Introduction to Software Engineering

11. 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
> **Software Engineering**. Ian Sommerville. Addison-Wesley, 10th edition, 2015

Roadmap

> What are metrics? Why do we need them?
> Metrics for cost estimation
> Metrics for software quality evaluation
> Object-Oriented metrics in practice
I often say that *when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, 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, whatever the matter may be.*

— Lord Kelvin
### Why Measure Software?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Estimate cost and effort</strong></td>
<td>measure correlation between specifications and final product</td>
</tr>
<tr>
<td><strong>Improve productivity</strong></td>
<td>measure value and cost of software</td>
</tr>
<tr>
<td><strong>Improve software quality</strong></td>
<td>measure usability, efficiency, maintainability ...</td>
</tr>
<tr>
<td><strong>Improve reliability</strong></td>
<td>measure mean time to failure, etc.</td>
</tr>
<tr>
<td><strong>Evaluate methods and tools</strong></td>
<td>measure productivity, quality, reliability ...</td>
</tr>
</tbody>
</table>

“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)

\[ 0.4 \times \left( \frac{\text{# words}}{\text{# sentences}} \right) + \left( \% \text{ words } \geq 3 \text{ syllables} \right) \]

— number of person-days required to implement a use-case
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”.
The term “metric” in mathematics refers to a distance function. Software “metrics” rarely measure “distances” however. The term is therefore somewhat unfortunate, but it is so widely used that it is unlikely to be displaced.

We will try to stick to the term “measure” and only use “metric” in reference to distance function.
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
We measure things into order to build a simple model of them, allowing us to reason about their properties. For example, we may measure the dimensions of things to allow us to determine if they will fit into a certain space without having to carry out a physical experiment.
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?
A great difficulty arises if we do not apply consistent rules to measure things. If you and I independently measure the sizes of two software systems, comparison will be meaningless unless we agree on the rules for measuring. For example, do we count lines of code or statements? Do we include comments and blank lines, or do we remove them first?
## Possible Problems

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

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do we use the same units to compare?</td>
<td>What is a “line of code”? What is the “time unit”?</td>
</tr>
<tr>
<td>Is the context the same?</td>
<td>Were programmers familiar with the language?</td>
</tr>
<tr>
<td>Is “code size” really what we want to produce?</td>
<td>What about code quality?</td>
</tr>
<tr>
<td>How do we want to interpret results?</td>
<td>Average productivity of a programmer? Programmers X is twice as productive as Y?</td>
</tr>
<tr>
<td>What do we want to do with the results?</td>
<td>Do you reward “productive” programmers? Do you compare productivity of software processes?</td>
</tr>
</tbody>
</table>
Lines of Code (LOC) is the most notorious software measure for the difficulties it poses. It is inherently difficult to pin down, and comparisons between different languages are clearly dangerous. Equating productivity with LOC produced is also dangerous, as it rewards verbosity. If two programmers are given the same task, and the first one produces a 100-line program in 1 day, while the second produces an equivalent 500-line program in 2 days, which one should be considered more productive?

Similarly, if a third programmer spends 1 day refactoring the second program down to an equivalent 50 lines of code (i.e., -450 LOC produced), how should we measure her productivity? (Is it negative?)
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 ...
The “Goal, Question, Metric” (GQM) approach warns against collection of random or arbitrary measures.

Before collecting any data, one should first establish the goal one wishes to achieve. Is it to arrive at an understanding of some problem? Is it to optimize some process? Is it to make a choice amongst some alternatives?

Once the goal is established, determine a set of questions that would help you to achieve that goal.

Finally, select or develop measures that will help you to answer those questions. Never collect data or measurements without a clear idea how these data will be used.

https://en.wikipedia.org/wiki/GQM
Validity and reliability

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

If your watch is set 5 minutes late, it will be reliable, but not valid.

If your watch randomly loses or gains a few minutes every hour, it may be valid, but not reliable.
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 …
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. …

But, also useless metrics satisfy these properties!

“in practice, almost any complexity measure can be successfully used in software development”
Cherniavsky and Smith give an example of a nonsensical metric that satisfies Weyuker’s 9 properties.
The lesson to draw from this is that it is a difficult (probably impossible) task to define sensible constraints that software measures should fulfil. Instead, one should focus on the general GQM methodology.
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

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

Each method has strengths and weaknesses!
Estimation should be based on several methods
• **Expert judgement**: ask an expert. Do you trust the expert’s judgement?

• **Estimation by analogy**: “this project is kind of like another one, but …” How do you account for the differences?

• **Parkinson’s Law**: If “work expands to fill the time available”, risk is unlimited. See: [https://en.wikipedia.org/wiki/Parkinson%27s_law](https://en.wikipedia.org/wiki/Parkinson%27s_law)

• **Pricing to win**: this is very much in-line with Scrum and agile methods — given a particular budget, try to deliver the best value by interactively adjusting the backlog of desired features. This is not very helpful for cost estimation, however.

• **Top-down estimation**: break down complex tasks into simpler ones that are easier to estimate.

• **Bottom-up estimation**: just measuring the cost of individual subtasks may not take into account integration aspects.

• **Algorithmic cost modelling**: uses a historical database of costs of past projects, but this can only work if projects and their contexts are similar enough.
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
The first step requires that you develop some kind of *model* of the system to be built. NB: Since the system does not exist yet, you cannot directly measure LOC. A typical system model might be a first rough design or domain model, possibly expressed as a UML class diagram, but it really could be any kind of initial specification.

The second step requires that you have previously built up an *empirical database* of the effort spent on *similar projects* (i.e., similar domain, team, and technology). This allows you to look up the estimated system size and determine the estimated effort.

Finally you need to *adapt and adjust your estimate* depending on various factors (differences in team, in project, in technology etc.)
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!
What measure do we use to estimate effort? Typically we translate our system model into some kind of measure of size, such as LOC, but we have already seen that this must be done with great care.

What’s “a glass of wine”? What’s “a line of code”? 
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
The function point model was first described by Albrecht in 1979.
https://en.wikipedia.org/wiki/Function_point

The graphic is from the following article:
http://www.codeproject.com/Articles/18024/Calculating-Function-Points
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

...
Although it is fairly impossible to come up with a sensible measure of productivity, we need it for cost estimation.
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
Programmer productivity is known to vary wildly depending on various factors:

- application domain (systems programming is harder than web development)
- according to folklore, it is common to see 10x difference in productivity between programmers
- the language and technology used may drastically impact productivity

https://en.wikipedia.org/wiki/Programming_productivity
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
The Constructive Cost Model (COCOMO) was published in 1981 in a book by Barry Boehm.

https://en.wikipedia.org/wiki/COCOMO
Basic COCOMO Formula

> 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

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

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
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
**Review: Quantitative Quality Model**

**Quality according to ISO 9126 standard**
- Divide-and-conquer approach via “hierarchical quality model”
- Leaves are simple metrics, measuring basic attributes

ISO 9126

Software Quality

- Functionality
- Reliability
- Efficiency
- Usability
- Maintainability
- Portability

ISO 9126

Factor

Characteristic

Metric

- Error tolerance
- Accuracy
- Consistency
- Simplicity
- Modularity

- Defect density = #defects / size
- Correction time
- Correction impact = #components changed
“Define your own” Quality Model

Define the quality model with the development team

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

Maintainability

Modularity

- design class as an abstract data-type
  - number of private attributes: [2, 10]
- encapsulate all attributes
  - number of public attributes: [0, 0]
- avoid complex interfaces
  - number of public methods: [5, 30]
  - average number of arguments: [0, 4]
Sample Size (and Inheritance) Metrics

Class Size Metrics
- # methods (NOM)
- # attributes, instance/class (NIA, NCA)
- sum of method size (WMC)

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

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

**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
**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 (III)

**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
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
This pattern is from the open-source book, “Object-Oriented Reengineering Patterns”. Like design patterns, reengineering patterns encode knowledge mined from experience with practical problems in real software systems. Instead of encoding design experience, however, these patterns express how to reverse engineer and reengineer legacy software systems.

This particular pattern is useful when you want to obtain an initial overview of a complex object-oriented software system. Since the code base may be very large, it is not feasible to read even a small portion of the code. Instead, by applying simple metrics and visualizing the results, your attention may be drawn to anomalous outliers (i.e., exceptional entities) that will tell you interesting things about the system.

The book may be freely downloaded:

http://scg.unibe.ch/download/oorp/
Nodes = Classes
Edges = Inheritance Relationships
Simply visualizing the inheritance hierarchy of a software system will not give you much insight into the code.
System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

Width = Number of Attributes
Height = Number of Methods
Color = Number of Lines of Code
The System Complexity View is an example of a *polymetric view*: a simple visualization in which *software measures are mapped to attributes of the visualized entities*. Here each node represents a class. We use the height, width and colour of each node to map up to three measures. In the System Complexity View, the width of a node is proportional to the number of attributes (instance variables) it defines, the height represents the number of methods, and the colour represents the number of lines of code (darker ⇒ more lines).

A tall, skinny node has many methods but few attributes, and a wide node has many attributes. A very dark node contains a lot of code. Hence a tall, dark node contains a lot of logic but little state. A large, wide, light node would contain a lot of data but very little behaviour. A tall dark node at the bottom of an inheritance hierarchy is inheriting little but adding a lot, whereas a small node at the leaf of a hierarchy is inheriting most of its behaviour.
Detection strategy

> A detection strategy is a *metrics-based predicate* to identify *candidate* software artifacts that *conform to* (or violate) a particular *design rule*
This book shows how to exploit software metrics to detect software entities that violate design rules.


A sample chapter is available online:

http://scgresources.unibe.ch/Literature/ESE/Lanz06a-OOMP-sample.pdf
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 understandability
  — Increases system fragility

- Class uses directly more than a few attributes of other classes
  - ATFD > FEW

- Functional complexity of the class is very high
  - WMC ≥ VERY HIGH

- Class cohesion is low
  - TCC < ONE THIRD
A “God class” is a sign of *procedural thinking*, as it *concentrates responsibility in a single class*, rather than distributing it. This makes the system harder to understand, as multiple concerns are tangled within a single class. It also makes the software more fragile to change, as the God class is involved in all aspects of the system. Any change will impact the God class, and likely many other parts of the system as well.

https://en.wikipedia.org/wiki/God_object
This God class detection strategy uses software metrics to identify classes that (1) access many other parts of the system; (2) have high functional complexity; and (3) exhibit low cohesion (many unrelated responsibilities).

ATFD, WMC and TCC are *metrics*, and FEW, VERY HIGH and ONE THIRD are *thresholds* that can be adjusted as needed.

- **ATFD** = Access to Foreign Data (counts how many attributes of other classes are accessed)
- **WMC** = Weighted Method Count (measures “cyclometric complexity” of methods of a class)
- **TCC** = Tight Class Cohesion (counts method pairs that access common attributes of a class)
ModelFacade (ArgoUML)

- 453 methods
- 114 attributes
- over 3500 LOC
- all methods and all attributes are static
This visualization is a class blueprint. It shows five categories of methods and attributes of a single class and their calling/accessing relations. In the first column at the left are constructors, followed by public methods, internal (protected or private) methods, accessors, and finally attributes. This class has no accessors, but accesses its attributes directly.

This class of the ArgoUML system (a UML editing framework) is identified as a potential God class. Strictly speaking, this is a Facade rather than a God class, but it has become a “black hole” of functionality in the system.

https://en.wikipedia.org/wiki/Facade_pattern
> Methods with **Feature Envy** are more interested in data of other classes than their own.

- Method uses directly more than a few attributes of other classes: \( \text{ATFD} > \text{FEW} \)
- Method uses more attributes of other classes than its own: \( \text{LAA} < \text{ONE THIRD} \)
- The used "foreign" attributes belong to very few other classes: \( \text{FDP} \leq \text{FEW} \)
Fowler et al. gave this colorful name to a class that pulls data from other classes:

“We’ve lost count of the times we’ve seen a method that invokes half-a-dozen getting methods on another object to calculate some value. Fortunately the cure is obvious, the method clearly wants to be elsewhere, so you use Move Method to get it there. Sometimes only part of the method suffers from envy; in that case use Extract Method on the jealous bit and Move Method to give it a dream home.”

This detection strategy uses three heuristics to identify a method with Feature Envy: it uses more than a “few” attributes of other classes, it uses more foreign attributes than those of its own class, and not too many other classes are involved.

• ATFD = Access to Foreign Data (counts how many attributes of other classes are accessed)

• LAA = Locality of Attribute Accesses (number of attributes of the method’s class divided by the number of variables accessed)

• FDP = Foreign Data Providers (number of classes of attributes accessed)
ClassDiagramLayouter
The `weightAndPlaceClasses` and layout methods of `ClassDiagramLayouter` are very large, and use many accessors and attributes of `ClassDiagramNode`. The latter has little behavior of its own. Likely fragments of code in the client methods can be extracted and moved to the data class.
> A *Data Class* provides data to other classes but little or no functionality of its own.
A Data Class is the complement to a God Class. It has almost no responsibility and serves mainly as a repository of information to be used and accessed by methods of other classes. The goal of refactoring is to bring the data and responsibility together, either by moving methods to the data class, or to move the data to the classes where the behaviour is.

- WOC = Weight Of Class (number of “functional” public methods divided by the total public methods)
Data Class (2)

Class has many public data
NOPA + NOAM > MANY

Complexity of class is not very high
WMC < VERY HIGH

More than a few public data
NOPA + NOAM > FEW

Complexity of class is not high
WMC < HIGH

Class reveals many attributes and is not complex
Here we have a case of a composite detection strategy, to avoid having a single, complex diagram.

- NOPA = Number Of Public Attributes
- NOAM = Number Of Accessor Methods
- WMC = Weighted Method Count (measures “cyclometric complexity” of methods of a class)
Property
Property is a classical data class with almost no behavior of its own. Sometimes data classes can be merged with their client class, and sometimes client methods or parts of client methods can be moved to the responsibility of the data class.
> A change in an operation implies many (small) changes to a lot of different operations and classes.
“Shotgun Surgery” refers to a method with a large number of incoming calls. Changes to it will have a large impact on clients.

- **CM** = Changing Methods (number of distinct methods calling the measured method)
- **CC** = Changing Classes (number of classes in which the calling methods are defined)
Project
Project has both a large number of clients and is also a client of many classes. A change to this class risks breaking much of the system.
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?
> What is a “detection strategy”?
> How does a polymetric view visualize software metrics?
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?
> Would you expect to find large or small classes at the leaves of an inheritance hierarchy in a System Complexity View?
Attrition-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format
Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/