I402A Software Architecture
and Quality Assessment

Session 4

Metric and Code Evaluation

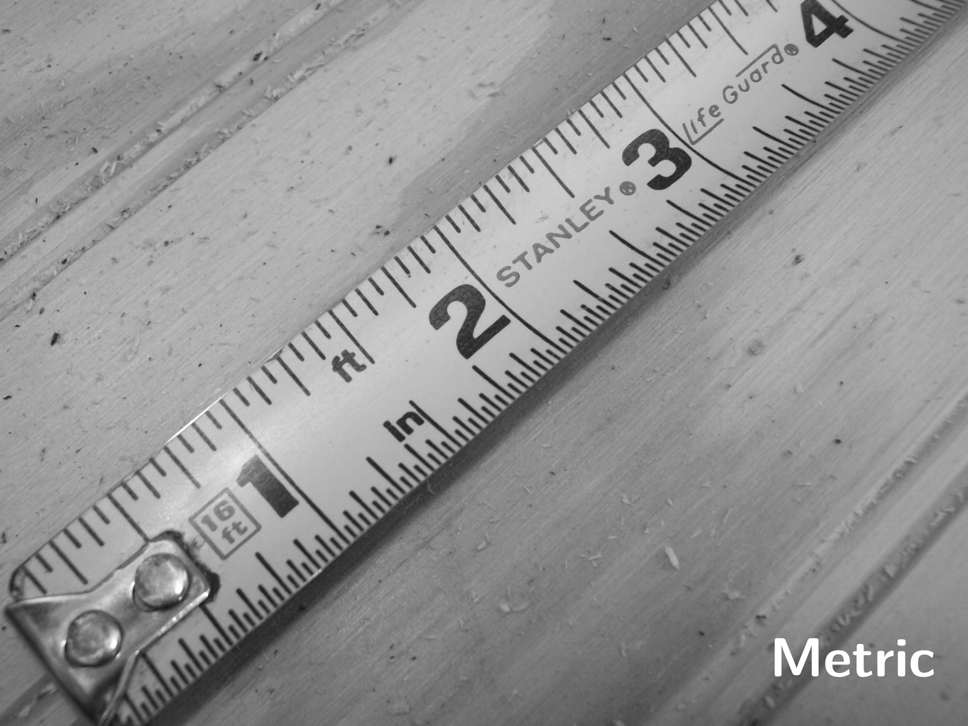
Sébastien Combéfis

Fall 2019

Objectives

- Notion of metric and evaluation of properties
  - Halstead software complexity
  - McCabe cyclomatic complexity
  - Henry and Kafura fan-in fan-out complexity
- Standard metrics to evaluate code
  - Presentation of several metrics and properties to evaluate
  - Particular case of object oriented systems

Metric



Metric (1)

- Measure a criterion to better understand it

  As a value to be able to evaluate, to compare, etc.

*"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 beginnings of knowledge but you have scarcely in your thoughts advanced to the stage of Science." — Lord Kelvin (physicien)*

Metric (2)

- Measuring to be able to control

  Evaluate and improve the quality according to the measure

"You cannot ***control*** *what you cannot measure." — Tom DeMarco (software engineer)*

- Not easy to determine what you want to measure

  Importance of the choice of measures and criterion to evaluate

"In truth, a good case could be made that if your knowledge is meagre and unsatisfactory, the last thing in the world you should do is make measurements; the chance is negligible that you will measure the right things accidentally." — George Miller (psychologist)

Measure

- Assign a number to an attribute of a real-world entity

  Description of entities using unambiguous rules

- Ability to measure products or processes

  A class, a module or documentation, tests, etc.

Entity	**Attribute examples**
Design	Number of defects detected by a review
Specification	Number of pages
Code	Number of lines of code, number of operations
Development team	Team size, average team experience

Metric Type (1)

- Direct measure of a property/a criterion

  Number of lines of code, number of classes, etc/

- Indirect measure or measure derived from other measures

  Defects density = number of defects / product size

- Prediction based on measures

  Effort required to develop a software

Prediction

- Using a variable prediction model

  Relationship between predicted and measurable variables

- Three hypotheses for a variable to be predictable

  1. Software properties can be measured accurately
  2. Link between what we want to and what we can measure
  3. Relation understood, validated, expressible as model/formula

- Only few metrics are predictable in practice

  Difficulty to establish a precise model

Metric Type (2)

- Several types of values for a metric
  - **Nominal** is a label without order

    Programming language: 3GL, 4GL
  - **Ordinal** with order but no quantitative comparison

    Programmer skills: low, medium, high
  - **Interval** between values

    Programmer skills: between 55th and 75th percentiles population
  - Proportionality **ratio** to compare

    Software: twice as big as the previous
  - **Absolute** with just a value

    Software: 350000 lines of code

Measured Entity

- Measurement of a concrete product, typically a software

  Criteria of size, complexity or product quality

- Other measurable entities related to development

  Criteria on a process, a resource or a project

Metric and Business Goal

- No software-quality metrics matter intrinsically

  Even though they can be interesting

- Measures should be designed to answer business questions

  Software development should focus on subjective metrics

- Everything is a snowflake, unique, valuable and incomparable
  - Component, person, team, project or product
  - You can always measure, but not always possible to compare

Success Metric

- Use metrics that can be used to improve business value

  Continuously making incremental improvements to processes

- Nine metrics that can make a real difference
  - **Agile process**: lead and cycle time, team velocity, open/close rate
  - **Production analytics**: MTBF, MTTR, application crash rate
  - **Security**: endpoint incidents, MTTR

- Ultimate metric is success

  Automate “standard” metrics to focus on achieving success

Complexity

Halstead Software Complexity (1)

- Measurement software complexity by Halstead in 1977

  On the basis of the actual implementation of a program

- A program is a sequence of operators and operands
  - η_1, η_2 number of unique operators/operands
  - N_1, N_2 total number of operators/operands

"A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operands" — Maurice Halstead

Halstead Software Complexity (2)

- Several properties computable on a software

  Based on the measured values η_1, η_2, N_1 and N_2

- Program information volume measured in bits

  Size of any implementation of the algorithm

- Several measures of difficulty and effort
  - Difficulty or propensity to make mistakes
  - Effort to implement or understand an algorithm

Halstead Software Complexity (3)

Propriété	**Formule**
Vocabulary	$\eta = \eta_1 + \eta_2$
Length	$N = N_1 + N_2$
Volume (bits)	$V = N \times \log_2 \eta$
Difficulty	$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
Effort (elementary mental discimination)	$E = D \times V = \frac{\eta_1 N_2 \log_2 \eta}{2\eta_2}$
Implementation time (seconds)	$T = \frac{E}{S} = \frac{\eta_1 N_2 N \log_2 \eta}{2\eta_2 S}$
Number of bugs	$B = \frac{E^{2/3}}{3000}$ or $B = \frac{V}{3000}$

- $20 \leq V(\mathit{fonction}) \leq 1000$ et $100 \leq V(\mathit{fichier}) \leq 8000$
- Difficulty due to new operator and repeated operands
- S is the Stoud number worth 18 for a computer scientist

Halstead Software Complexity (4)

- **Advantages**
  - No need for advanced analysis of program control flow
  - Predictions on effort, error rate and implementation time
  - Gives overall quality measures
- **Disadvantages**
  - Depends on the use of operators and operands in the code
  - No prediction at the design level of a program

Halstead Software Complexity Example

```c
main()
{
    int a, b, c, avg;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a + b + c) / 3;
    printf("avg = %d", avg);
}
```

- **Unique operators** $(\eta_1 = 10)$: `main () {} int scanf & = + / printf`
- **Unique operands** $(\eta_2 = 7)$: `a b c avg "%d %d %d" 3 "avg = %d"`
- **Vocabulary and length** : $\eta = 10 + 7 = 17$ et $N = 16 + 15 = 31$
- **Volume, difficulty, effort** : $V = 126.7$ bits, $D = 10.7$, $E = 1355.7$
- **Implementation time** : $T = 75.4$ seconds
- **Number of bugs** : $B = 0.04$

McCabe Cyclomatic Complexity (1)

- Measuring the number of decision statements

  Many possible choices involve greater complexity

- Model based on a graph representing the decisions

  If-else, do-while, repeat-until, switch-case, goto, etc. statements

- Cyclomatic complexity computed on the flow graph

$$V(G) = e - n + 2$$

with e number of edges and n number of vertices

McCabe Cyclomatic Complexity (2)

- Several possible variants depending on what is measured
  - **Cyclomatic complexity** ($V(G)$)

    Number of independent linear paths
  - **Real cyclomatic complexity** (ac)

    Number of independent paths traversed by tests
  - **Complexity of the module design** ($IV(G)$)

    Pattern of calls from one module to others
- Ideally, the two first variants should match

  $V(G) = ac$

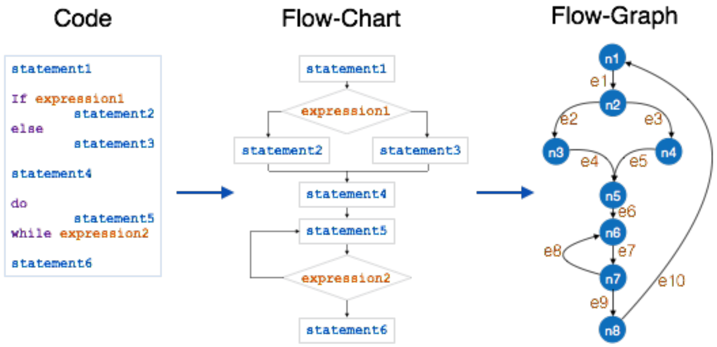
McCabe Cyclomatic Complexity (3)

- **Advantages**
  - Metric to evaluate ease of maintenance
  - Identifies the best zones where testing will be important
  - Easy to compute and implement
- **Disadvantages**
  - Does not evaluate the complexity of data, only of control
  - Same weight for loops, should they be nested or not

McCabe Cyclomatic Complexity Example

- Identify blocks delimited by decision statements

  Graph construction with nodes and edges



```
statement1

If expression1
        statement2
else
        statement3

statement4

do
        statement5
while expression2

statement6
```

$$V(G) = e - n + 2 = 10 - 8 + 2 = 4$$

Fan-In Fan-Out Complexity (1)

- Taking into account the data flow (Henry and Kafura)

  Number of data streams and global data structure

- Uses a length like SLOC or McCabe Cyclomatic Complexity

$$HK = Length \times (Fan_{in} \times Fan_{out})^2$$

with incoming (Fan_{in}) and outgoing (Fan_{out}) local information

- Variation by Shepperd without multiplying by a length

$$S = (Fan_{in} \times Fan_{out})^2$$

Fan-In Fan-Out Complexity (2)

- Information flow from procedure A to B
  - A calls B
  - B calls A and uses the returned value
  - A and B called by C, which passes return value from A to B
- Definition of incoming and outgoing data flows
  - Fan_{in} = procedures called by it + read parameters + global variables accessed
  - Fan_{out} = procedures calling this one + output parameters + global variables written

Fan-In Fan-Out Complexity (3)

- **Advantages**
  - Can be evaluated before having the implementation
  - Take into account the programs controlled by data
- **Disadvantages**
  - Zero complexity for procedure without external interaction

Fan-In Fan-Out Complexity Example

```c
char * strncat(char *ret, const char *s2, size_t n) {
    char *s1 = ret;
    if (n > 0) {
        while (*s1)
            s1++;
        while (*s1++ = *s2++) {
            if (--n == 0) {
                *s1 = '\0';
                break;
            }
        }
    }
    return ret;
}
```

- **Input** $(fan_{in} = 3)$
- **Output** $(fan_{out} = 1)$
- **Unweighted Fan-In Fan-Out complexity** : $S = 3^2 = 9$
- **Weighted Fan-In Fan-Out complexity** : $HK = 10 \times 9 = 90$

Measuring Modularity

- Evaluation of coupling and cohesion of modules
  - Fan_{in} of M counts modules calling functions from M
  - Fan_{out} of M counts modules called by M
- Modules with a zero Fan_{in} suspicious
  - Dead code
  - Outside the borders of the system
  - Approximations of the notion of call is not precise enough

Complexity Metric

- Three main complexity metrics
  - Measuring the effort with Halstead
  - Measuring the structure with McCabe
  - Measuring information flow with Henry and Kafura/Shepperd
- Metrics developed for imperative languages

  Can nevertheless be used with object-oriented programming

Standard Code Metric

Cyclomatic Complexity

- Cyclomatic complexity of a function or method

  Number of linear paths in a function

- Not always measurable by static analysis tools

  Estimate with number of if, while, for, etc. statements

- Should not exceed a value of 10 on average
  - Decrease in readability and understanding by others
  - Less debug information, less accurate stack trace
  - More complex and less efficient unit tests

Source Line of Code

- Number of source line of code (SLOC)
  - “Physical” lines present directly in the file
  - “Logical” lines actually executed
- Boehm classification to interpret with caution

  Small (S) : 2 KLOC, Intermediate (I) : 8, Medium (M) : 32
  Large (L) : 128, Very Large (VL) : 512
- Metric to use with great care

  Do not measure the production effort, nor the value of software

Density of Comment

- Density of comment (DC) regarding lines of code

  DC = SLOC / CLOC (comment line of code)

- Number of comment lines do not define their quality

  Between 20% and 40% seems normal, otherwise suspicious

- Similar precautions with SLOC to take

  In addition, well-written code is its own documentation

Code Coverage

- Proportion of source code covered by tests

  Number of run through statements, methods, classes, etc.

- Usually automated tests, but also manual ones

  Covers unit, functional and validation tests

- Decrease the runtime errors probability or bugs

  Easier to evolve, maintain, refactor thanks to tests

Code Duplication

- Repetition of similar or identical code in a source code

  Clear violation of the DRY principle (don't repeat yourself)

- Four main types of code duplication

  Imposed, inadverttently, impatiently and interdeveloper

- Decreased code maintainability

  Increases the risk of introducing bugs

Coupling

- Coupling measures the links existing between classes
  - **Afferent** (Ca): number of references to measured class

    Only external references
  - **Efferent** (Ce): number of types that the class knows

    Inheritance, implementation, parameter, variable, exception, etc.
- Class referenced a lot (afferent) is important

  Big impact of a modification, but good reuse
- Violation of single responsibility if large efferent coupling

  Potentially high instability with increasing dependencies

Instability

- Instability of a module measures its resistance to change

  A stable module is difficult to change

- Measured by the efferent over total coupling ratio

$$\text{Instability} = \frac{Ce}{Ce + Ca}$$

- Quality code easy to modify thanks to instability

  Very stable from 0.0 to 0.3 or very unstable from 0.7 to 1.0

Abstractness

- Abstraction level compared to other classes

  Ratio between internal abstract types and other internal types

- High abstraction recommended in a very stable class

  Fully concrete (1.0) or fully abstract (1.0) module

- Metric usually not used alone

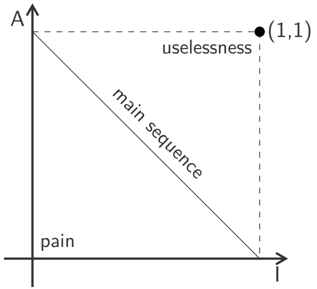
  Often combined with instability, for example

Distance from Main Sequence (1)

- Balance of a module between abstractness and instability

$$D = |A + I - 1|$$

- Non-standard measure by dividing the result by $\sqrt{2}$



Distance from Main Sequence (2)

- Oblique main sequence line gives the good situations
  - Very stable class should be abstract
  - Very unstable class should be concrete
- As close as possible to the main sequence when $D \to 0$

  Value greater than 0.7 can indicate an issue

Lack of Cohesion of Methods (1)

- Lack of cohesion of methods (LCOM)

$$LCOM = 1 - \frac{\sum_F MF}{M \times F}$$

with M methods, F instance fields, and MF methods calling a given field

- A class should have only one reason to change
  - Makes it possible to evaluate single responsibility principle
  - Cohesion decrease if little common things in the features
- Strong cohesion many methods/instance fields references

  $LCOM > 0.8$ and $F, M > 10$ is suspicious

Lack of Cohesion of Methods (2)

- Other measures for the lack of cohesion of methods

$$LCOM\ HS = \left(M - \frac{\sum_F MF}{F} \right) \times (M - 1)$$

with M methods, F instance fields, and MF methods calling a given field

- Henderson-Sellers variation simplified and normalised

Value higher than 1 is suspicious

Relational Cohesion

- Average number of internal relations to a module

$$H = \frac{R + 1}{N}$$

with R internal references to the module, and N types contained in it

- Large value if strong relation between classes of a module

  Value typically between 1.5 and 4.0

Instance Size

- Measure the amount of memory used for an instance

  Number of memory bytes to stored an instantiated object

- Sum of sizes of class fields and inherited ones

  Can be statically computed

- Large objects can degrade performance

  Maximum recommended value is 64

Specialisation Index

- Specialisation index of a class

$$\frac{NORM \times DIT}{NOM}$$

with $NORM$ redefined methods, NOM methods,
and DIT depth inheritance tree

- Increase of the index with the depth and redefinitions

  Value greater than 1.5 suspicious

Number of Elements

- Count a number of elements in a class
  - **Parameters** of a method to be limited to 5
  - **Variables** declared in a method to be limited to 8
  - **Overload** of methods to be limited to 6
- Simplification using structured data

  Class, structure, tuple, dictionary, etc.

Coding Rule Violation

- Count number of coding rules violated

  Rules often specific to a given programming language

- Rules covering several categories of evaluated criteria

  Maintainability, reliability, efficiency, portability, usability

- Some alerts do not necessarily represent a bug

  Not necessary to correct all of them

Quality Code (1)

- Optimal values to reach for standard metrics

  These are average values outside of which it is suspicious

Metric	**Optimal value**
Cyclomatic Complexity (CC)	10
Source Line of Code (SLOC)	$>$ 20 difficult to understand, $>$ 40 complex
Density of Comment	between 20%–40%
Code Coverage	100%
Code Duplication	0%
Distance from Main Sequence (D)	< 0.7
Lack of Cohesion of Methods (LCOM)	< 0.8 (with $F, M < 10$)
Lack of Cohesion of Methods (LCOM HM)	< 1

Quality Code (2)

- Optimal values to reach for standard metrics

  These are average values outside of which it is suspicious

Metric	**Optimal value**
Relational Cohesion (H)	between 1.5–4.0
Instance size (bytes)	< 64
Specialisation index	< 1.5
Number of Method Parameters	≤ 5
Number of Method Local Variables	≤ 8
Overloaded Versions of a Method	≤ 6

Static Code Analysis

- Static code analysis performed with a syntactic analyser

  Easy since any programming language has such a parser

- Several problems with measuring quality
  - Not often a real intuition for most metrics
  - Ignore environment, domaine of application, particular algorithms, users, etc.
  - Easy to get around by an obscure code

Object Oriented System

Object Oriented System (1)

- Methods Weighted by Class

$$WMC = \sum_{i=1}^{n} c_i$$

with $M_1, ..., M_n$ *methods with complexities* $c_1, ..., c_n$

- Depth of the inheritance tree of a class
  - DIT the maximum in case of multiple inheritance
  - Complex reusability and maintainability if large DIT

Object Oriented System (2)

- Number of children (direct) of a class

  Should be minimised otherwise the design is considered bad

- Coupling between classes when calling methods/variables
  - An encapsulated design will give a small *CBO*
  - An independent class is easy to test and reuse

- Class response to external solicitations

  Number of methods that can be executed following a message

Object Oriented System (3)

- Number of variables by class (NVC)

  Average number of public and private variables by class

- Number of parameters by method (APM)

  Number of parameters divided by number of methods (< 0.7)

- Number of objects (NOO)

  Number of objects extracted from source code

MOOD Metric

- Metrics proposed by the MOOD project team in 1994

  Under the direction of Abreau

- Complete system level to measure several aspects
  - **Encapsulation**: with method/attribute hiding factor
  - **Inheritance**: with method/attribute inheritance factor
  - **Polymorphism**: with polymorphism factor
  - **Coupling**: with coupling factor

Encapsulation (1)

- Measure of variables and methods encapsulation

$$MHF = \frac{\sum_{i=1}^{M}(1 - V(M_i))}{M}$$

with M methods with visiblity $V(M_i)$ each

- Measure of the visibility of a method

$$V(M_i) = \frac{\#\{C_j \mid \text{classe } C_j \text{ can call } M_i \text{ and } M_i \text{ not in } C_j\}}{C - 1}$$

with C classes throughout the system

Encapsulation (2)

- 100% MHF if all the methods are private

  Tend to 0% when the number of public methods increases

- Hiding methods is a good practice
  - Increasing reusability and decreasing complexity
  - A low MHF indicates an implementation not abstract enough
  - A high MHF reflects a low number of features

- Increasing MHF decreases bug density and increases quality

  Acceptable values between 8% and 25%

Inheritance

- Inheritance method factor of a class

$$MIF = \frac{\sum_{i=1}^{C} Mi(C_i)}{\sum_{i=1}^{C} Ma(C_i)}$$

with $Mi(C_i)$ methods inherited by C_i and not redefined,
and $Ma(C_i)$ methods in C_i

- Acceptable values for method/attribute inheritance factors
  - Between 20% and 80% for MIF
  - AIF should be between 0% and 48%

Polymorphisme

- Polymorphism factor based on redefinition

$$PF = \frac{\sum_{i=1}^{C} Mo(C_i)}{\sum_{i=1}^{C}(Mn(C_i) \times DC(C_i))}$$

with $Mo(C_i)$ redefined methods in C_i, $Mn(C_i)$ new methods defined in C_i, and DC ancestors of class C_i

- Indirect measure of the dynamic link in a system

  Opportunities for redefinition $Mn(C_i) \times DC(C_i)$

Coupling

- A is coupled to B if A calls methods/variables in B

  Does no take into account coupling by inheritance

- Coupling of a class with another class

$$CF = \frac{\sum_{i=1}^{C} \sum_{j=1}^{C} is\_client(C_i, C_j)}{C(C-1)}$$

$$with\ is\_client(A, B) = \begin{cases} 1 & if\ A \neq B\ and\ A\ coupled\ to\ B \\ 0 & otherwise \end{cases}$$

- Increasing CF increases density of defects

  Rework effort to find and fix defects increases

References

- Stackify, *What Are Software Metrics and How Can You Track Them?*, September 16, 2017. https://stackify.com/track-software-metrics
- Steven A. Lowe, *Why metrics don't matter in software development (unless you pair them with business goals)*, June 1, 2016. https://techbeacon.com/app-dev-testing/why-metrics-dont-matter-software-development-unless-you-pair-them-business-goals
- Steven A. Lowe, *9 metrics that can make a difference to today?s software development teams*, June 6, 2016. https://techbeacon.com/app-dev-testing/9-metrics-can-make-difference-todays-software-development-teams
- TutorialsPoint, *Software Design Complexity*. https://www.tutorialspoint.com/software_engineering/software_design_complexity.htm
- ndepend, *Code Metrics Definitions*. https://www.ndepend.com/docs/code-metrics
- Manikanta Pattigulla, *Measure Your Code Using Code Metrics*, July 31, 2017. https://www.c-sharpcorner.com/article/measure-your-code-using-code-metrics

Credits

- Mad House Photography, May 17, 2009, https://www.flickr.com/photos/mad_house_photography/4311409835.
- Dany Sternfeld, September 9, 2018, https://www.flickr.com/photos/sternfeld/43661676445.
- TutorialsPoint, https://www.tutorialspoint.com/software_engineering/images/cyclomatic_complexity.png.
- Alain Bachellier, April 8, 2006, https://www.flickr.com/photos/alainbachellier/125739388.
- MitsukoTonouchi, January 8, 2011, https://www.flickr.com/photos/nomadcrocheter/5334037661.