1DV610 – Introduction to Software Quality

Daniel Toll – daniel.toll@lnu.se

Dr. Rüdiger Lincke – rudiger.lincke@lnu.se

HT 2016

Course Structure

- Course Introduction/Lecture 1 – Introduction to Software Quality
- Lecture 2 – Software Requirements and Testing
- Lecture 3 – Metrics, Smells & Heuristics
- Lecture 4 – Daniel…
- Lecture 5 –
- Lecture 6
- Lecture 7
- Lecture 8
- Lecture 9

Metrics, Smells & Heuristics

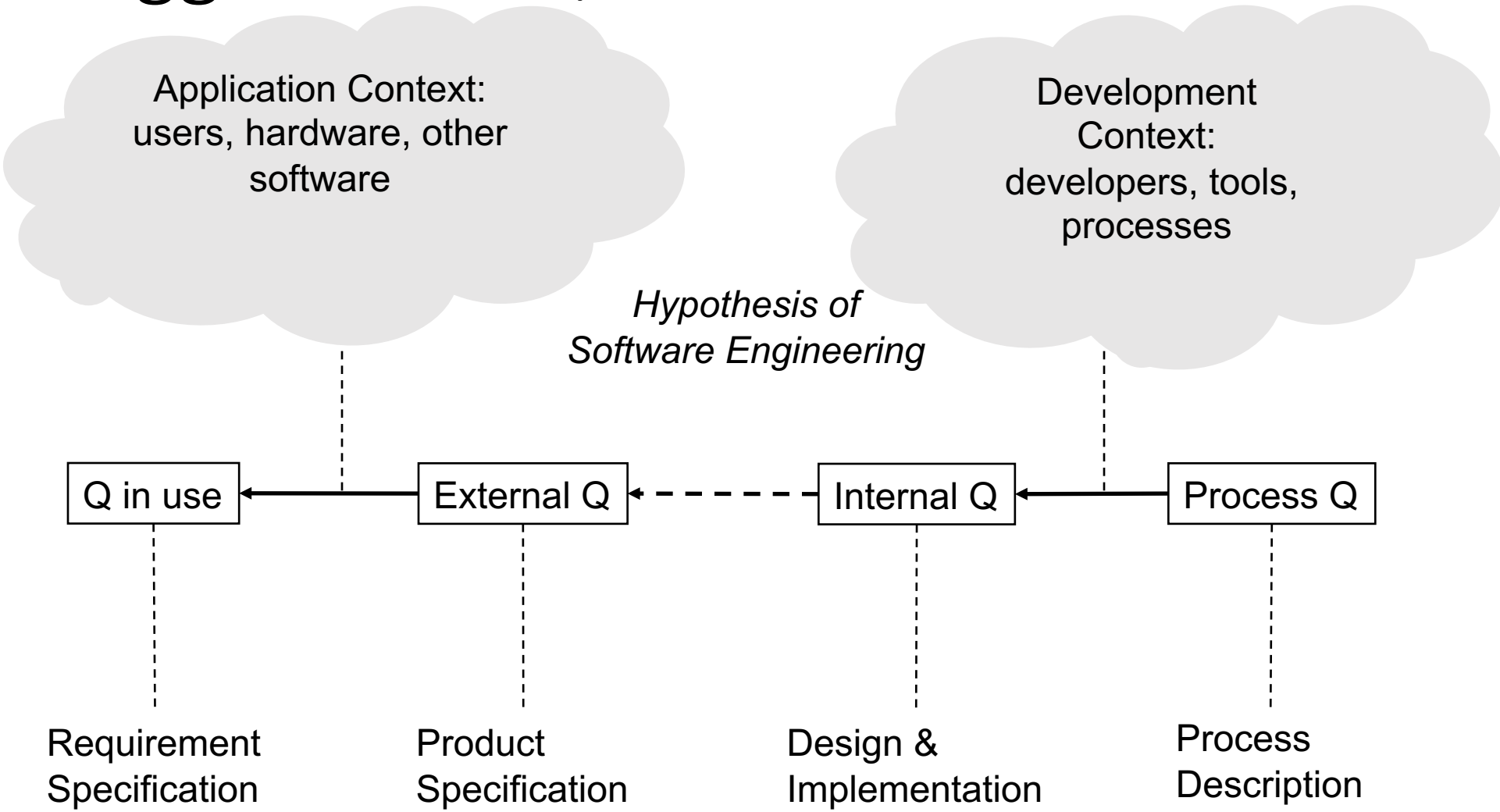
- Metrics – Measure properties in software
  - Definition
  - Role – Hypothesis of Software Engineering
  - Software Measurement
  - Software Quality Models
  - Some Metrics with Examples and Recommendations
    - Size – LOC, NOM
    - Complexity – CC, WMC
    - Coupling – CBO, Ca, Ce
    - Cohesion – LCOM, TCC
    - Architecture
    - Tracking evolution
- Smells & Heuristics – More fuzzy rules for good/bad practices

Software Quality Metrics – The Solution?

(Software Quality) Metrics Definition

- Well-defined mapping of software (process) entities to numerical values.
- A software metric is a function measuring a property in a software system or processes.
- The obtained measure expresses this property or quality as a numerical value.
- Often metric and measure is used synonymous.
- Quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development.
- The goal is obtaining objective, reproducible and quantifiable measurements.
- Useful, e.g., in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments.

The Bigger Picture/Context



Software Quality Levels

- Quality in use: How well does the software [in a Hardware environment] support/enable the user to accomplish a task.
- External Quality: How well does the software [in a Hardware environment] fit the product specifications (functionality, reliability, usability, efficiency)
- Internal Quality: How easy is the software to understand, maintain, etc. (properties not visible to user)
- Process Quality: How well do methods, tools, … support the construction of the software

Hypothesis of Software Engineering

- Process and internal quality affects external quality and quality in use.
- Our focus is the **direct** assessment, i.e., measurement, of internal quality to **indirectly** assess external quality and quality in use.
- I.e., if the process and code is good, usually the finished software is good as well, and vice versa.
- Measuring internal quality (size, complexity, etc.) is cheaper and allows earlier detection of problems than measuring external quality (bugs, failures, etc.).

Software Measurement

- Our working definition (repeated):

  A well-defined mapping of software (process) entities to numerical values.

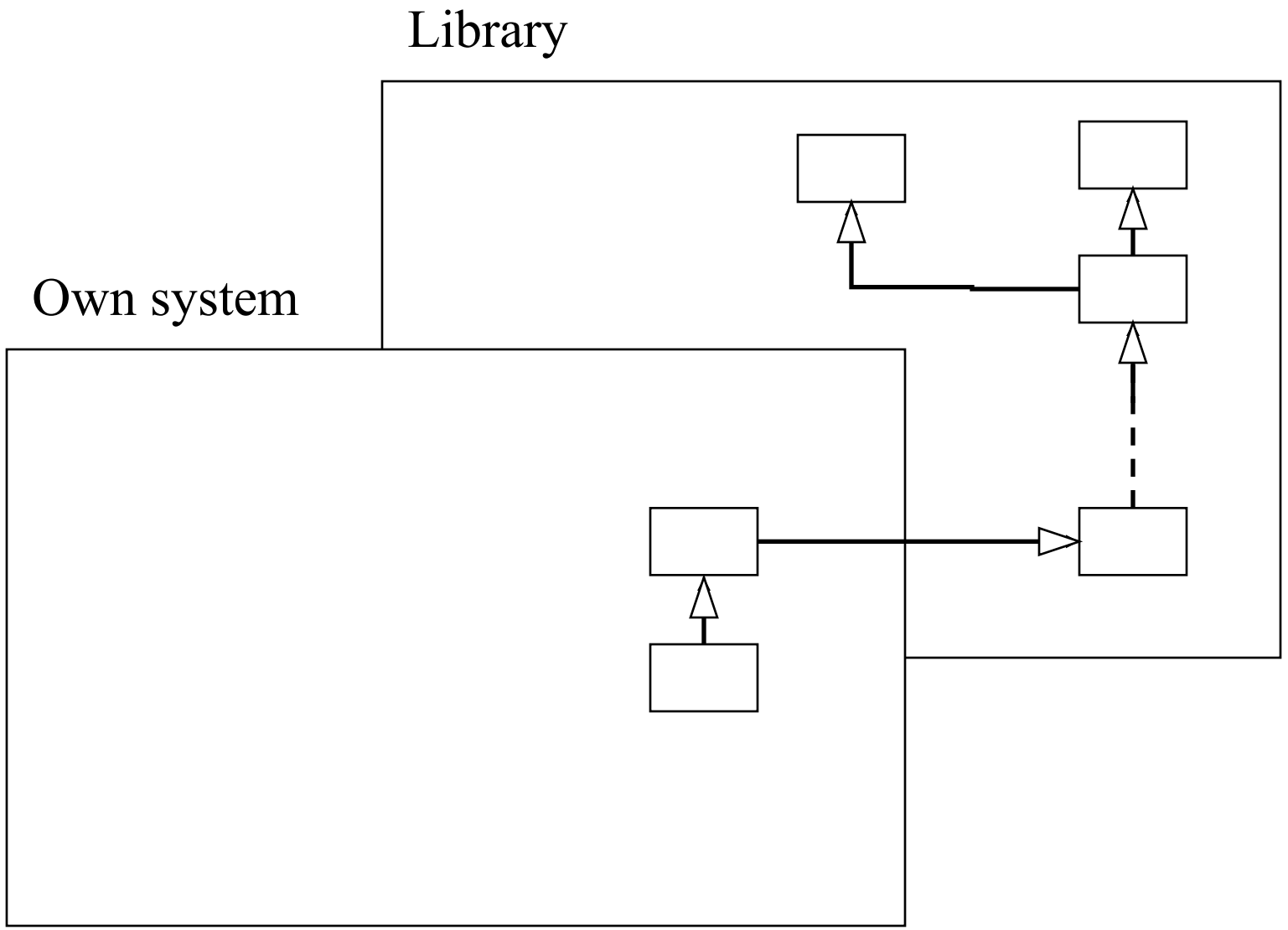
- Entity: the (kind of) objects that we assess.
- Scope: which subset all given objects do we assess.
- Attribute: the property of the object we assess.
- Measurement/Metric: the way we assess the attribute.
- Analysis: how to we further process the metric values and derive conclusions.

Example: Number of Methods (NOM)

- NOM = number of methods of a Class

- Private, public, friend methods?
- Setter/getter methods?
- Inherited methods?
- What is a method in non-object-oriented languages?

Discussion



Entities

- Software product:
  - E.g., source code, use case, test specification
  - structure and behavior of software
- Software process:
  - E.g., inspection, design, testing, maintenance
  - Indirectly measured using software product metrics over development time.
  - Direct process metrics, e.g., time between two check-ins, etc.
- Organization:
  - E.g., productivity, process maturity, etc.
  - Indirectly measured using process metrics
  - Direct, e.g., team size, structure, ...

Scope

- E.g., whole system, sub-systems, probes, ...
- Often you don’t want to assess all possible entities
  - Time constrains
  - Premature parts you know of
  - Exceptions you deliberately made
- You need agreement on your scope, otherwise (a real problem in practice):
  - Different scopes
  - Different metric values
  - Different analysis and conclusions

Attributes

- Internal attributes:
  - Connected to the subject itself
  - E.g., size of software
  - Objective
- External attributes:
  - Properties related to the subject environment
  - E.g., resource usage of software (Hardware environment), complexity (environment: person to perceive/judge it)
  - Often subjective

Measurement Properties

Measurement Properties cont.

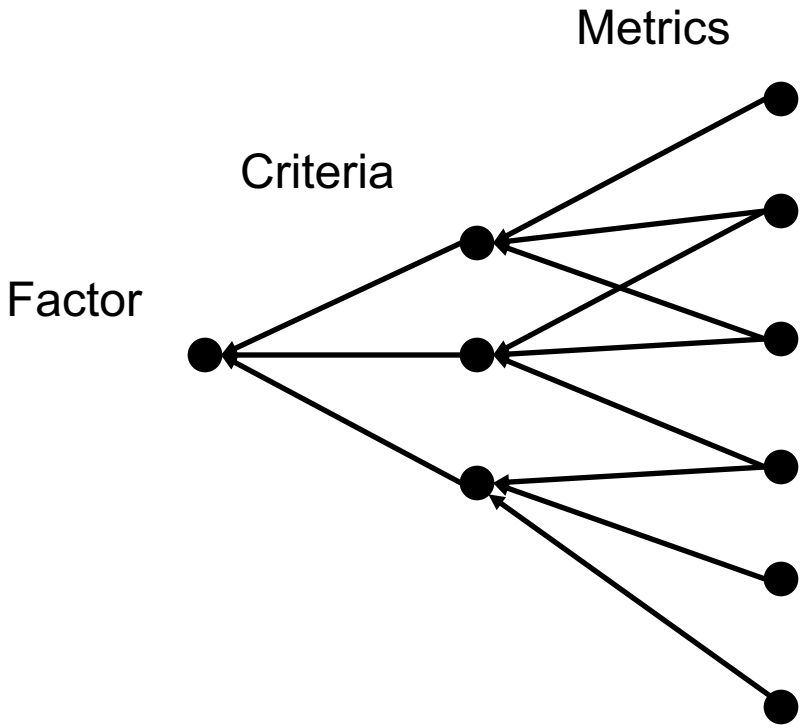
- Validity: measure reflects intuition and real world observations
- Reliability: stability and reproducibility (in contrast, e.g., in physics where one can destroy the object measured)
- Objectivity: actually, in measurement theory (we also allow subjective measurements, e.g., expert knowledge, code inspections, peer reviews)
- Robustness: small variations on the object's properties yield small variations in measurement values (allows dealing with incomplete objects)
- Sensitivity: measurement should be precise enough for the goal

Software Quality Models

- The concept of quality operationalized.
- Definition and measurement/decision procedure.
- Questions to discuss:
  - Subject of Q (specific product, domain, type)
  - Q-goal(s)
  - Properties (supporting the goal) and attributes (direct measurable, supporting properties)
  - Measurement/decision procedure for attributes/properties

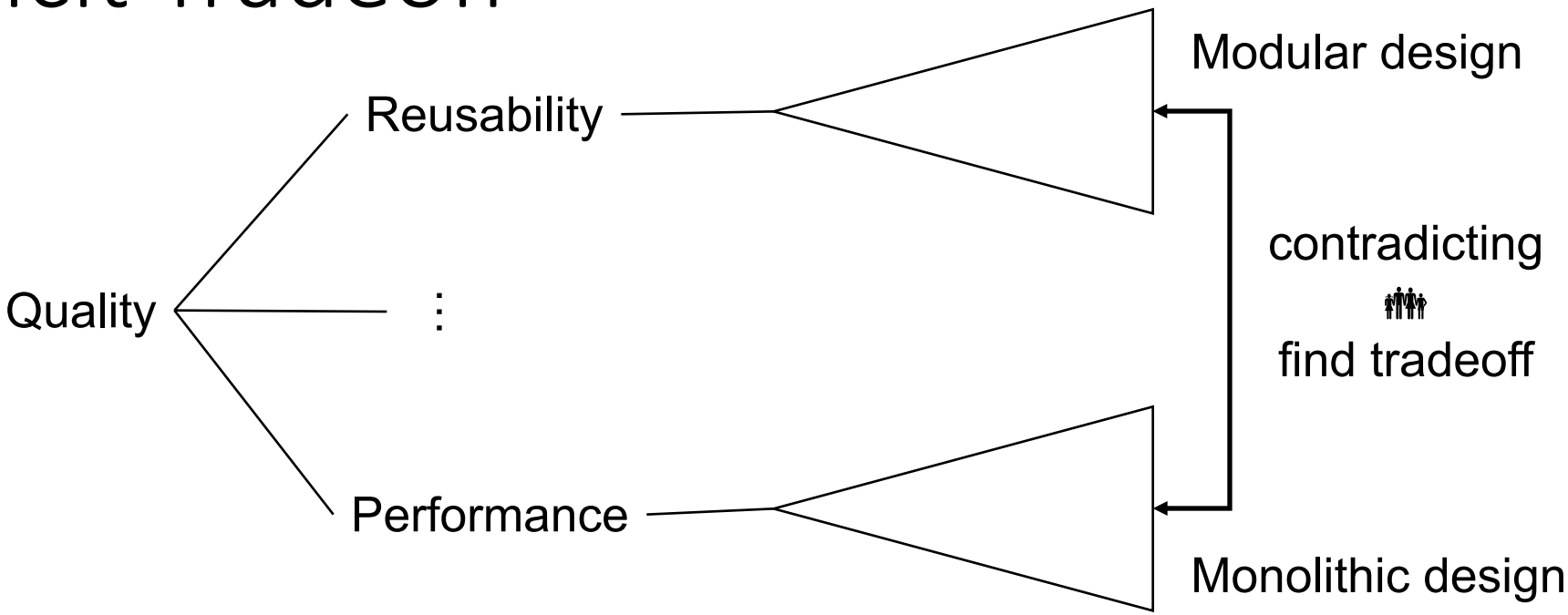
Factor-Criteria-Metrics (FCM) Model

[McCall, Richards 77]



Basis of
ISO 9126

Explicit Tradeoff



- Contradictions within one QM possible,
- Almost guaranteed when integrating different QM (e.g. reusability, performance, safety)

Common Metrics

Brief overview (examples on following slides):

- Test/code coverage
- Size and Complexity metrics (LOC, CC, WMC, etc.)
- Cohesion/coupling (TCC, LCOM, CBO, etc.)
- Function Points (FP)
- Halstead Complexity (not good)
- All kinds of scores
- Features/bugs/epics open/closed
- Time spent/est. left
- Etc.

Test Coverage Metrics

- In computer science, code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite.
- A program with high code coverage, measured as a percentage, has had more of its source code executed during testing which suggests it has a lower chance of containing undetected software bugs compared to a program with low code coverage.
- Many different metrics can be used to calculate code coverage; some of the most basic are the percent of program subroutines and the percent of program statements called during execution of the test suite.
- Good values should be around 80-90%. 100% is usually not needed/achievable.
- Assuming you write good and thoughtful tests (not cheating just to meet the numbers).
- Goal is to see how well requirements, tests and code are aligned and to achieve a good cost value ratio.

Coverage Criteria

- There are a number of coverage criteria, the main ones being:
- Function coverage - Has each function (or subroutine) in the program been called?
- Statement coverage - Has each statement in the program been executed?
- Branch coverage - Has each branch (also called DD-path) of each control structure (such as in if and case statements) been executed? For example, given an if statement, have both the true and false branches been executed? Another way of saying this is, has every edge in the program been executed?
- Condition coverage (or predicate coverage) - Has each Boolean sub-expression evaluated both to true and false?

Coverage Criteria Example

For example, consider the following C function:

```
int foo (int x, int y)
{
    int z = 0;
    if ((x>0) && (y>0))
    {
        z = x;
    }
    return z;
}
```

- Assume this function is a part of some bigger program and this program was run with some test suite.
- If during this execution function 'foo' was called at least once, then *function coverage* for this function is satisfied.
- *Statement coverage* for this function will be satisfied if it was called e.g. as foo(1,1), as in this case, every line in the function is executed including z = x;.
- Tests calling foo(1,1) and foo(0,1) will satisfy *branch coverage* because, in the first case, the 2 if conditions are met and z = x; is executed, while in the second case, the first condition (x>0) is not satisfied, which prevents executing z = x;.
- *Condition coverage* can be satisfied with tests that call foo(1,1), foo(1,0) and foo(0,0). These are necessary because in the first two cases, (x>0) evaluates to true, while in the third, it evaluates false. At the same time, the first case makes (y>0) true, while the second and third make it false.

Size Metrics: Lines of Code (LOC)

- Lines of Code (LOC)
- Lines of code simply counts the lines of source code (line break characters) of a certain software entity.
- Can be used on:
  - Methods
  - Classes
  - Files
  - (Sub-)Systems
- Too large methods/classes/files should be avoided and split up.
- They are harder to understand, test, debug, etc.
- Size of a system could give indications on effort spent, effort needed.
- Indicates how much testing could should be expected.
- Growth trends can be deduced.

Size Metrics: Number of Methods (NOM)

- NOM = number of methods of a Class

- Private, public, friend methods?
- Setter/ getter methods?
- Inherited methods?
- What is a method in non-object-oriented languages?

Size Metrics: Recommendations

- Too large methods, classes, files should be avoided.
- They are difficult to understand, read, test, debug.
- Splitting them is advisable.
- Check if the five basic principles are applied.

Complexity Metrics: Cyclomatic Complexity

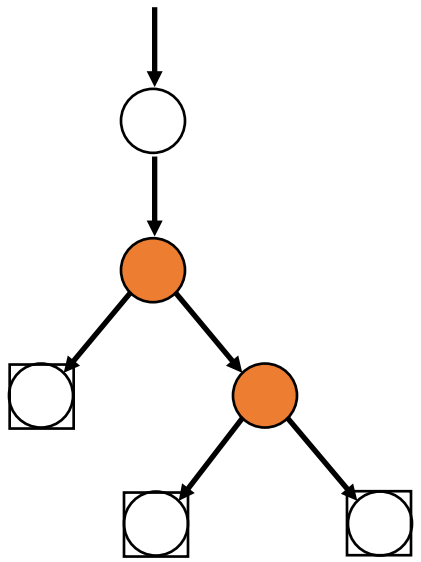
- Cyclomatic Complexity (CC) counts number of independent control path in a method.
- Can be used on:
  - Methods
  - Classes (as aggregation of methods)
- Too complex methods/classes/files should be avoided and split up.
- They are harder to understand, test, debug, etc.
- Complexity of a system/part could give indications on effort spent, effort needed.
- Indicates how much test could should be expected.
- Part of WMC (Weighted Method Count) aggregating CC to class level.

MCC – Direct Approx. Definition

- Let B_m=(V, E) a basic block graph of a method m with
- MCC(m) ≈ |E| - |V| + 2
- Preferred implementation, if B_m is computed anyway

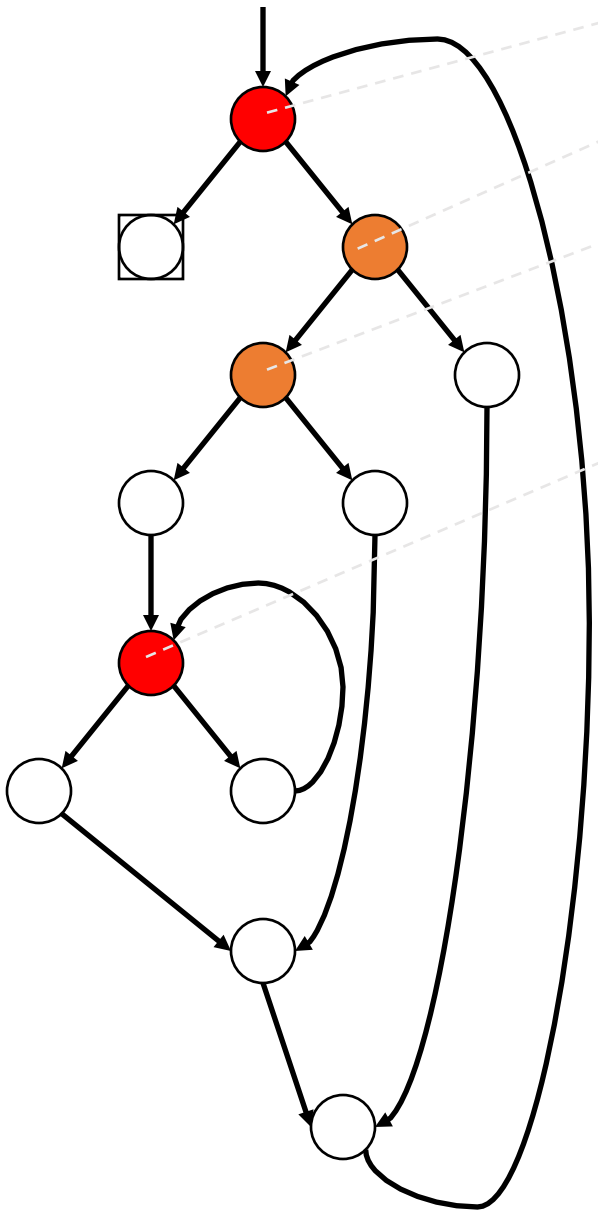
id (int *a*) {
    return *a*
}

- $|E| = 1$
- $|V| = 2$
- MCC(*id*) = 1 - 2 + 2 = 1



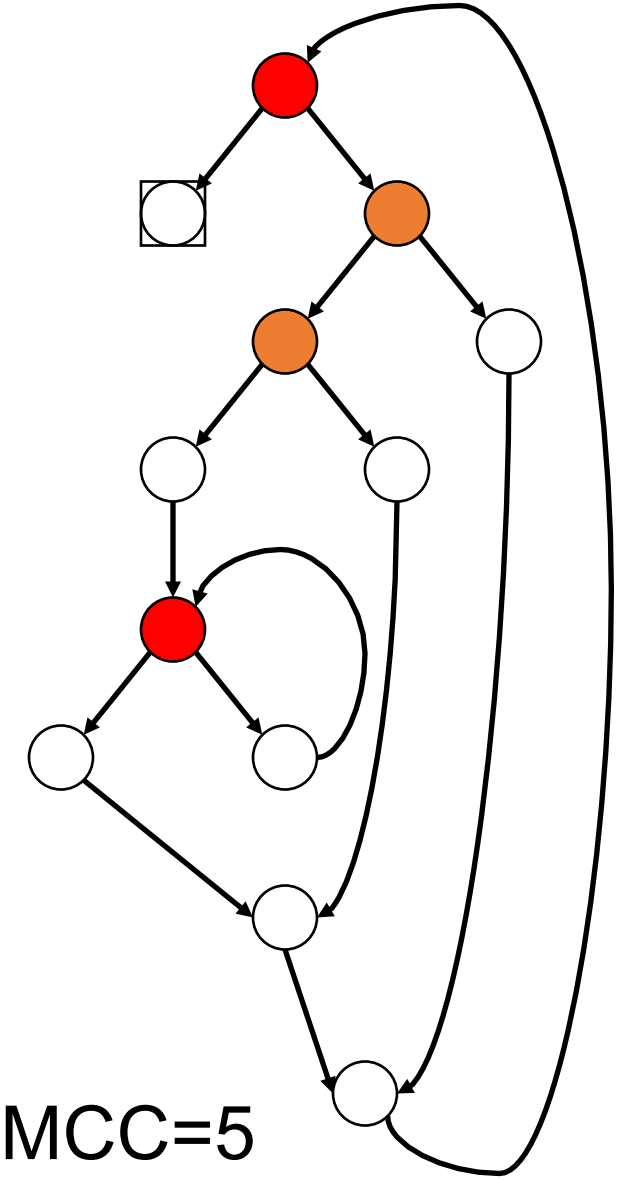
```
gcd (int a, b) {
  if (a == b)
        return a
  if (a < b)
        return gcd (b, a)
  else
        return gcd (b, a - b)
}
```

- $|E| = 5$
- $|V| = 6$
- MCC(*gcd*) = 5 - 6 + 2 = 1

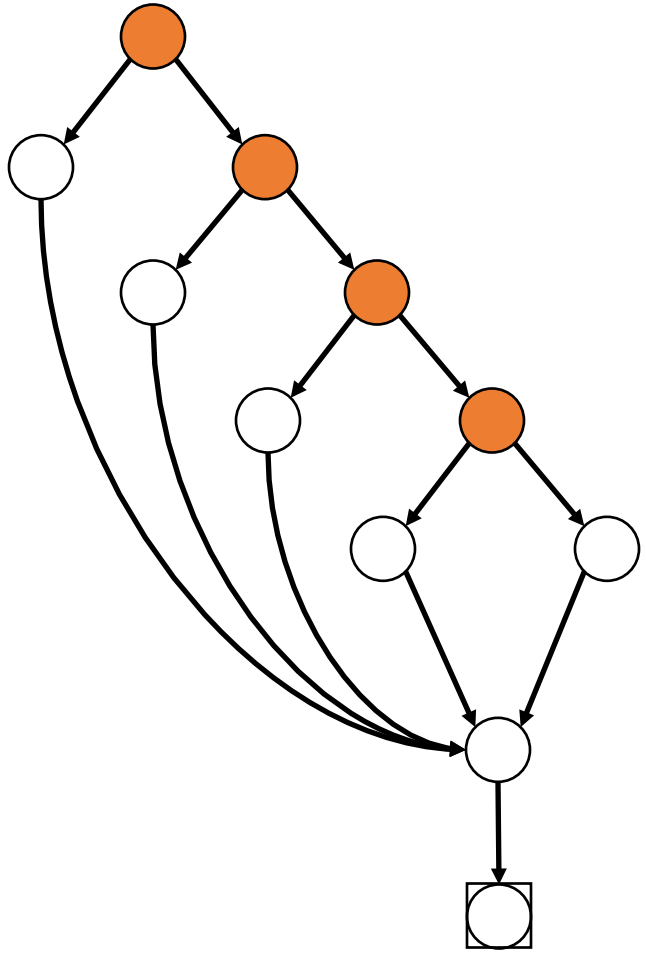


```
while (x<0) {
  if (y<0) then {
       if (x<-100) then {
              x:=y;
              while (y<-100){
                     y++;}
              y:=x;}
       else{
              x:=y;}
       }
       x:=y;
  else{
       x:=y;}
  x:=y;
}
return x;
```

- $|E| = 15$
- $|V| = 12$
- MCC(m) = 15 - 12 + 2 = 5



MCC=5



MCC=14-11+2=5

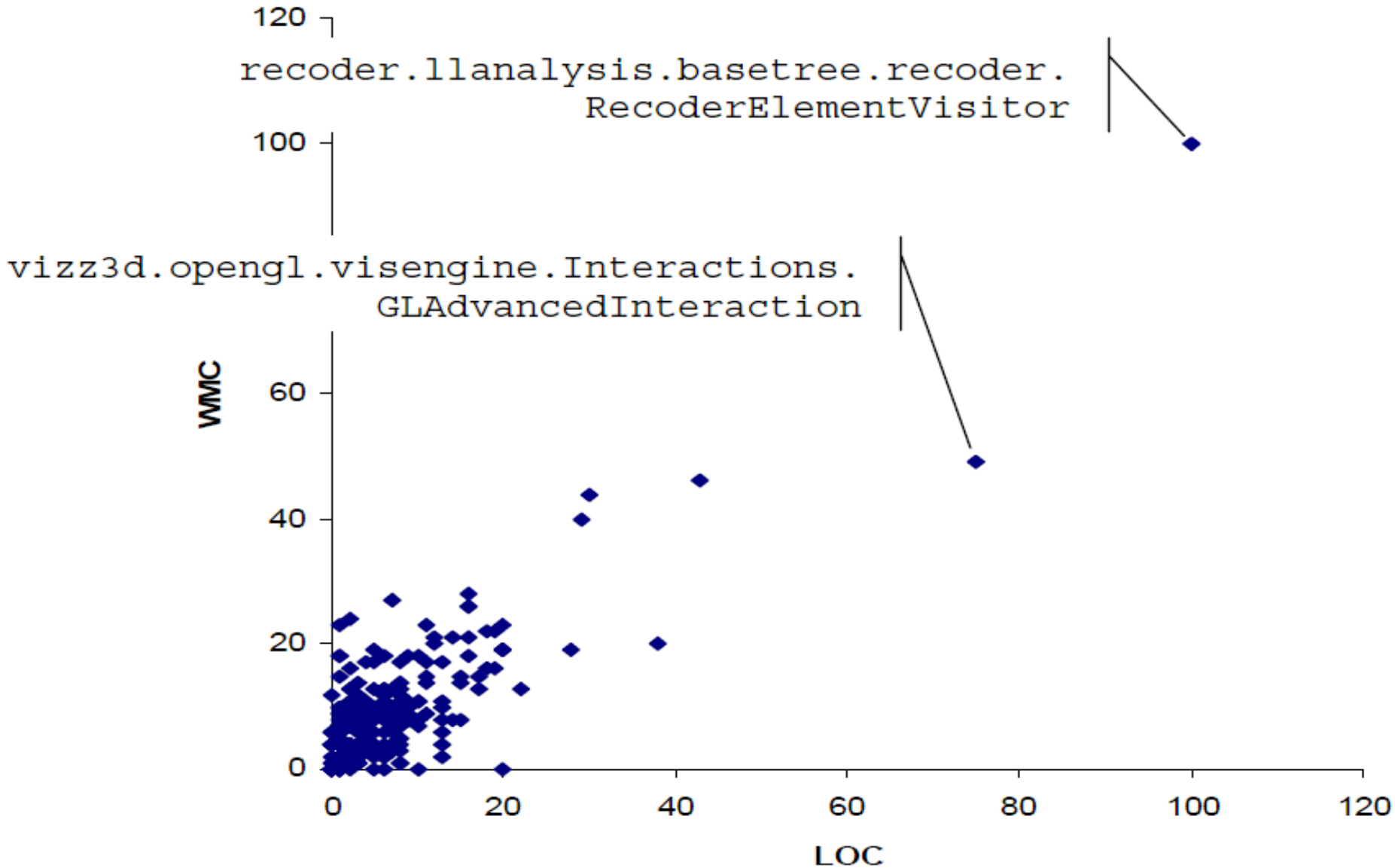
MCC – Discussion

- MCC is not intuitive (cf. examples before)
- OO complexity (with hidden control flow due to polymorphism) not captured
- By splitting methods and introduction of polymorphism MCC can be reduced (at the cost of increased programming effort)
- But, validation confirmed: high agreement of MCC and maintenance effort for methods. How come?

Complexity Metrics: Recommendations

- Too complex methods and classes should be avoided.
- They are difficult to understand, read, test, debug, change.
- Splitting them is advisable.
- Check if the five basic principles are applied.

Size vs. Complexity

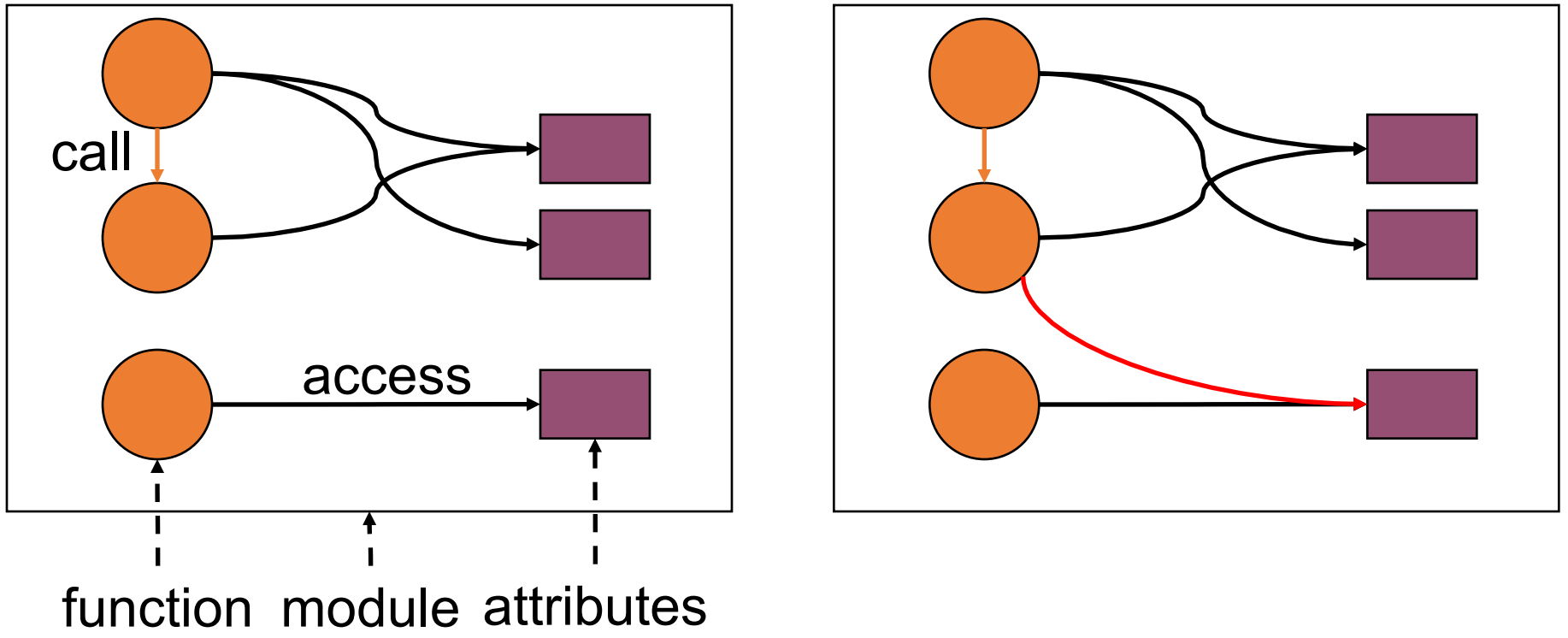


Coupling / Cohesion Metrics

- Both architecture and design metric.
- Coupling:
  - Connectivity between software entities.
  - E.g. a class is coupled to another if it refers to its attribute / invokes its methods, computation based on class usage graph.
- Cohesion:
  - Connectivity between software entities contained in the same container software entity.
  - E.g. connectivity of classes in a package or methods and attributes in a class.

Example Coupling / Cohesion

Left module is less cohesive than the right one since it has a smaller degree of coupling



Coupling Metrics: Coupling Between Objects

- Coupling between objects (CBO) counts number of other classes a class is coupled (type references, field access, method calls) to.
- Can be used on:
  - Classes
- Too coupled classes should be avoided and split up.
- They are harder to understand, test, debug, etc.
- Changes to coupled classes might have side effects in other classes.

Coupling Metrics: Recommendations

- Avoid high coupling between classes.
- In particular over package/component borders.
- They are difficult to understand, read, test, debug, change.
- Splitting them is advisable.
- Check if the five basic principles are applied.

Cohesion Metrics: Lack of Cohesion of Methods (LCOM)

- Lack of Cohesion of Methods (LCOM) counts number of independent “clusters” of methods in a class.
- That is unrelated methods and thus separate concerns.
- Can be used on:
  - Classes
- Too low cohesion in classes should be avoided and split up.
- They are harder to understand, test, debug, etc.

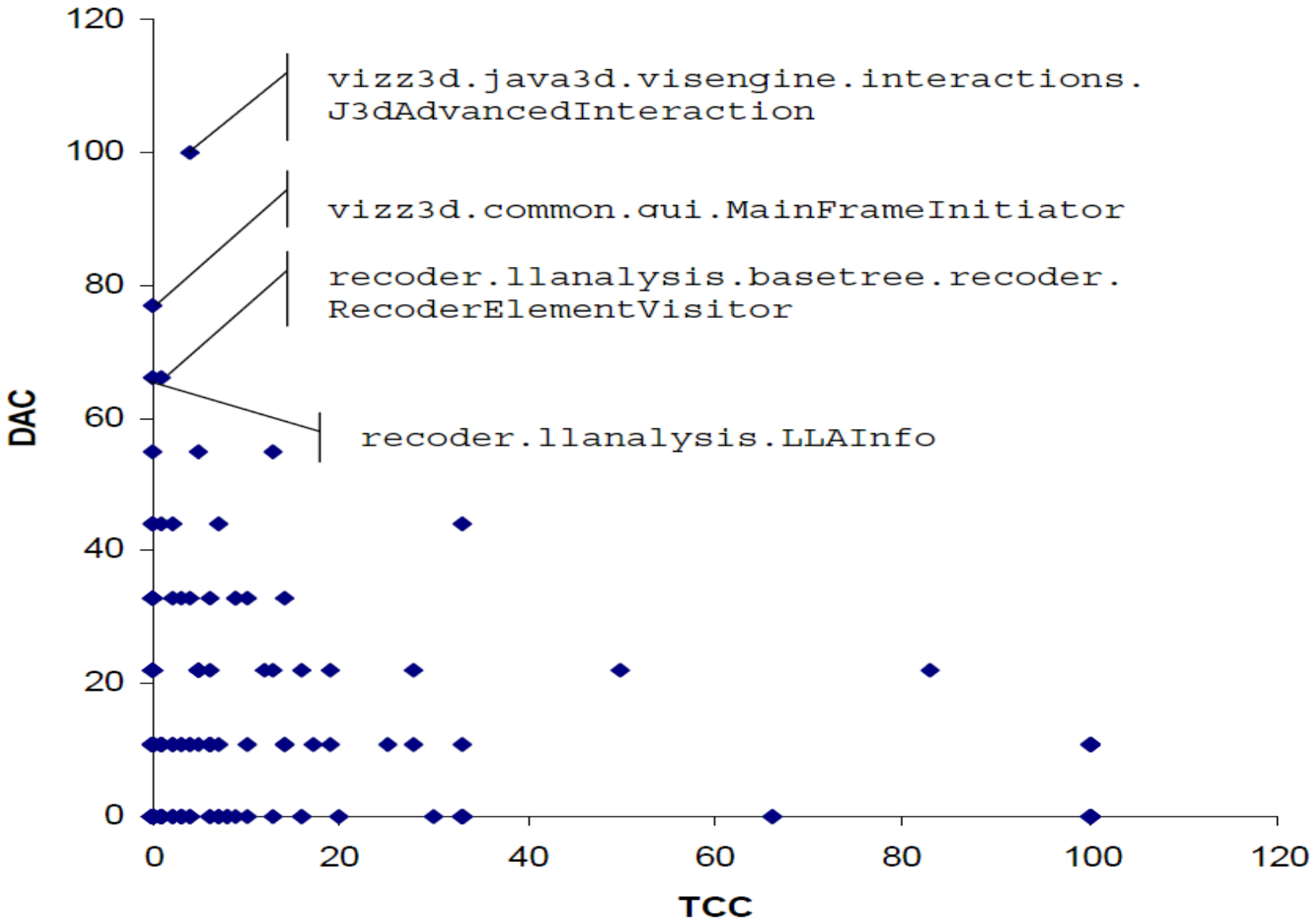
Cohesion Metrics: Recommendations

- Avoid low cohesion within a class (between methods and fields of a class).
- Distinguish classes which are simple data structures (low cohesion ok), and classes implementing business logic (low cohesion not ok).
- They are difficult to understand, read, test, debug, change.
- Splitting them is advisable.
- Check if the five basic principles are applied.

General Recommendations

- Use automated metric tools frequently measuring the metrics.
- Observe in particular classes being outliers with respect to more than one area, e.g., classes being large, complex, having high coupling and low cohesion.
- Strong indication that they do not meet the basic principles.
- They usually tent to be difficult to understand, read, test, debug, change.

Cohesion vs. Coupling



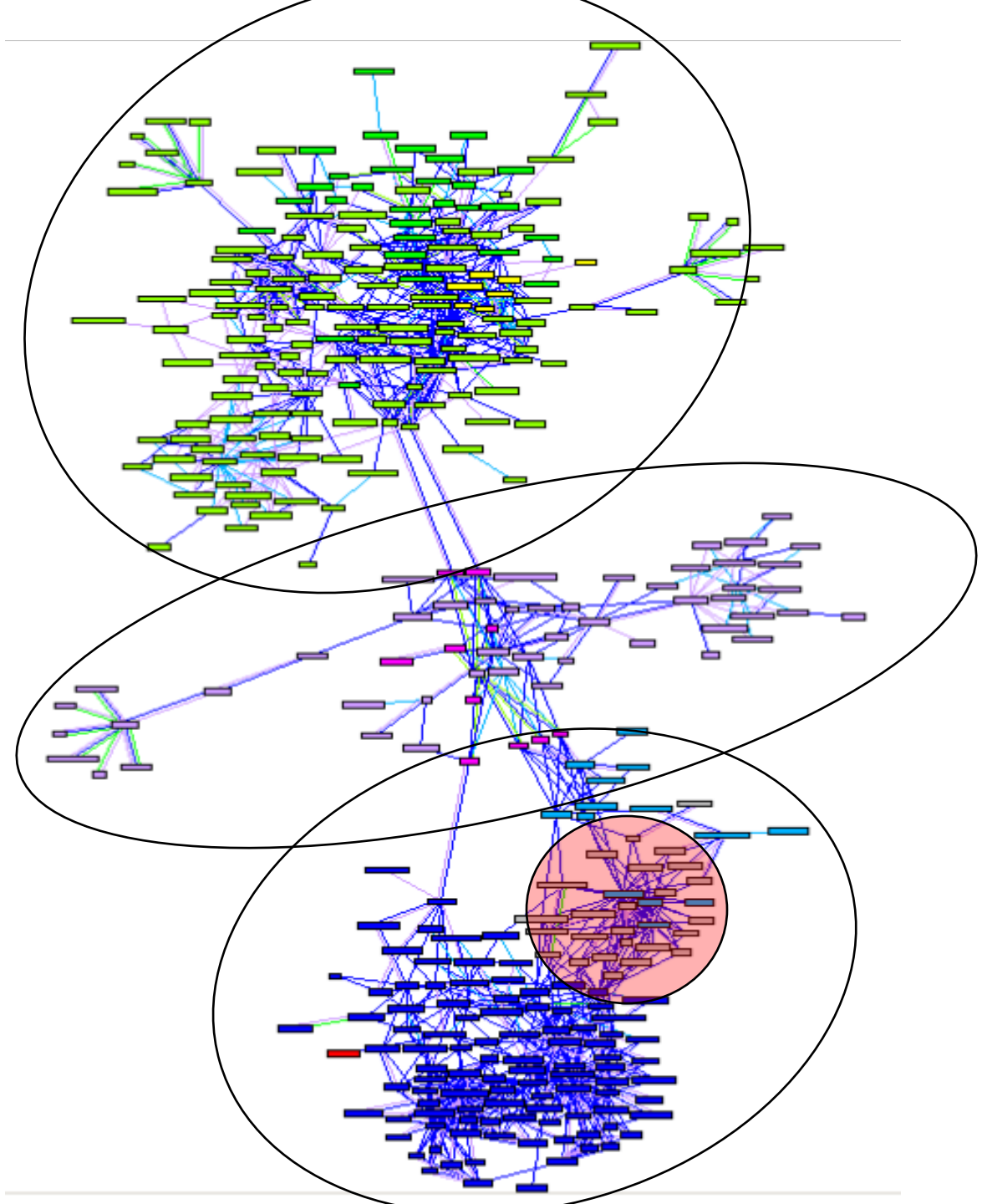
Architecture

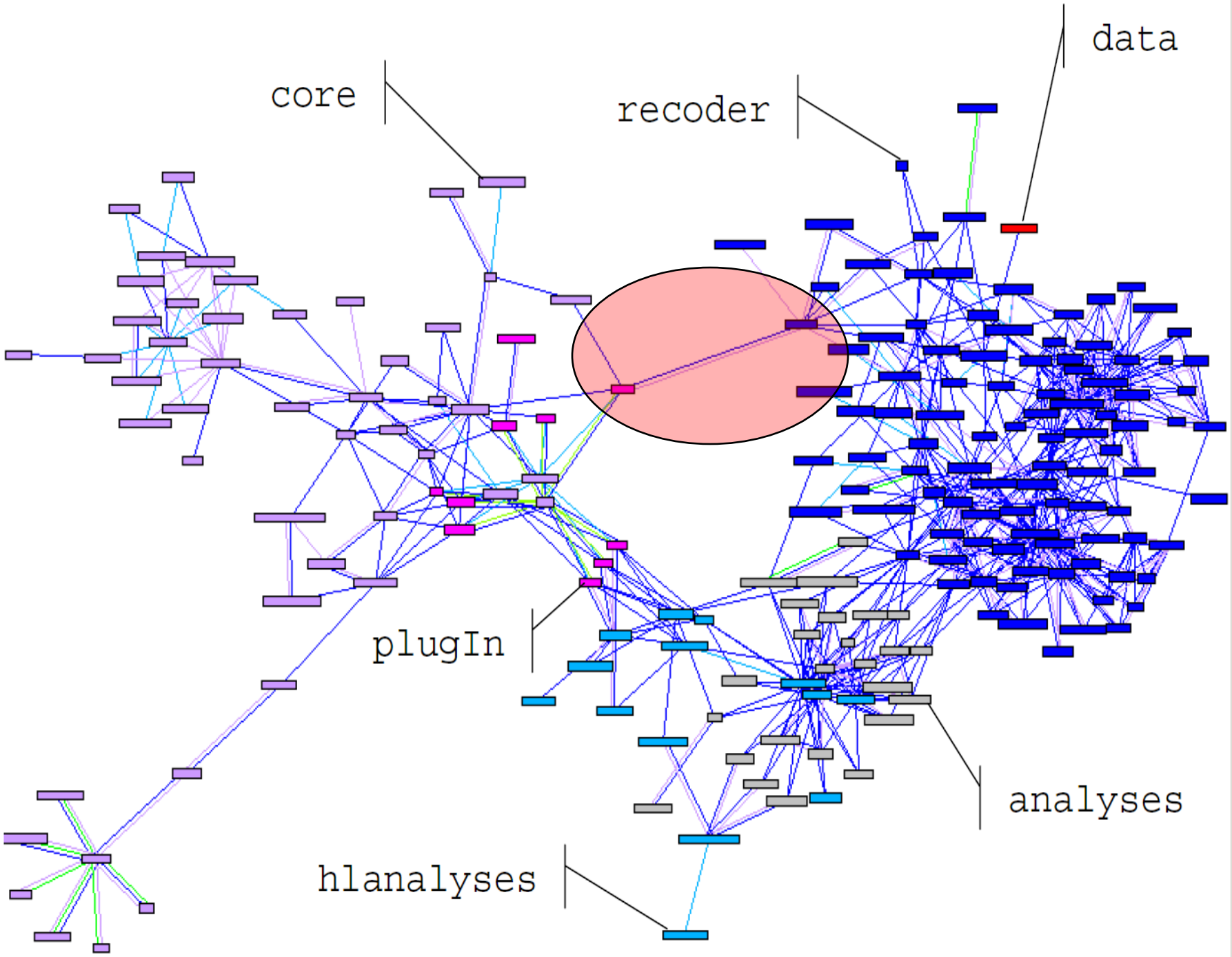
- Systems are designed with a target architecture
  - E.g., three-tier-architecture, client-server, …
- Architecture might deteriorate during development and maintenance (time pressure, misunderstandings, complexity)
- Measurement tools may
  - Visualize the implementation architecture
  - Check for conformance of existing to intended architecture
- Metrics to analyze architecture: assess class coupling and package cohesion, e.g.,
  - TPC – Tight Package Cohesion (package cohesion metric)
  - DAC – Data Abstraction Coupling (class coupling metric)
  - RFC – Response Set of a Class (coupling metric)

Usage Examples

Architecture Recovery

- Component
  - High cohesion of contained classes.
  - Low coupling to other components.
- Problem
  - Cohesion is defined as connectivity between software entities contained in the same containing software entity i.e. of classes within the same component.
  - Defining components is the problem of architecture recovery.
- Solution
  - Measure dependencies of classes.
  - Find a partitioning that maximizes cohesion and minimizes coupling.
  - Checking each partitioning is exponential but good heuristics exist.

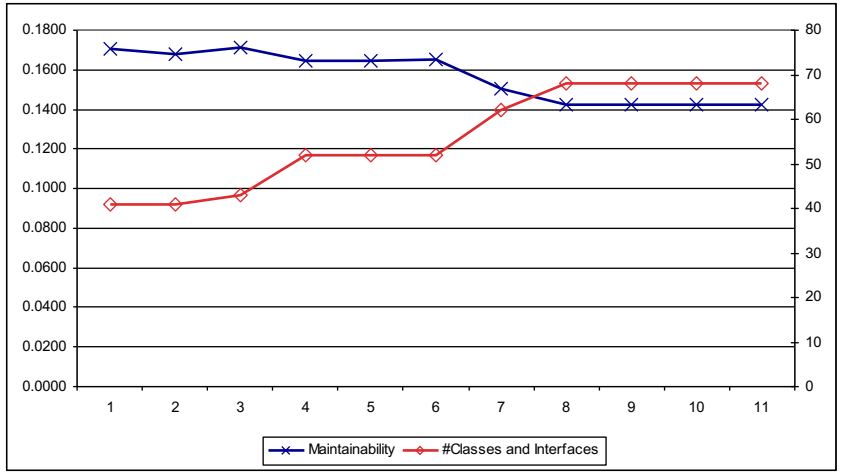



data
core
recoder
plugIn
analyses
hlanalyses

Track System Evolution

- Establish quality goals
- Refine them to a set of metrics using a quality model
- Monitor the metrics regularly over development/maintenance period
- Does quality improve?
- Example:
  - Quality: maintainability (high is good)
  - Metrics: fewer change sensitive & complex classes
  - Monitor: change sensitivity (next slide) and complexity

Example: Track System Evolution

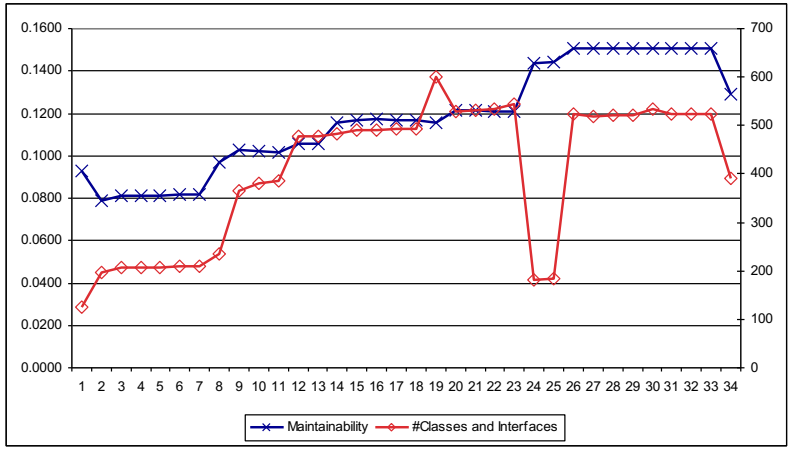


Maintainability vs. size,

over versions of a system

a sane system (left)

a critical one (right)



Change Sensitive Parts

- Systems change over time
- Some parts will often change since they depend on a lot of classes
- Inversely, changes to some parts cause more work since they are complex or have many depending classes
- Dependencies can be measured
- Helps to estimate change effort specific for different parts of the system

Metrics Conclusions

- Metrics need to be well-defined
  - Meta-model
  - Language mapping
  - Metrics themselves
- OO metrics suites are not well-defined per se but useful nonetheless, e.g., for assessing maintainability
- When using them in a continuous process,
  - first define the metrics exactly
  - than apply them and analyze and store the results

Smells & Heuristics

- Loose list of (fuzzy) recommendations.
- If something smells bad, you should get rid of it (before it spreads).
- This is usually done by refactoring.
- Refactoring means improving code structure without changing the behavior.
- Automated tests are prerequisite for:
  - doing frequent refactoring's improving code quality without
  - changing implemented requirements.
- Clean Code book, Chapter 17 includes an extensive (but incomplete) selection of 66 smells & heuristics.
- Please read this by yourself.

Software Quality Metrics – The Solution?

Summary

- Metrics measure properties in code.
- Automated metrics help to find problems in large amounts of code.
- Main areas are size, complexity, cohesion and coupling.
- Metrics can be used for:
  - following trends,
  - recovering architecture,
  - planning refactoring,
  - Etc.
- Applying insights gained from metrics are difficult without good test coverage (maybe most important metric).
- Solution to some problems, but not all!
- Need to be used in connection with others, most of all good testing.
- Without good testing (test coverage) it is not possible to apply any refactoring's suggested by use of metrics.