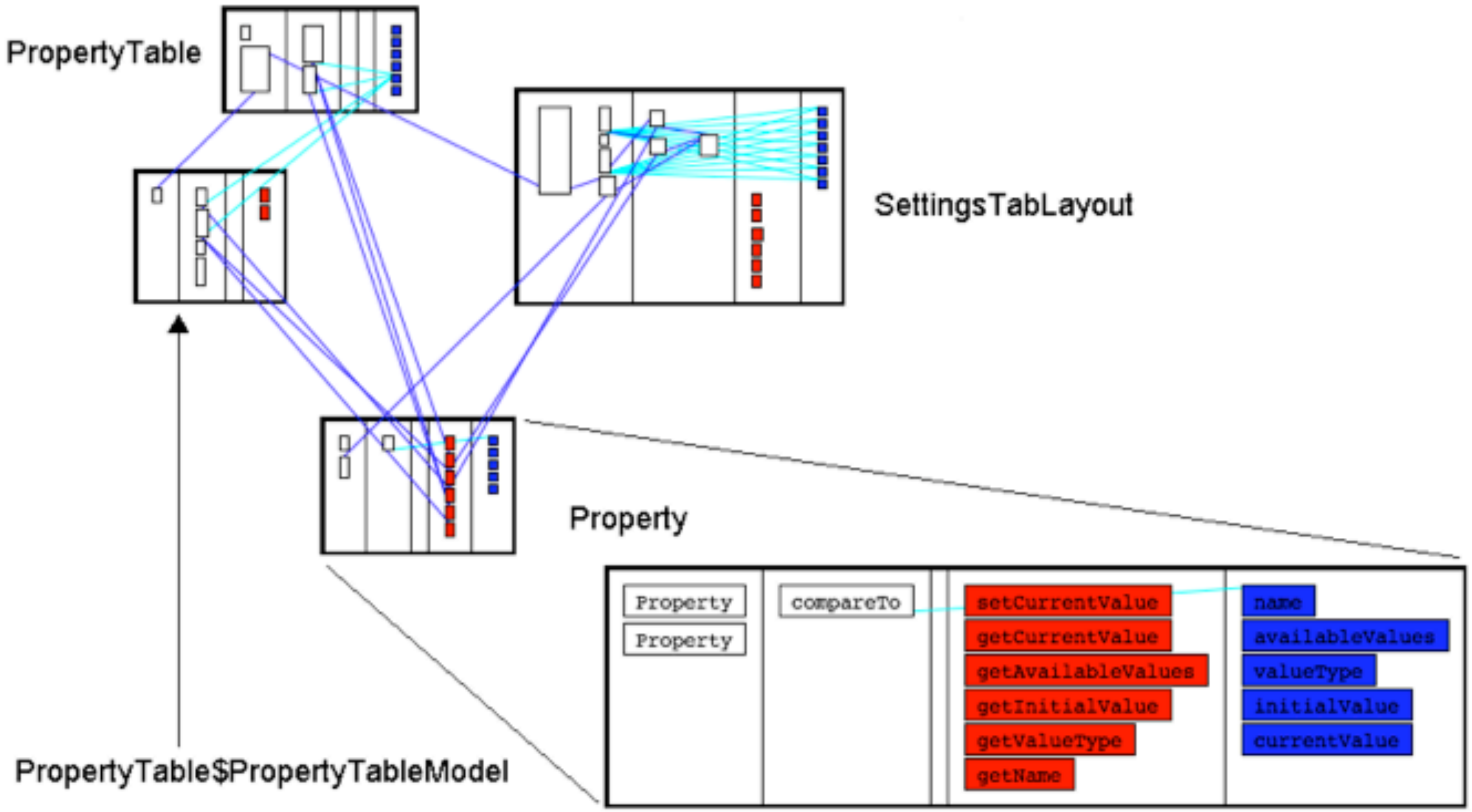


Data Class (2)



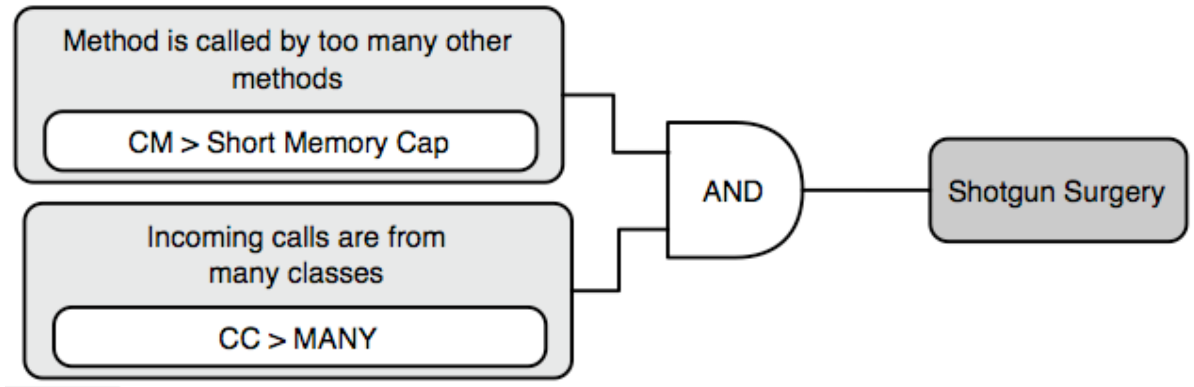
Class has many public data
NOAP + NOAM > MANY
Complexity of class is not very high
WMC < VERY HIGH
More than a few public data
NOAP + NOAM > FEW
Complexity of class is not high
WMC < HIGH
AND
AND
OR
Class reveals many attributes and is not complex

Property

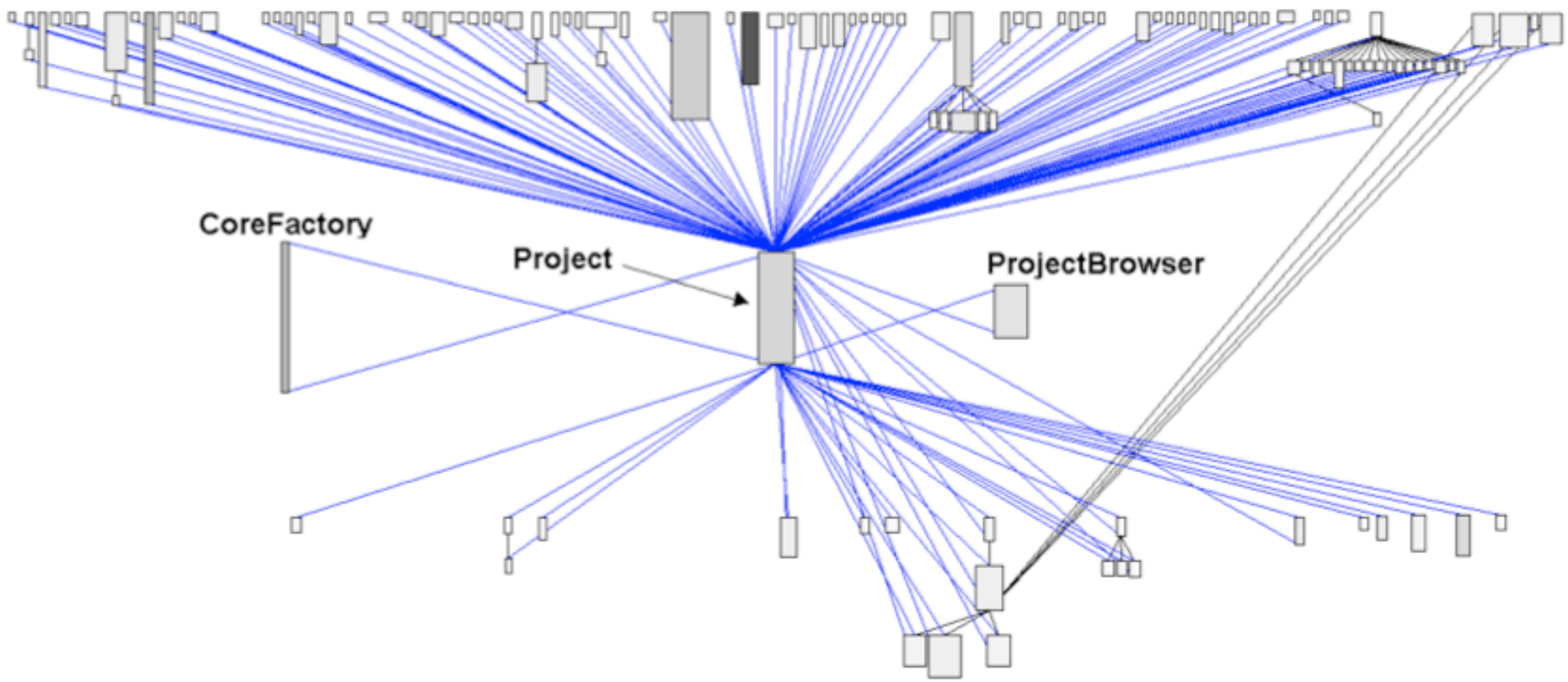

PropertyTable
SettingsTabLayout
Property
PropertyTable$PropertyTableModel
Property
Property
compareTo
setCurrentValue
getCurrentValue
getAvailableValues
getInitialValue
getValueType
getName
name
availableValues
valueType
initialValue
currentValue

Shotgun Surgery

> A change in an operation implies many (small) changes to a lot of different operations and classes



Project



What you should know!

> What is a measure? What is a metric?
> What is GQM?
> What are the three phases of algorithmic cost modelling?
> What problems arise when using LOC as a software metric?
> What are the key ideas behind COCOMO?
> What's the difference between "Mean time to failure" and "Average time between failures"? Why is the difference important?

Can you answer the following questions?

> During which phases in a software project would you use metrics?
> Is the Fog index a “good” metric?
> How would you measure your own software productivity?
> Why are coupling/cohesion metrics important? Why then are they so rarely used?

License