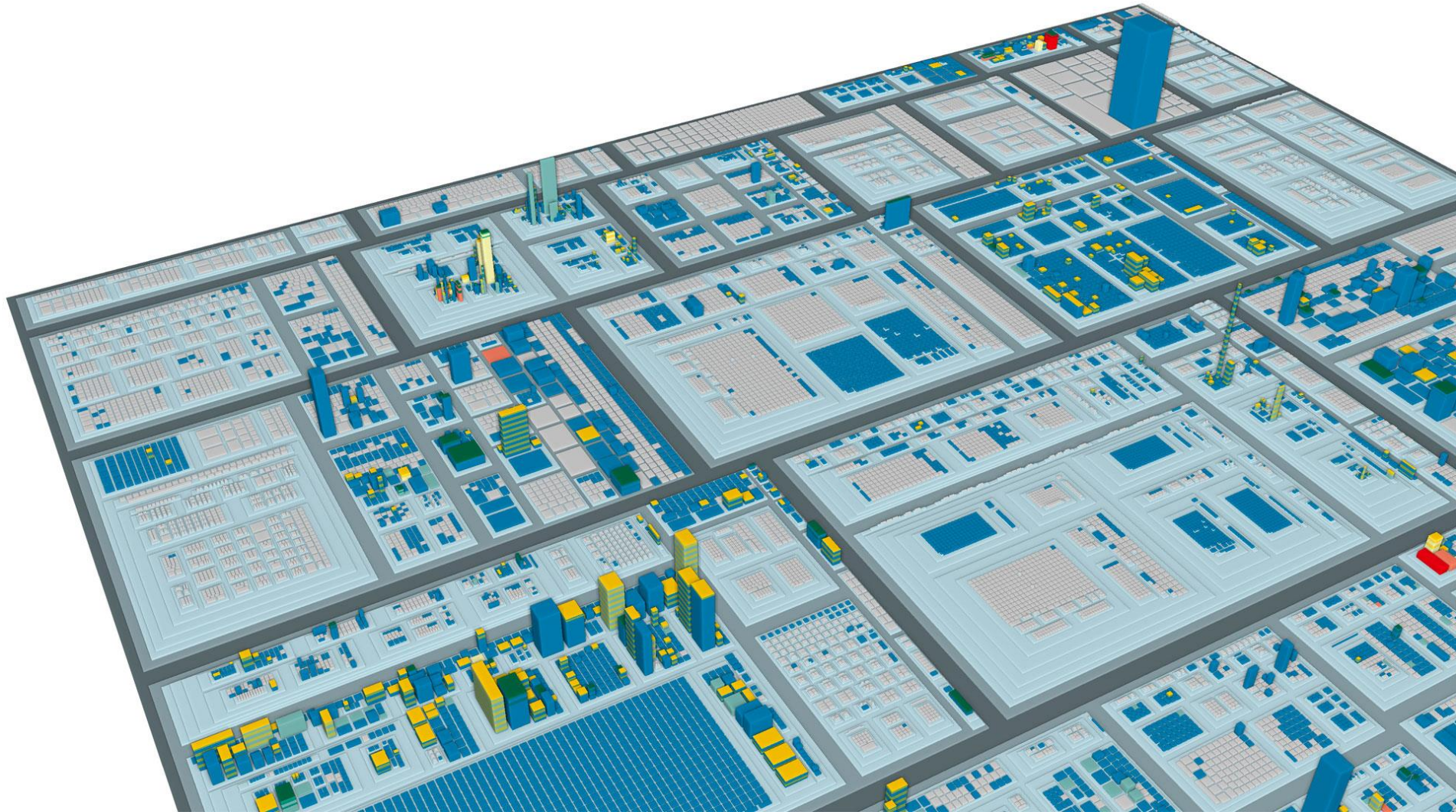
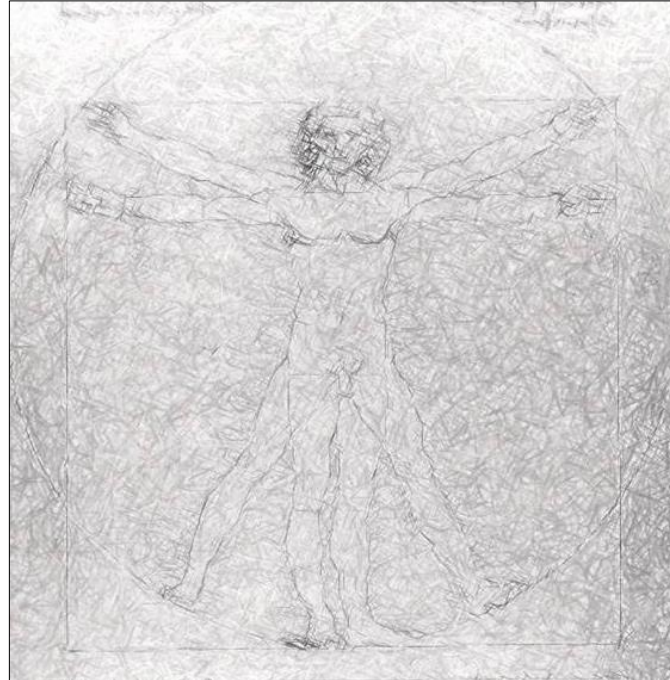


Automated Visual Software Analytics

OpenHPI – Prof. Dr. Jürgen Döllner – 2015

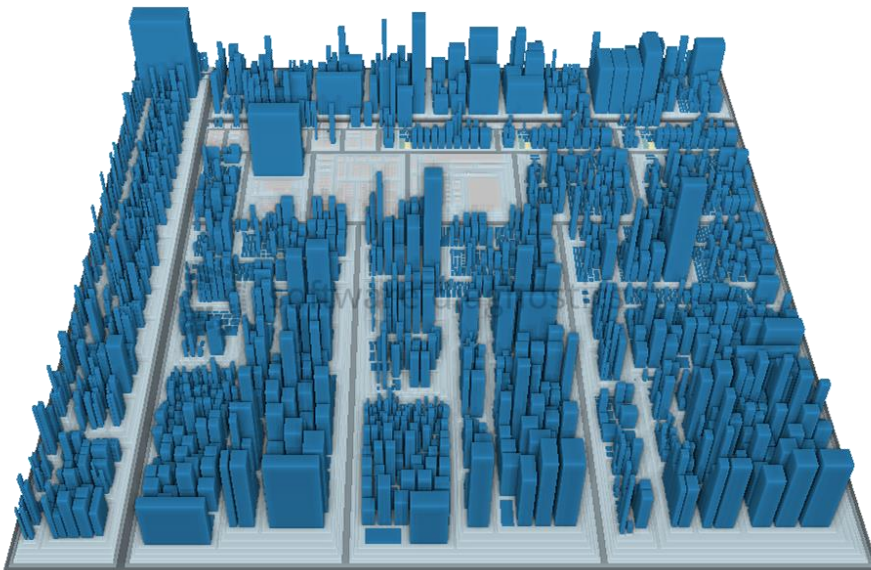




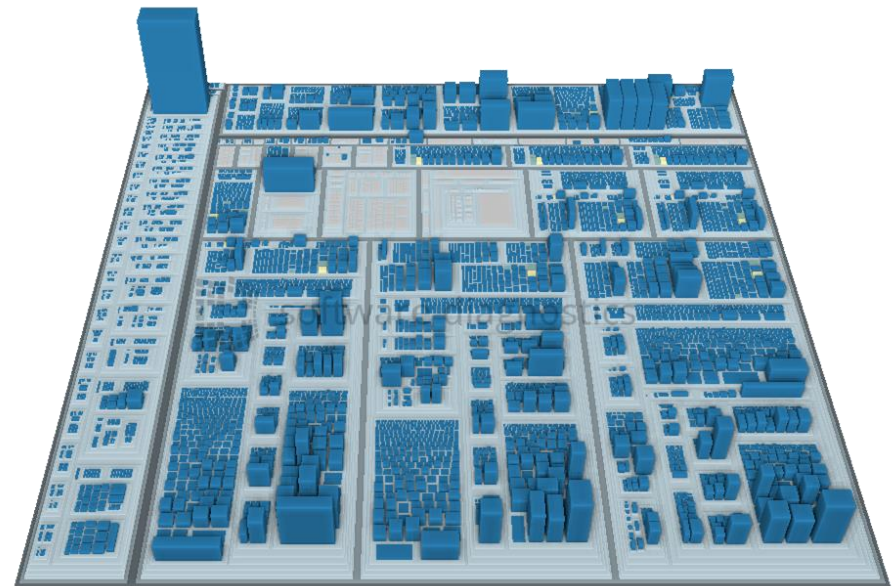
Principles of Software Metrics
Software Quality Metrics
Limitations of Software Metrics

Example: Software map of a software system, visualizing software quality metrics

Nesting-level per module



McCabe code complexity per module



"The purpose of software metrics is to make assessments throughout the software life cycle as to whether the software quality requirements are being met. The use of software metrics reduces subjectivity in the assessment and control of software quality by providing a quantitative basis for making decisions about software quality. However, the use of software metrics does not eliminate the need for human judgment in software evaluations. The use of software metrics within an organization or project is expected to have a beneficial effect by making software quality more visible."

– IEEE Standard 1061, 1998

Definition

- A *software metric* represents a quantitative measure of a degree to which a software system, component, or process possesses a defined property or characteristic.
 - Software metrics are applied by corresponding *measurement techniques*.
The results are *measured values*.
 - For example, Lines-of-Code (LOC) is a software metric that measures the size of a source code unit by counting source code lines.
- A *software quality metric* refers to a software metric that characterizes quality aspects.
- The interpretation of measured values of software metrics has a strong empirical character.

General Types of Software Metrics

- **Product metrics** characterize an intermediate or final product of the software development process. Examples include code size, code complexity, and customer satisfaction.
- **Process metrics** characterizes methods, techniques, and tools employed in developing, implementing, and maintaining the software system. Examples include the effectiveness of defect removal during development or the response time of bug fix processes.
- **Project metrics** characterize the project in terms of time, budget, and management. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity.
- A specific software metrics may belong to multiple categories.

– Adapted from Stephen H. Kan: "Metrics and Models in Software Quality Engineering", Pearson Edu., 2003.

Requirements for software metrics include:

- **Simplicity:** interpretation should be possible with appropriate effort
- **Validity:** measurements should be strongly correlated with the measured property
- **Stability:** measurements should be stable against small changes
- **Timeliness:** measurements should be taken within a reasonable time frame
- **Analyzability:** measurements should support statistical/mathematical analysis
- **Reproducibility:** measurements should be objective, that is, without subjective factors

Code Complexity Measures

- Derived by static source code analysis
- Relate to and reflect aspects of the coding quality, coding complexity, and modularization

Examples

- Lines-of-Code (LOC, NLOC)
- Nesting Level (NL)
- Cyclomatic Complexity (McCabe)
- Halstead Complexity
- Coupling
- Cohesion
- Code Clones
- Function Points

Dynamics-Related Measures

- Analyzed based on recording system dynamics at run-time
- Relate to and reflect aspects of the system behavior

Examples

- Code Coverage
- Bugs-per-Line-of-Code
- Feature Usage

Organizational Measures

- Analyzed based on information about the development activities and developers.
- Relate and reflect aspects of the system evolution and changes applied to the system over time

Examples

- Number of Engineers (NOE)
- Number of Ex-Engineers (NOEE)
- Edit Frequency (EF)
- Depth of Master Ownership (DMO)

Limitations

- It is difficult to find general “normal ranges” for measured values of individual software quality metrics. Normal ranges can be empirically determined for a specific software project.
- Simplistic measurements may cause “more harm than good”. Measured values suggest precision. But there is “a strong tendency for professionals to display over-optimism and over-confidence”.

– Carolyn Mair, Martin Sheppard: “Human Judgement and Software Metrics: Vision for the Future”, ICSE ’11, 2011

- Developers can adapt their work to software quality metrics (if used as strict software quality instrument) by metric-related opportunistic behavior.

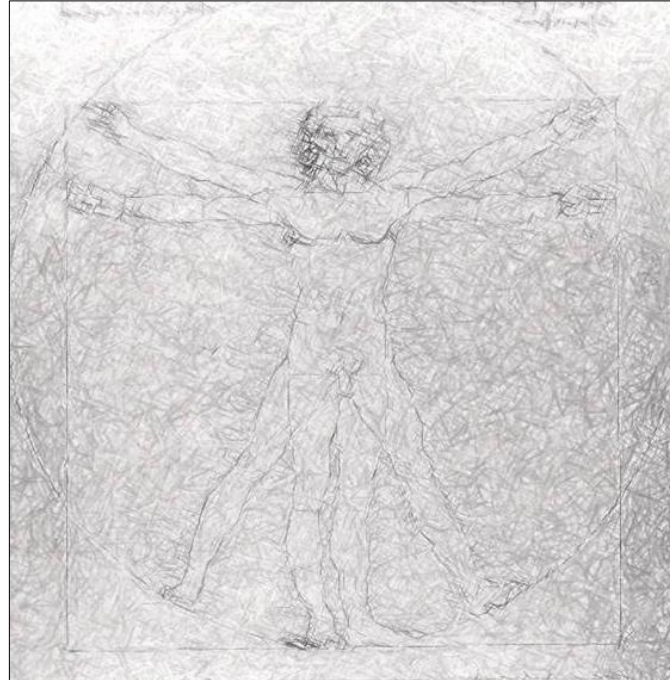
Software metrics are essential to quantify aspects of software quality regarding the product, process, and project aspects. Software metrics have an **empirical nature**, that is, they have to be interpreted for a given software system and within the frame of experience of a development team.

There is a large collection of **software metric tools** that perform software metrics measurements, either as a separate tool or integrated into a development environment.

Most common programming languages are supported.

For **software analytics**, software metrics represent analysis techniques that deliver valuable indicators about the status of the software system's implementation, its behavior and dynamics, and its software development process and organizational development.

Software analytics tools (e.g., Software Diagnostics software maps) fully automate software metric measurements as part of the overall analytics processes.



Principles of Software Metrics
Software Quality Metrics
Limitations of Software Metrics

Software Metric

"A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute of a defined property. Software metric measures deliver key performance and quality indicators to software analytics."

Source: IEEE Standard 1061, 1998

Software Quality

“Software quality is the degree to which software possesses a desired combination of attributes.”

Software Quality Metric

“A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.”

Source: IEEE Standard 1061, 1998

Further Reading



Ming-Chang Lee, To Chang: "Software Measurement and Software Metrics in Software Quality", International Journal of Software Engineering and Its Applications. 7(4), 2013.



Stephen H. Kan: "Metrics and Models in Software Quality Engineering", Pearson Education Inc., 2003.

Software metrics rely on empirical observations; they are not laws of physics.

There is not a single ultimate software metric for complex computer-based systems. Each software metric measures only one aspect that can help to determine and interpret aspects of software quality.

In a complex computer-based system, there are most likely measures of different software metrics outside the normal ranges for some modules – if there are good reasons. Due to the nature of a module, due to its development history, due to its status, or due to technical or organizational constraints, there can be such good reasons.

A single software metric does not give too much insights; combined and monitored over a period of time, they do. Different metrics, if measured frequently and combined to higher-level metrics, start to provide effective insights. This core task is up to software analytics.

```
#include <stdlib.h>    // card > aek.ppm
#include <stdio.h>
#include <math.h>
typedef int i;typedef float f;struct v{
f x,y,z;v operator+(v r){return v(x+r.x
,y+r.y,z+r.z);}v operator*(f r){return
v(x*r,y*r,z*r);}f operator%(v r){return
x*r.x+y*r.y+z*r.z;}v(){}v operator^(v r
){return v(y*r.z-z*r.y,z*r.x-x*r.z,x*r.
y-y*r.x);}v(f a,f b,f c){x=a;y=b;z=c;}v
operator!(){return*this*(1/sqrt(*this%*
this));};i G[]={247570,280596,280600,
249748,18578,18577,231184,16,16};f R(){
return(f)rand()/RAND_MAX;}i T(v o,v d,f
&t,v&n){t=1e9;i m=0;f p=-o.z/d.z;if(.01
<p)t=p,n=v(0,0,1),m=1;for(i k=19;k--;)
for(i j=9;j--;)if(G[j]&1<<k){v p=o+v(-k
,0,-j-4);f b=p*d,c=p*p-1,q=b*b-c;if(q>0
){f s=-b-sqrt(q);if(s<t&&s>.01)t=s,n=!
(p+d*t),m=2;}}return m;}v S(v o,v d){f t
;v n;i m=T(o,d,t,n);if(!m)return v(.7,
.6,1)*pow(1-d.z,4);v h=o+d*t,l=(v(9+R(
),9+R(),16)+h*-1),r=d+n*(n*d*-2);f b=1%
n;if(b<0||T(h,l,t,n))b=0;f p=pow(1%r*(b
>0),99);if(m&1){h=h*.2;return(i)ceil(
h.x)+ceil(h.y)&1?v(3,1,1):v(3,3,3))*(b
*.2+.1);}return v(p,p,p)+S(h,r)*.5;}i
main(){printf("P6 512 512 255 ");v g=!v
(-6,-16,0),a=(v(0,0,1)^g)*.002,b=(g^a
)*.002,c=(a+b)*-256+g;for(i y=512;y--;)
for(i x=512;x--;){v p(13,13,13);for(i r
=64;r--;){v t=a*(R())-.5)*99+b*(R())-.5)*
99;p=S(v(17,16,8)+t,!(t*-1+(a*(R()+x)+b
*(y+R()+c)*16))*3.5+p);}printf("%c%c%c"
,(i)p.x,(i)p.y,(i)p.z);}}
```

Definition of LOC Metrics

LOC Metric Properties

LOC Metric Tools

Definition

The *lines-of-code (LOC) metrics* is a set of software metrics that quantitatively measure source code size by counting “source code lines”.

- What is considered to be a “source code line” depends on the specific type of LOC metric.
- LOC metrics can be applied to all types of code units such as functions, methods, classes, packages, or files.
- LOC convey an order of magnitude of a given code unit
- Computed by static source code analysis
- Also measured in 1000-units, abbreviated by **KLOC**

- There are different variants of LOC metrics according to the definition of “source code line”:
 - **LOC** – physical lines: a line corresponds to a line in the source code's text representation
 - **SLOC** – source code lines of code, without comment and blank lines
 - **CLOC** – comment lines of code
 - **NCLOC** – non comment lines of code; does exclude comments, headers and footers such as include statements, braces of function bodies, etc.
 - **LLOC** – logical lines of code, counts the number of statements by parsing the source code, regardless of their layout within a single line or across many lines
 - **BLOC** – blank lines: lines without text, used to format the source code
- All LOC metrics can be computed in an easy, fast, and robust way.

Definition of LOC Metrics

Example 1:

- LOC = 8
- SLOC = 6
- NCLOC = 4

```
01 void GraphicsItem::setScheme(ColorScheme * scheme)
02 {
03     m_scheme = scheme;
04
05     //call the update methods
06     updateBrushes();
07     updateTooltips();
08 }
```

Definition of LOC Metrics

Example 2: LOC of the Linux Operating Systems as Order-of-Magnitude Measure

Version	Date	LOC
2.6.20	Feb 2007	8,1 M
2.6.21	Apr 2007	8,2 M
2.6.24	Jan 2008	8,8 M
2.6.29	Mar 2009	11,0 M
2.6.33	Feb 2010	12,9 M
2.6.37	Jan 2011	13,9 M
3.2	Jan 2012	14,9 M
3.8	Feb 2013	16,4 M
3.13	Jan 2014	17,9 M
3.18	Dec 2014	18,9 M

LOC Applications

- LOC metrics are used to determine the “order of magnitude” of the implementation of a complex computer-based system; the order of magnitude is one key weight used by predictive software analytics
- LOC metrics serve as a statistical measurements used to estimate and predict efforts.
- LOC metrics are used to build higher level metrics, for example, to estimate general cost and efforts (e.g., for maintenance, error handling, bug fixing).

LOC Applications

- The ratio of comment lines and source code lines (CLOC/LOC) reflects the degree to which source code is documented by comments.
- LOC metrics measure “how much” of the source code of a system is currently (or was in the past) modified, under revision, added, or removed within a given time frame, that is, LOC metrics can be used to quantitatively determine the degree to which source code changed or evolved over time.
- LOC measures “how much” individual developers contributed to a system's implementation in given time frame.

Disadvantages

- LOC can be **correlated with effort but not with functionality**, that is, it does not tell much about the real underlying complexity of an implementation.
For example, a computational geometry algorithm sometimes requires months to be developed but results in less than 1.000 LOC.
- LOC does not consider complexity that results from **automatic code generation** using metaprogramming, generic programming, or frameworks.
- If LOC is used as **measure of activity**, developers can easily undermine that measure by “opportunistic coding styles” such as writing long-winded code.
- Implementations of complex computer-based systems frequently tend to use different languages – LOC by its nature **depends on the programming language** to a certain degree (way of writing comments, functions, methods, classes, etc.). For that reason, it is difficult to compare LOC of source code units written in different programming languages.

- **CLOC** is a tool that counts blank lines, comment lines, and physical lines of source code in a large number of supported programming languages. Given two versions of a code base, cloc can compute differences in blank, comment, and source lines.

<http://cloc.sourceforge.net/>

- **LocMetrics** is a free tool that “counts total lines of code (LOC), blank lines of code (BLOC), comment lines of code (CLOC), lines with both code and comments (C&SLOC), logical source lines of code (SLOC-L), McCabe VG complexity (MVG), and number of comment words (CWORDS). Physical executable source lines of code (SLOC-P) is calculated as the total lines of source code minus blank lines and comment lines. Counts are calculated on a per file basis and accumulated for the entire project. LocMetrics also generates a “comment word histogram.”

<http://www.locmetrics.com/>

The **LOC metrics** represent base quantitative measurements of source code size. They are computed based on a static analysis of the source code by counting the number of lines of code. Different types of LOC metrics focus on different types of “source code lines”.

The LOC metric represents a base measure for the general size of a system's implementation. As **a statistical weight factor**, it is used to estimate or forecast maintenance efforts. LOC, in general, should be used in a differential perspective, that is, it measure incremental change, or as a statistical measure.

Software analytics uses the LOC metrics as base measures for source code size to weight, evaluate, interpret or benchmark other key performance indicators and software metrics.

```
#include <stdlib.h>    // card > aek.ppm
#include <stdio.h>
#include <math.h>
typedef int i;typedef float f;struct v{
f x,y,z;v operator+(v r){return v(x+r.x
,y+r.y,z+r.z);}v operator*(f r){return
v(x*r,y*r,z*r);}f operator%(v r){return
x*r.x+y*r.y+z*r.z;}v(){}v operator^(v r
){return v(y*r.z-z*r.y,z*r.x-x*r.z,x*r.
y-y*r.x);}v(f a,f b,f c){x=a;y=b;z=c;}v
operator!(){return*this*(1/sqrt(*this%*
this));};i G[]={247570,280596,280600,
249748,18578,18577,231184,16,16};f R(){
return(f)rand()/RAND_MAX;}i T(v o,v d,f
&t,v&n){t=1e9;i m=0;f p=-o.z/d.z;if(.01
<p)t=p,n=v(0,0,1),m=1;for(i k=19;k--;)
for(i j=9;j--;)if(G[j]&1<k){v p=o+v(-k
,0,-j-4);f b=p*d,c=p*p-1,q=b*b-c;if(q>0
){f s=-b-sqrt(q);if(s<t&&s>.01)t=s,n=!
(p+d*t),m=2;}}return m;}v S(v o,v d){f t
;v n;i m=T(o,d,t,n);if(!m)return v(.7,
.6,1)*pow(1-d.z,4);v h=o+d*t,l=(v(9+R(
),9+R(),16)+h*-1),r=d+n*(n*d*-2);f b=1%
n;if(b<0||T(h,l,t,n))b=0;f p=pow(1%r*(b
>0),99);if(m&1){h=h*.2;return(i)ceil(
h.x)+ceil(h.y)&1?v(3,1,1):v(3,3,3))*(b
*.2+.1);}return v(p,p,p)+S(h,r)*.5;}i
main(){printf("P6 512 512 255 ");v g=!v
(-6,-16,0),a=(v(0,0,1)^g)*.002,b=(g^a
)*.002,c=(a+b)*-256+g;for(i y=512;y--;)
for(i x=512;x--;){v p(13,13,13);for(i r
=64;r--;){v t=a*(R())-.5)*99+b*(R())-.5)*
99;p=S(v(17,16,8)+t,!(t*-1+(a*(R()+x)+b
*(y+R()+c)*16))*3.5+p);}printf("%c%c%c"
,(i)p.x,(i)p.y,(i)p.z);}}
```

Definition of LOC Metrics

LOC Metric Properties

LOC Metric Tools

LOC

Lines-of-code is a quantitative software metric measuring the size of a source code unit by counting the lines of source code according to a specific definition of "source code line".

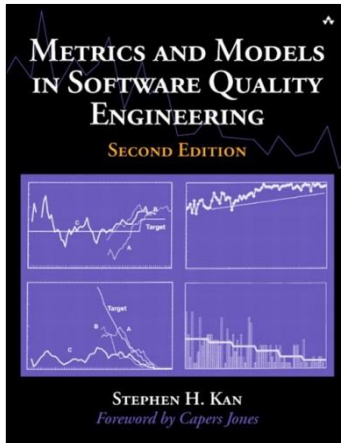
SLOC, CLOC, NCLOC, LLOC, BLOC

Specialized LOC metrics for source code lines with comments and blank lines (SLOC), comment lines (CLOC), non-comment and non-header/footer lines (NCLOC), logical source code "lines" in terms of statements (LLOC), and blank lines (BLOC).

KLOC

Abbreviation for 1000-units of LOC.

Further Reading



Stephen Kan: "Metrics and Models in Software Quality Engineering", Addison Wesley, 2002 (revised version 2014)

See also: Google Books version (pp. 88ff)

LOC metrics are useful to monitor and observe how systems evolve over a period of time. The LOC of different versions of a system can help to verify hypothesis about the development progress and development tendencies as well as given insights into certain software quality aspects such as documentations and comments.

“I wrote x KLOC last time!” or “I own x KLOC of the code” – these are examples of how LOC should not be used (danger of malpractice).

LOC is frequently misinterpreted. For example, it does NOT make sense to correlate software developer productivity with LOC. With respect to an individual developer, LOC can measure, for example, how many lines a developer wrote, modified, or managed but this is not a qualitative statement.

Code Duplication



Code Duplicates
Code Similarity
Consequences of Code Clones

Repeated code sequences

- result from copying and pasting code ("code clones")
- result from independent development activity
- tend to evolve independently
- are not explicitly visible or recorded
- violate the principle "Don't repeat yourself" (DRY)



Reasons for Code Duplicates

Reasons that lead to code duplications:

- copy & paste, forking, templating, customization of codes
- insufficient information on a global project scale
- badly organized reuse process (type-4 clones)
- questionable productivity measures (LOCs per day)
- time pressure
- educational deficiencies, ignorance, or shortsightedness
- intellectual challenges (e.g., generics)
- professionalism/end-user programming (e.g., HTML, Visual Basic, etc.)
- development process
- organizational issues, e.g., distributed development organizations



R. Koschke: "Survey of Research on Software Clones"

Program representation based similarity

- Text-based clone search (syntactic matches)
- Token-based clone search (robust against reformatting, documentation changes, or renaming of variables, classes, or methods)
- Abstract Syntax Tree (AST) approaches (robust against differences in keywords or parentheses)
- Normalization can further help to detect equivalent code segments
- Characteristic vectors can be computed for code fragments (high-dimensional vectors); the distance of those vectors can be used as measure for similarity.

Behavioral similarity

E. Juergens, F. Deissenboeck, B. Hummel: "Code Similarities Beyond Copy & Paste", 2010

- For a code sequence, all variables that read are denoted as *input variables*, all variables that are written are denoted by *output variables*
- The code segment is interpreted as a function from valuations of input variables to valuations of output variables (state-less, side-effect free)
- Two code segments are called **behaviorally equal**, if and only if they have the same sets of input and output variables (modulo renaming) and are equal with respect to their function interpretation.

Clone types can be categorized based on both textual and semantic similarities:

C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Science of Computer Programming, vol. 74, no. 7, pp. 470–495, May 2009

- **Type-1:** Identical code fragments except for variations in whitespace, layout and comments.
- **Type-2:** Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.
- **Type-3:** Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.
- **Type-4:** Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Clone Types – Type-1 Example

Type-1: Identical code fragments except for variations in whitespace, layout and comments:

```
template <>
void Value<const gl::GLubyte *>::printOn(std::ostream & stream) const
{
    std::string s = wrapString(reinterpret_cast<const char*>(value));
    stream.write(s.c_str(), static_cast<std::streamsize>(s.size()));
}
```

```
template <>
void Value<const gl::GLchar *>::printOn(std::ostream & stream) const
{
    std::string s = wrapString(reinterpret_cast<const char*>(value));
    stream.write(s.c_str(), static_cast<std::streamsize>(s.size()));
}
```

Clone Types – Type-2 Example

Type-2: Syntactically identical fragments, variations in identifiers, parameters, etc.:

```
{
    QImage image("data/hpicgs_label.png");
    DistanceTransform DT(image, 512, 128, 0.0625f);
    glGenTextures(1, &m_hpicgsLabelDM);
    glBindTexture(GL_TEXTURE_2D, m_hpicgsLabelDM);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F, 512, 128, 0, GL_RED, GL_FLOAT, DT.data());
    glGenerateMipmap(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, 0);
}

{
    QImage image("data/companion_cube_label.png");
    DistanceTransform DT(image, 256, 64, 0.0625f);
    glGenTextures(1, &m_portccLabelDM);
    glBindTexture(GL_TEXTURE_2D, m_portccLabelDM);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F, 256, 64, 0, GL_RED, GL_FLOAT, DT.data());
    glGenerateMipmap(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Clone Types – Type-4 Example

Type-4: Two or more code fragments, same computation but different syntactic variants:

```
glBegin(GL_QUADS); // Begin drawing a cube with quads
    glVertex3f( 1.f, 1.f,-1.f); glVertex3f(-1.f, 1.f,-1.f); // top
    glVertex3f(-1.f, 1.f, 1.f); glVertex3f( 1.f, 1.f, 1.f);
    glVertex3f( 1.f,-1.f, 1.f); glVertex3f(-1.f,-1.f, 1.f); // bottom
    glVertex3f(-1.f,-1.f,-1.f); glVertex3f( 1.f,-1.f,-1.f);
    glVertex3f( 1.f, 1.f, 1.f); glVertex3f(-1.f, 1.f, 1.f); // front
    glVertex3f(-1.f,-1.f, 1.f); glVertex3f( 1.f,-1.f, 1.f);
    glVertex3f( 1.f,-1.f,-1.f); glVertex3f(-1.f,-1.f,-1.f); // back
    glVertex3f(-1.f, 1.f,-1.f); glVertex3f( 1.f, 1.f,-1.f);
    glVertex3f(-1.f, 1.f, 1.f); glVertex3f(-1.f, 1.f,-1.f); // left
    glVertex3f(-1.f,-1.f,-1.f); glVertex3f(-1.f,-1.f, 1.f);
    glVertex3f( 1.f, 1.f,-1.f); glVertex3f( 1.f, 1.f, 1.f); // right
    glVertex3f( 1.f,-1.f, 1.f); glVertex3f( 1.f,-1.f,-1.f);
glEnd(); // Done drawing a cube with quads
```

vs. same visual result, but optimized/obfuscated source code:

```
static const GLshort v [] =
    { -1, -1, -1, -1, -1, 1, -1, 1, -1, -1, 1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1 };
glBegin(GL_TRIANGLE_STRIP);
    for (GLuint64 i(0 x7b79d5f02b64); i /= 10 ul; glVertex3sv(&v[i % 10 ul * 3]));
glEnd();
```

Count Matrix Clone Detection (CMCD)

Y. Yuan and Y. Guo: "CMCD: Count Matrix Based Code Clone Detection", in 2011 18th Asia-Pacific Software Engineering Conference. IEEE, Dec. 2011, pp. 250–257.

The CMCD technique determines the similarity between two code fragments by modelling each with a *count matrix* and comparing the count matrices. A count matrix is made up of a set of *count vectors*. In the original CMCD, there is one count vector for each variable that appears in the code fragment. The values in a count vector come from a set of *counting conditions* that are applied to the variable that vector represents. The counting conditions represent how a variable is “used”. The intuition is, if two code fragments are indeed clones, then a variable in one fragment will have a counterpart in the other fragment that is used in very similar ways, so the count vectors will be very similar. Also, most variables in one fragment will have counterparts in the other fragment, so the count matrices will be very similar. If, on the other hand, the fragments are very different, then there is a high probability that many variables in one fragment will have no obvious counterpart in the other, so the count matrices will look different.

Free detection tools (excerpt):

- Duplo is a tool to find duplicated code blocks in large C/C++/Java systems.
<http://duplo.sourceforge.net/>
- Simian (Similarity Analyzer) identifies duplication in various languages.
<http://www.harukizaemon.com/simian/>
- Clone Doctor, detects parameterized clones and “near-miss” duplicates.
<http://www.semdesigns.com/products/clone/>
- ConQAT, includes detecting duplicated code due to copy&paste programming.
<https://www.cqse.eu/en/products/conqat/overview/>
- Atomiq, finds and eliminates similar code.
<http://www.getatomiq.com/>
- ...

Consequences of code clones:

- If errors are fixed or functional changes are made, all instances of duplicated code have to be identified and modified.
- Repetitions indicate failure to use appropriate abstraction mechanisms (e.g., subroutines, functions, methods, subclasses, templates)
- Length of code increases
- Inconsistent changes lead to faults
- Redundant code hinders software maintenance and increase maintenance costs
- ...

Code duplicates are both, a threat to software quality and an opportunity for improvement.

Code Clones occur in nearly all large, long-term projects. There are different causes (copy&paste, incidentally cloning, restructuring artefacts, ...). Code clones impact software quality in a negative way: software maintenance is hindered and respective costs grow. Error handling becomes extremely difficult.

Specialized **tools for code clone detection** allow to systematically identify clone-suspect parts. Restructuring, refactoring, and abstractions can help to reduce the amount of duplicated or similar code.

Software analytics uses code-clone detection information as base information used to report on the as-is status of a software system and as key input for prediction analysis.

Code Duplication



Code Duplicates
Code Similarity
Consequences of Code Clones

Code Clone

Clones are segments of code that are identical or similar according to some definition of similarity. The notion of similarity can be based on text, lexical or syntactic structure, or semantics.

Related to: Ira Baxter, 2002; Rainer Koschke, 2007.

HPI Hasso Plattner Institut

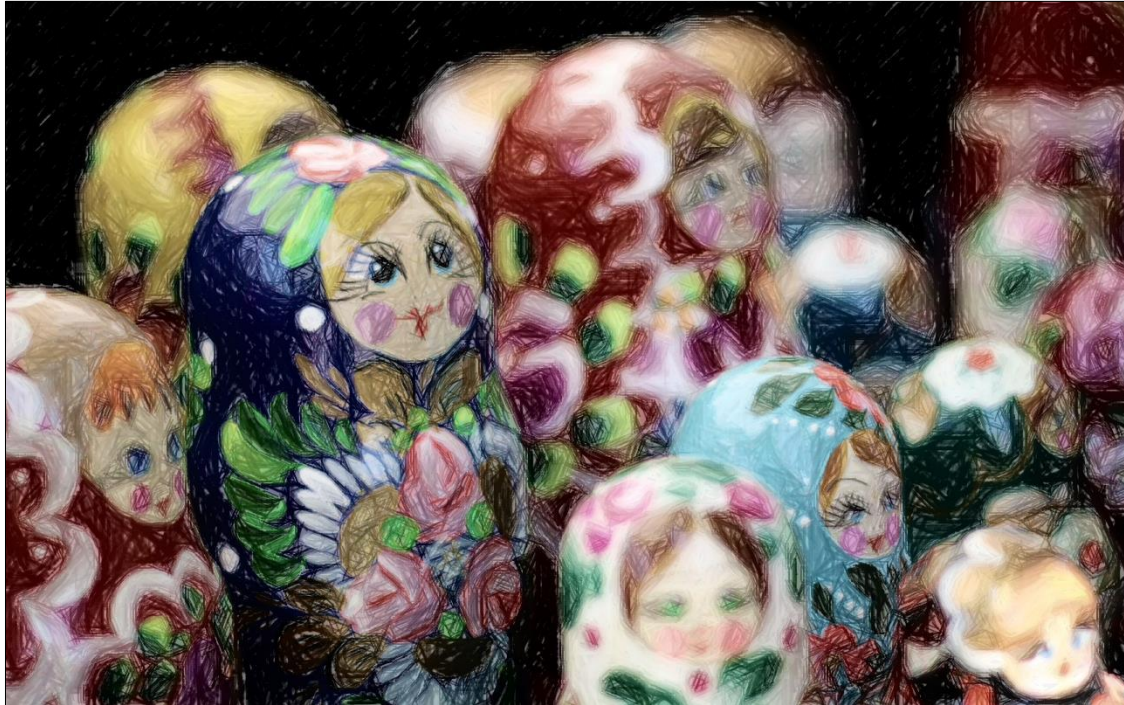


"Cloning can be a good strategy if you have the right tools in place.

Let programmers copy and adjust, and then let tools factor out the

differences with appropriate mechanisms." Ira Baxter, 2002

Nesting Level Metric



Definition of NL Metric

NL Metric Examples

NL Metric Properties

Definition

The *nesting level (NL) metric* is a software metric that quantitatively measures source code complexity by measuring the depth of enclosing of control structures one into another, usually indicated through different indentation levels within the source code.

- Source codes apply nesting in several forms including nested calls, nested levels of parentheses in arithmetic expressions, nested blocks of source code (such as nested if-clauses, while-clauses, repeat-until clauses), nested namespaces, nested classes, nested functions, etc.
- The nesting level denotes the level of containment (depth) of a logical source code block
- Used to measure the complexity of source code units such as functions or methods
- Computed by static source code analysis

Variations of NL Metrics

- **NL:** Highest nesting level of a given source code unit
- **LoopNL:** Highest nesting level in a loop of a given source code unit
- **NLMean:** Mean nesting level of multiple source code units
(e.g., whole class, module)
- **NL#+:** Number of lines-of-code above a certain nesting level, starting at #
(e.g., NL4+, NL5+)

NL Metric Examples

Example 1: Includes and declarations (e.g., header):

```
01  #pragma once                                //NL = 0
02
03  #include <string>                             //no change
04  #include <vector>                             //no change
05
06  template <typename Type>                     //no change
07      Type fromString(const std::string & string); //no change
08  template <>                                   //no change
09      char fromString<char>(const std::string & string); //no change
```

→ Resulting value for nesting level metric: $NL_{\max} = 0$

Example 2: A function with a single linear control flow, that is, no decisions nor repetitions:

```
01 void GraphicsItem::setScheme(ColorScheme * scheme)
02 {
03     m_scheme = scheme;
04
05     updateBrushes();
06     updateTooltips();
07 }
```

→ Resulting value for nesting level metric: $NL_{\max} = 1$

Example 3: A function with an if statement (with one condition), that is, with a single decision:

```
01 void Timer::restart()  
02 {  
03     const bool wasRunning(m_time->isRunning());  
04     m_time->reset();  
05  
06     if (wasRunning)  
07         m_time->start();  
08 }
```

// NL = 0
//++NL, NL = 1
//no change

//no change
//++NL, NL = 2

→ Resulting value for nesting level metric: $NL_{\max} = 2$

NL Metric Examples

Example 4: The control flow contains two for loops:

```
01  int main(int argc, char * argv[])
02  {
03      for (int i = 0; i < 10; ++i)
04          std::cout << i << std::endl;
05      for (int j = 9; j >= 0; --j)
06          std::cout << j << std::endl;
07  }
```

```
//      NL = 0
//++NL, NL = 1
//++NL, NL = 2
//--NL, NL = 1
//++NL, NL = 2
```

→ Resulting value for nesting level metric: $NL_{\max} = 2$

Example 5: The control flow contains two nested loops and an if statement:

```
01  int main(int argc, char * argv[])
02  {
03      for (int i = 0; i < 10; ++i)
04          for (int j = 0; j < 10; ++j)
05              if ((i + j) % 2 == 0)
06                  std::cout << i << ", " << j << std::endl;
07  }
```

// NL = 0
//++NL, NL = 1
//++NL, NL = 2
//++NL, NL = 3
//++NL, NL = 4

→ Resulting value for nesting level metric: $NL_{\max} = 4$

- High NL values for a source code unit generally indicate that more effort has to spend to understand that unit with respect to program understanding and comprehension.
The underlying observation is that deeply nested code becomes extremely difficult to understand from a cognitive point of view.
- NL metric measurements can be used to identify those parts of the source code that should be redesigned or refactored using modularization, abstraction, or encapsulation strategies.
- Nesting level measures beyond a threshold identify those source code units or parts that are likely to be difficult to understand, fix, or maintain.

Example for resolving nesting level issues:

```
01 void Function::provideState(const int pos)
02 {
03     if (s_maxpos < pos)
04     {
05         for (Function * function : functions())
06             function->m_states.resize(pos + 1);
07
08         s_maxpos = pos;
09     }
10 }
```

→ Resulting value for metric: $NL_{\max} = 3$

'if' condition negated,
statement block dissolved:

```
01 void Function::provideState(const int pos)
02 {
03     if (s_maxpos >= pos)
04         return;
05
06     for (Function * function : functions())
07         function->m_states.resize(pos + 1);
08
09     s_maxpos = pos;
10 }
```

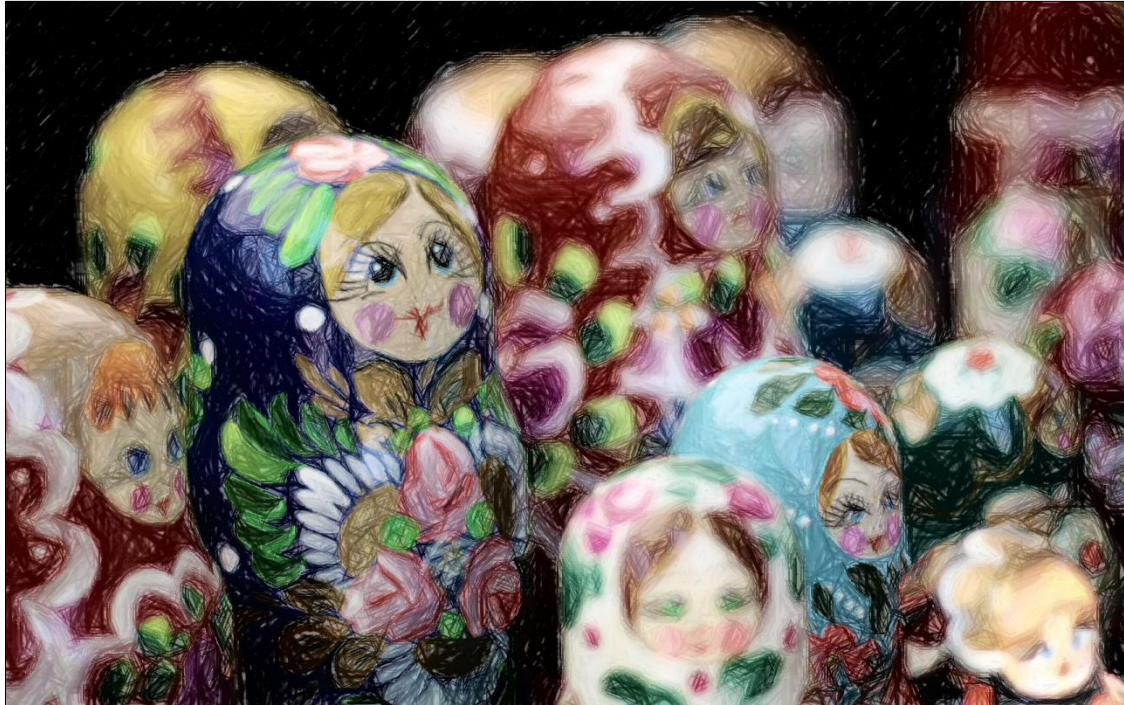
→ Resulting value for metric: $NL_{\max} = 2$

The **NL metric** represents base measurements of source code complexity. It is a simple and easy-to-compute source code metric that quantifies the depth of the logically nesting of source code in a source code unit. It is computed based on a static analysis of the source code.

A **high NL** for a source code unit generally indicates that more effort has to spend to understand that unit with respect to program understanding and comprehension. For that reason, nesting level metrics can be used to identify those parts of the source code that should be redesigned or refactored using modularization, abstraction, or encapsulation strategies.

Software analytics uses the NL metrics as base measures for source code complexity to weight, evaluate, interpret or benchmark other key performance indicators and software metrics.

Nesting Level Metric



Definition of NL Metric

NL Metric Examples

NL Metric Properties

NL

Nesting level is a quantitative software metric measuring the depth of the logically nesting of source code that reflects one aspect of source code complexity; it is computed by static source code analysis.



Questions

Tags

Users

Badges

Unanswered

Ask Question

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required.

Take the 2-minute tour

How many nested code blocks do you end up with in practice when coding?

I just finished writing a function that has ended up with nested code blocks something like the following:

4

```
class ... {  
  void Method() {  
    while (...) {  
      ...  
      switch (...) {  
        while (...) {  
          switch (...) {  
            if (...) {  
              ...  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

Do you find this is standard in your day-to-day coding work, or do you quickly set about trying to redesign and break up the code when you see something like this?

coding-style

asked 5 years ago

viewed 1803 times

active 1 year ago

Linked

21 Does anyone disagree with the statement: "using switch is bad OOP style"?

Related

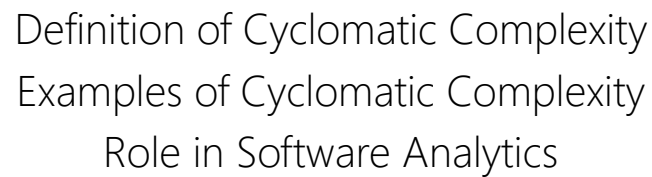
87 How many lines of code is too many?

185 How do you tell someone they're writing bad code?

173 What was the strangest coding standard rule that you were forced to follow?

NL is both an easy-to-compute and effective measure of source code complexity. It helps to identify relevant source code parts that deserve redesign.

NL also helps to identify those parts of the system that show an uncommonly high algorithmic complexity as indicated by high NL values. There may be good reasons for deep nesting but those parts should be carefully checked.



- Quantitative measure of code complexity, originally developed by McCabe (1976), also called “McCabe metric”
- Defined by the **number of linearly independent paths** of the control flow through a source code unit
- Measured for a given **code unit** such as functions, methods, classes, or modules
- **Upper bound for the number of test cases** that would be needed to fully cover all branches of the unit
- Based on static analysis of the source code

Definition of McCabe metrics:

- For a given source code, build the **control flow graph**.
- The control flow graph contains nodes (entry point, exit point, decision points);
Nodes are connected by directed edges.
- The cyclomatic complexity of a control flow graph is defined by:

$$M = E - N + 2P$$

with

- the number of edges **E**,
- number of nodes **N**,
- and number of connected components **P**.

Cyclomatic Complexity – Example 1

Example 1: A function with a single linear control flow, that is, no decisions nor repetitions:

```
01 void GraphicsItem::setScheme(ColorScheme * scheme)
02 {
03     m_scheme = scheme;
04
05     updateBrushes();
06     updateTooltips();
07 }
```

(simplified diagram)



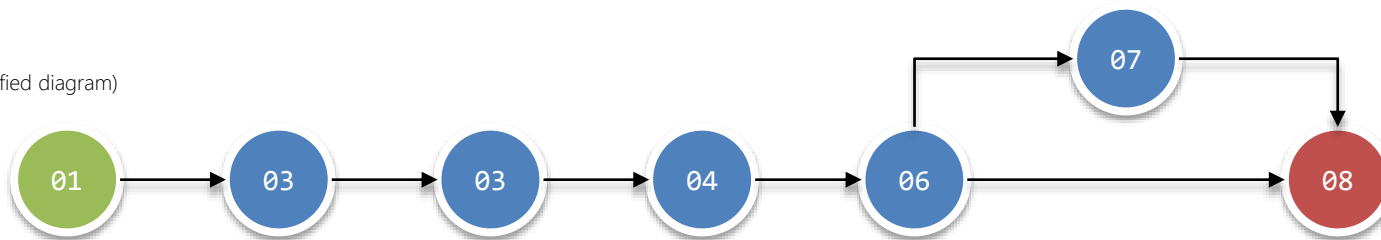
→ Resulting value for McCabe metric: $M = E - N + 2P = 4 - 5 + 2 \cdot 1 = 1$

Cyclomatic Complexity – Example 2

Example 2: A function with an if statement (with one condition), that is, with a single decision:

```
01 void Timer::restart()  
02 {  
03     const bool wasRunning(m_time->isRunning());  
04     m_time->reset();  
05  
06     if (wasRunning)  
07         m_time->start();  
08 }
```

(simplified diagram)



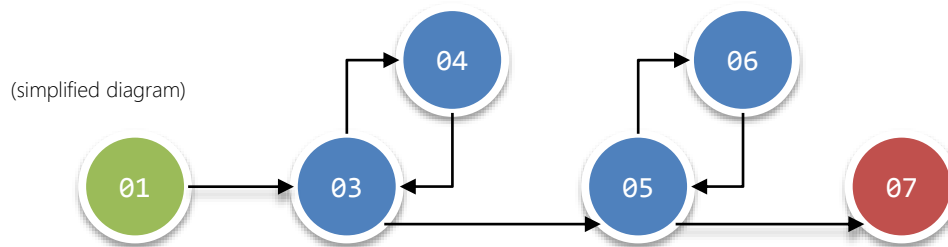
→ Resulting value for McCabe metric: $M = E - N + 2P = 7 - 7 + 2 \cdot 1 = 2$

Cyclomatic Complexity – Example 3

Example 3: Entry point of a function (green node), exit point (red node).

The control flow contains two for loops:

```
01 int main(int argc, char * argv[])
02 {
03     for (int i = 0; i < 10; ++i)
04         std::cout << i << std::endl;
05     for (int j = 9; j >= 0; --j)
06         std::cout << j << std::endl;
07 }
```



→ Resulting value for McCabe metric: $M = E - N + 2P = 7 - 6 + 2 \cdot 1 = 3$

Interpretation

If the number of linearly independent control flows exceeds a certain number, it becomes increasingly difficult to comprehend the control flow.

Experience shows that these source codes

- tend to degrade over years,
- are hard to test, maintain, or troubleshoot,
- indicate how likely the code will be to produce errors.

Application of McCabe complexity measure

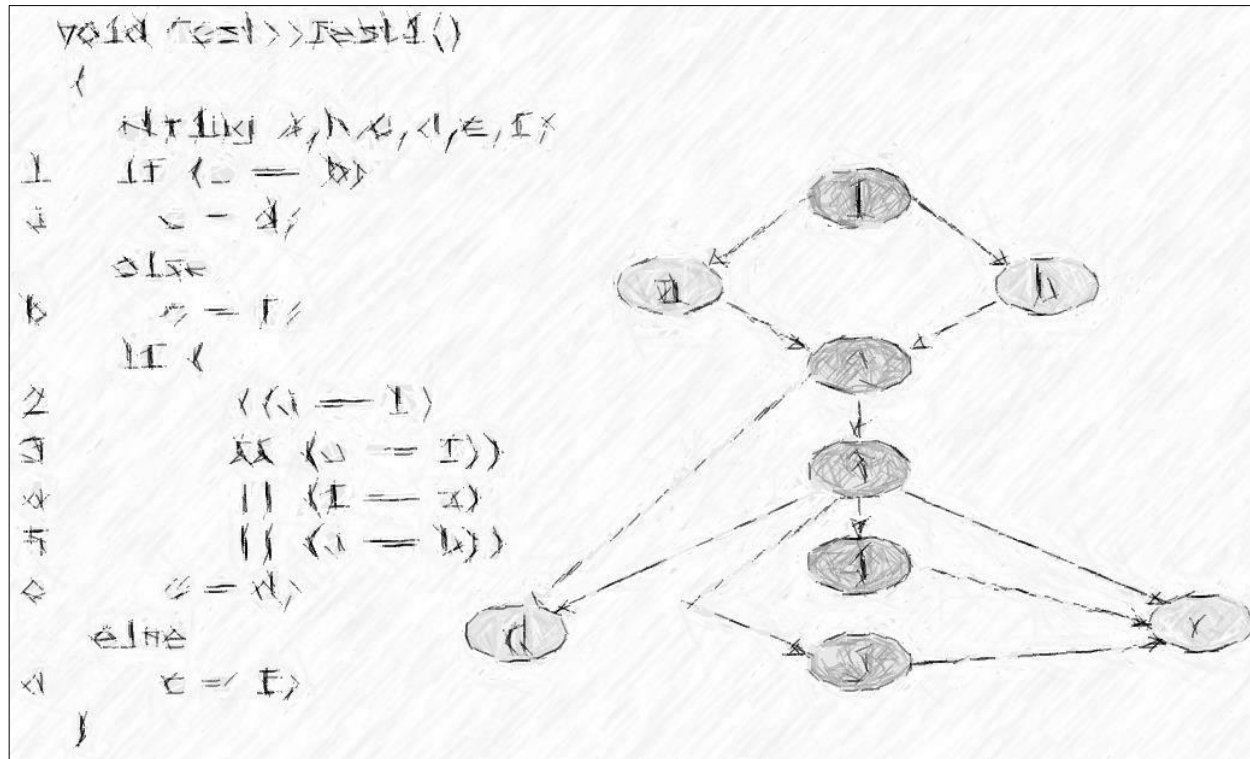
- Limit the complexity during program development to an agreed maximum (e.g., 10)
- Identify parts of the source code with high complexity for redesign or refactoring

The cyclomatic complexity represents a **fundamental complexity metrics** for source code. It is based on a **static analysis of the source code** by counting the number of linearly independent paths of the control flow.

The cyclomatic complexity represents a base measure for evaluating the **maintenance efforts** of a given source code. The higher the measure, the higher the costs for understanding and comprehending the source code; the higher the measure, the higher are the risks that source codes is faced by deficiencies.

Software analytics uses cyclomatic complexity as a kind of **“key performance indicator”** that evaluates and benchmarks a given complex computer-based system.

Cyclomatic Complexity



Definition of Cyclomatic Complexity

Examples of Cyclomatic Complexity

Role in Software Analytics

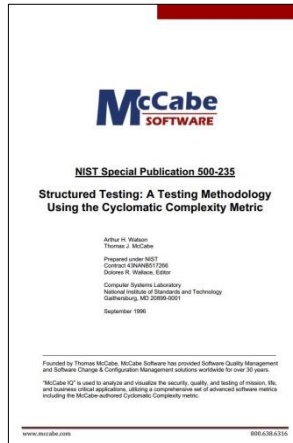
Control Flow Graph

A directed graph, whose nodes correspond to entry, exit, or decision points of a given unit of source code, and whose directed edges denote the flow of control. It is an abstraction of the behavior of the given unit.

Cyclomatic Complexity (McCabe Metric)

A quantitative quality metric, whose input are code units and whose outputs are the number of linearly independent paths of the control flow.

Further Reading



Arthur H. Watson, Thomas J. McCabe: "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", NIST Special Publication, 500-235, 1996

Retrieved May 14, 2011, available from McCabe Software:
<http://www.mccabe.com/pdf/mccabe-nist235r.pdf>

Cyclomatic complexity for a whole system provides a general overview of how extensive software maintenance tends to be.

Cyclomatic complexity offers a good measure for identifying those parts of a complex software system that need to be refactored and redesigned.

As a base measure, it can be used as "weight" for other metrics.

Definition of Halstead Metrics

Example Calculations

Definition (M. Halstead, 1977)

The *Halstead metrics* is a set of software quality metrics that quantitatively measure source code complexity. The *Halstead Length* computes a measure for the number of *distinct* operators and operands used in a source code unit; the *Halstead Volume* computes a measure of the total number of *used* operators and operands in a source code unit.

- Halstead software metrics are based on static source code analysis that concentrates on the **lexicographically complexity** of source code.
- Operators and operands are defined by their relationship to each other – in general, an **operator** carries out an action and an **operand** participates in such an action.
- The underlying assumption of Halstead is that programming requires cognitive efforts (**elementary cognitive steps**) that can be quantitatively related to operators and operands.

Base Measurements

- The number of distinct operators n_1 and operands n_2 build the program vocabulary n .
- The total number of used operators N_1 and operands N_2 build the program length N .

- **Halstead Length** $H_L = n_1 \log_2(n_1) + n_2 \log_2(n_2)$

- **Halstead Volume** $H_V = N \log_2(n)$

$$(N = N_1 + N_2, n = n_1 + n_2)$$

Definition of Halstead Metric

```
01 int main(int argc, char** argv) {  
02     if(argc!=2) {  
03         printf("ERROR, Program usage: helloworld [ARGUMENT]");  
04         return -1;  
05     }  
06     else {  
07         printf("Your argument: %s", argv[1]);  
08         return 0;  
09     }  
10 }
```

- Distinct operators: $n_1 = 10$
- Total number: $N_1 = 18$

Operator Freq.	
*	2
!=	1
if()	1
else	1
[]	1
{}	3
;	4
return	2
,	2
-	1

Definition of Halstead Metric

```
01 int main(int argc, char** argv) {  
02     if(argc!=2) {  
03         printf("ERROR, Program usage: helloworld [ARGUMENT]");  
04         return -1;  
05     }  
06     else {  
07         printf("Your argument: %s", argv[1]);  
08         return 0;  
09     }  
10 }
```

- Distinct operands: $n_2 = 11$
- Total number: $N_2 = 16$

Operand Freq.	
int	2
main	1
argc	2
argv	2
char	1
printf	2
2	1
0	1
1	2
"ERROR ...]"	1
"Your ... %s"	1

Program Difficulty Level

- Calculation: $D = (n_1/2)(N_2/n_2)$
- The program difficulty level or error proneness of a source code is proportional to the number of distinct operators (n_1) and is also proportional to the ratio between the total number of used operands (N_2) and the number of distinct operands (n_2).
- A low level of program difficulty means that the program is less prone to errors.

Effort to Implement

- Calculation: $E = DH_V$
- The effort to implement (E) or understand a program is proportional to the difficulty level (D) and volume (H_V).

Time to Implement

- Calculation: $T = E/18$ sec
- The time to implement or understand a program (T) is proportional to the effort.
- Empirical experiments can be used for calibrating this quantity.
- Halstead has found that dividing the effort by 18 give an approximation for the time in seconds for the projects he investigated. (18 is frequently used time-for-effort factor.)
- The time-for-effort factor depends, for example, on programming language, program type, and programming conventions.

Bugs Delivered

- Calculation: $B = E^{2/3}/3000$
- Halstead's delivered bugs (B) is an estimate for the number of errors in a source code.
- The number of delivered bugs approximates the number of errors in a module. When dynamic testing is concerned, the most important Halstead metric is the number of delivered bugs. As a goal at least that many errors should be found from the module in its testing.

Halstead Example - Calculation

```
01 int main(int argc, char** argv) {  
02     if(argc!=2) {  
03         printf("ERROR, Program usage: helloworld [ARGUMENT]");  
04         return -1;  
05     }  
06     else {  
07         printf("Your argument: %s", argv[1]);  
08         return 0;  
09     }  
10 }
```

- Halstead length: $H_L = 10 \log_2(10) + 11 \log_2(11) \approx 71.27$
- Halstead volume: $H_V = 34 \log_2(21) \approx 149.34$
- Program difficulty: $D = (10/2) \cdot (16/11) = 7.\overline{27}$
- Effort: $E = 7.27 \cdot 149.34 \approx 1086.10$
- Time: $T = 1086.10/18 \approx 60.34$
- Bugs: $B = E^{2/3}/3000 \approx 0.035$

Halstead Example - Interpretation

```
01 int main(int argc, char** argv) {
02     if(argc!=2) {
03         printf("ERROR, Program usage: helloworld [ARGUMENT]");
04         return -1;
05     }
06     else {
07         printf("Your argument: %s", argv[1]);
08         return 0;
09     }
10 }
```

- Program difficulty $D=7.27$ is the basic complexity measurement that indicates how difficult it is to understand or write this particular code segment.
- The Effort $E \approx 1086.10$ can be translated into actual coding time: $T = E/18 \approx 60.34$ (seconds). The commonly used constant “18” has been determined by a number of empirical studies over the years.
- The measurement for bugs B is an estimate for the minimum number of errors in the implementation that should be found with automated tests.

The **Halstead metrics** is a set of software quality metrics that quantitatively measure code complexity. Derived measures include **program difficulty level, effort to implement, time to implement, and delivered bugs.**

Halstead metrics are easy to compute and can be applied to any programming language. They are basic measures of source code complexity. Its intrinsic value is Halstead metrics are related to the **cognitive effort** of developers when they write, read, and modify source code. However, Halstead metrics do not explicitly consider modern programming language constructs such as classes and namespaces.

Software analytics uses the Halstead metrics as base measures for source code complexity.

Halstead Metrics

```
def halsteds_metrics(program, test_cases):  
  
    n = 0  
    for token in program.tokenize():  
        n += 1  
    n_test_cases = len(test_cases)  
  
    print("Example: Halsteads Metrics for program <program>  
          <expression>")  
  
    print  
  
    result = halsteds_metrics(program, test_cases)  
    print("The program's Halstead Metrics: <n>, <N>")  
    print
```

Definition of Halstead Metrics
Example Calculations

Halstead Metrics

The *Halstead metrics* are a set of software quality metrics that measure source code complexity by lexicographic analysis in terms of operands and operators used by an implementation. Derived measures serve to estimate program difficulty level, effort to implement, time to implement, and delivered bugs.

Further Reading



Yu, Sheng and Zhou, Shijie: "A Survey on Metric of Software Complexity", Information Management and Engineering (ICIME), 2010, IEEE.

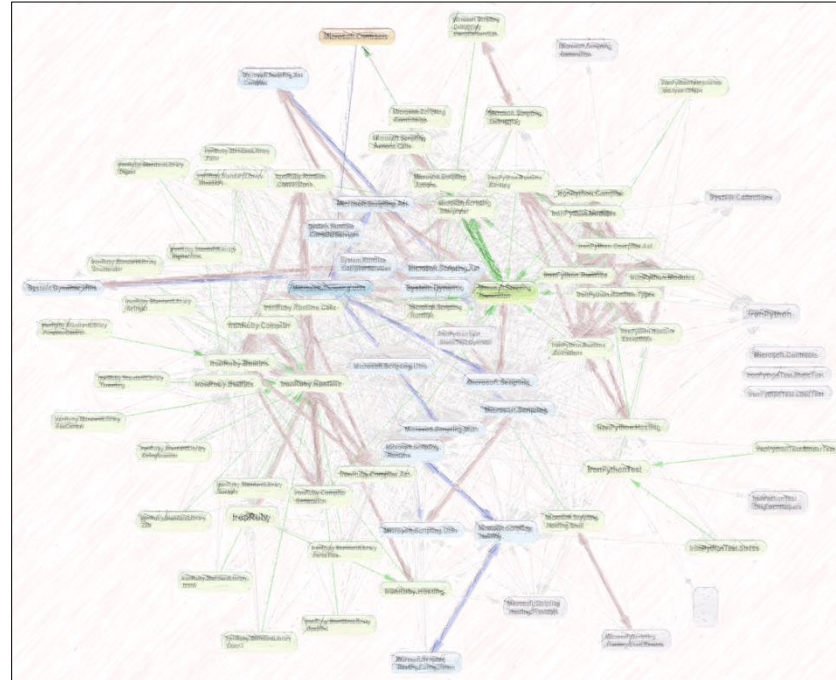
There is no sharp definition of “operators” and “operands”, and their definition depends on the programming language used.

The code analysis function in MS Visual Studio computes a Maintainability Index, which also takes into account the Halstead Volume as one factor.

Halstead metrics apply if test and maintenance efforts are to be estimated and controlled.

There are widely differing opinions on the worth of Halstead measures, ranging from "convoluted... [and] unreliable" [Jones 94] to "among the strongest measures of maintainability" [Oman 91].

Module Dependencies

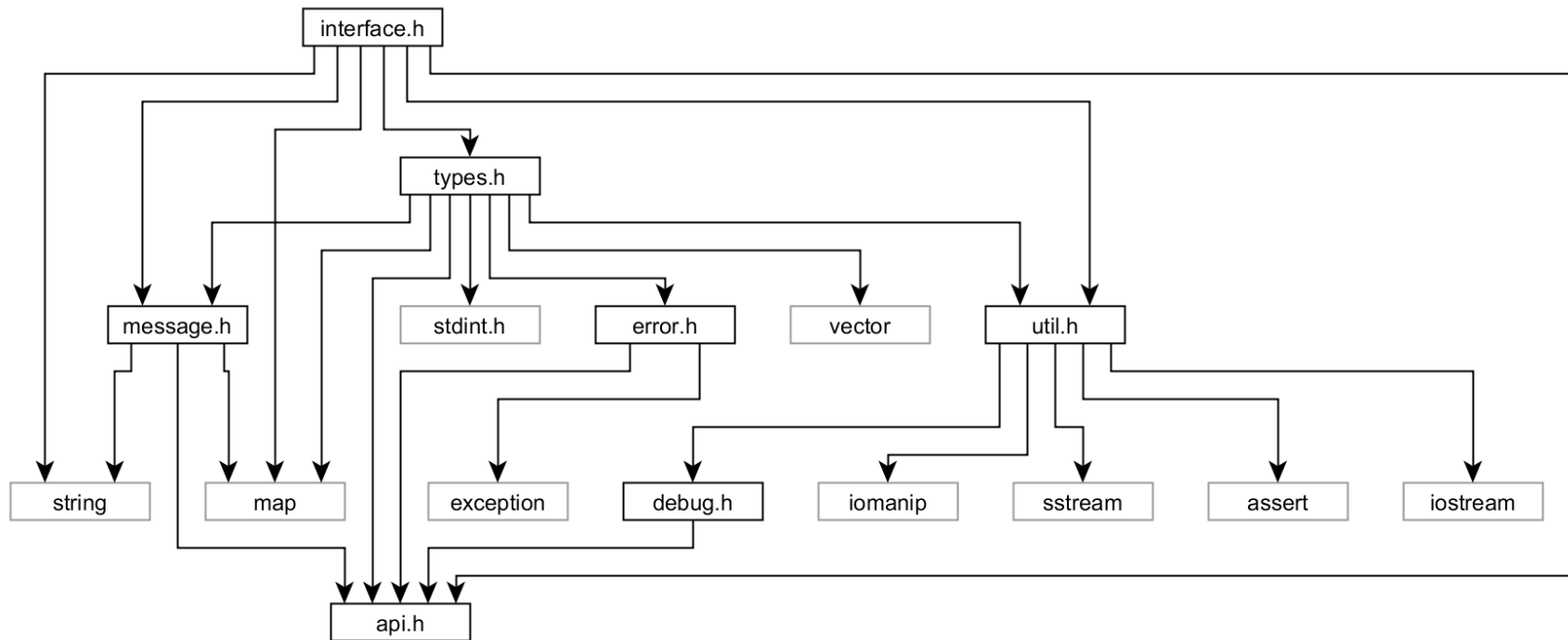


Principles of Module Dependencies
Coupling
Cohesion

- **Modularization** is a key construction principle for complex software systems.
 - Module types include subsystems, components, packages, libraries, classes, or namespaces.
 - Modules are organized in hierarchies
- **Dependency** results from usage relations (modules include interfaces of other modules), inheritance relations (modules are specialized versions of other modules, e.g., subclasses), or from any other type of logical dependency or dependency at run-time.
- **Dependency network graphs** model dependencies among a collection of modules. Nodes of the graph represent modules; directed edges represent dependencies.

Example of a dependency graph for a collection of C++ files:

Dependencies are shown for the file "interface.h". The outgoing edges of "interface.h" denote the direct include dependencies. Additionally, indirect include dependencies are shown.



Definition

- A *dependency* denotes a formal relationship between two modules. Dependencies can be detected by static source code analysis.
- Module *A* *depends* on module *B*, for example, if
 - *A* calls *B* (call dependency)
 - *A* inherits from *B* (inheritance dependency)
 - *A* uses a variable of type *B* (data/data type dependency)
- A set of modules $A_1, A_2, \dots$ is called a *clique* if for each pair (A_i, A_j) a dependency exists (neglecting the direction). Modules of a common clique are considered to be closer to each other (*closeness*).

Internal and External Dependency

- **Coupling** of a module refers to external dependencies to other modules, that is, the degree of *inter-module dependency*.
- **Cohesion** of a module refers to internal dependencies within the same module, that is, the degree of *intra-module dependency*.

Coupling Types

- *Efferent coupling*: the number of ("outgoing") dependencies to other modules of a given module, that is, a measure of how dependent the given module is.
- *Afferent coupling*: the number of modules that depend on the given module ("ingoing"), that is, a measure of the responsibility that module has.

Undesired

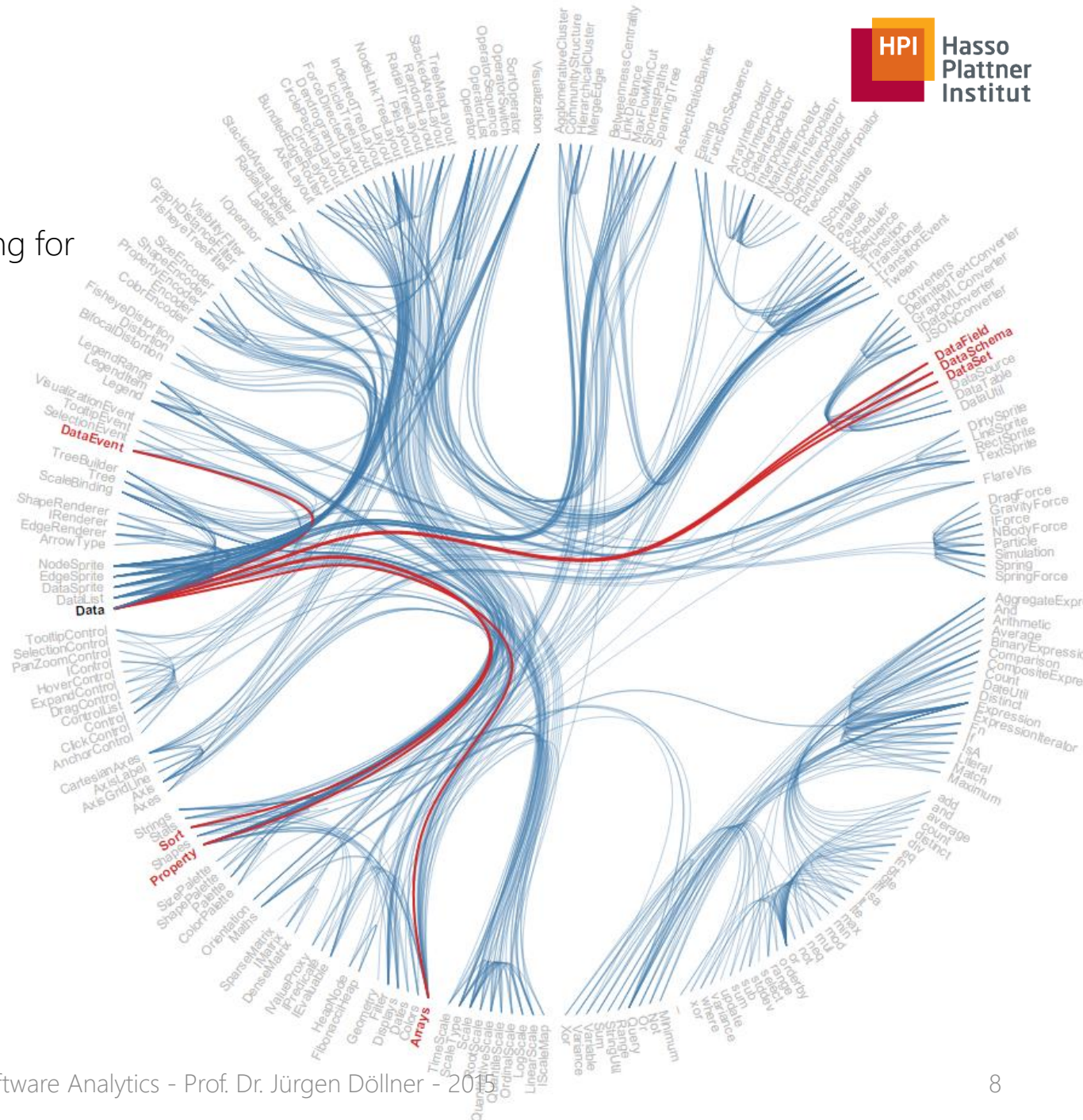


Desired

- **Content**: one module knows internals of another module
- **Global**: two modules share the same global data
- **External**: two modules share an external interfaces
- **Control**: one module controlling the flow of another
- **Data-structured**: modules share only parts of related data
- **Data**: modules share data
- **Message coupling**: communication via parameters or message passing
- **No coupling**: no communication between modules

Coupling

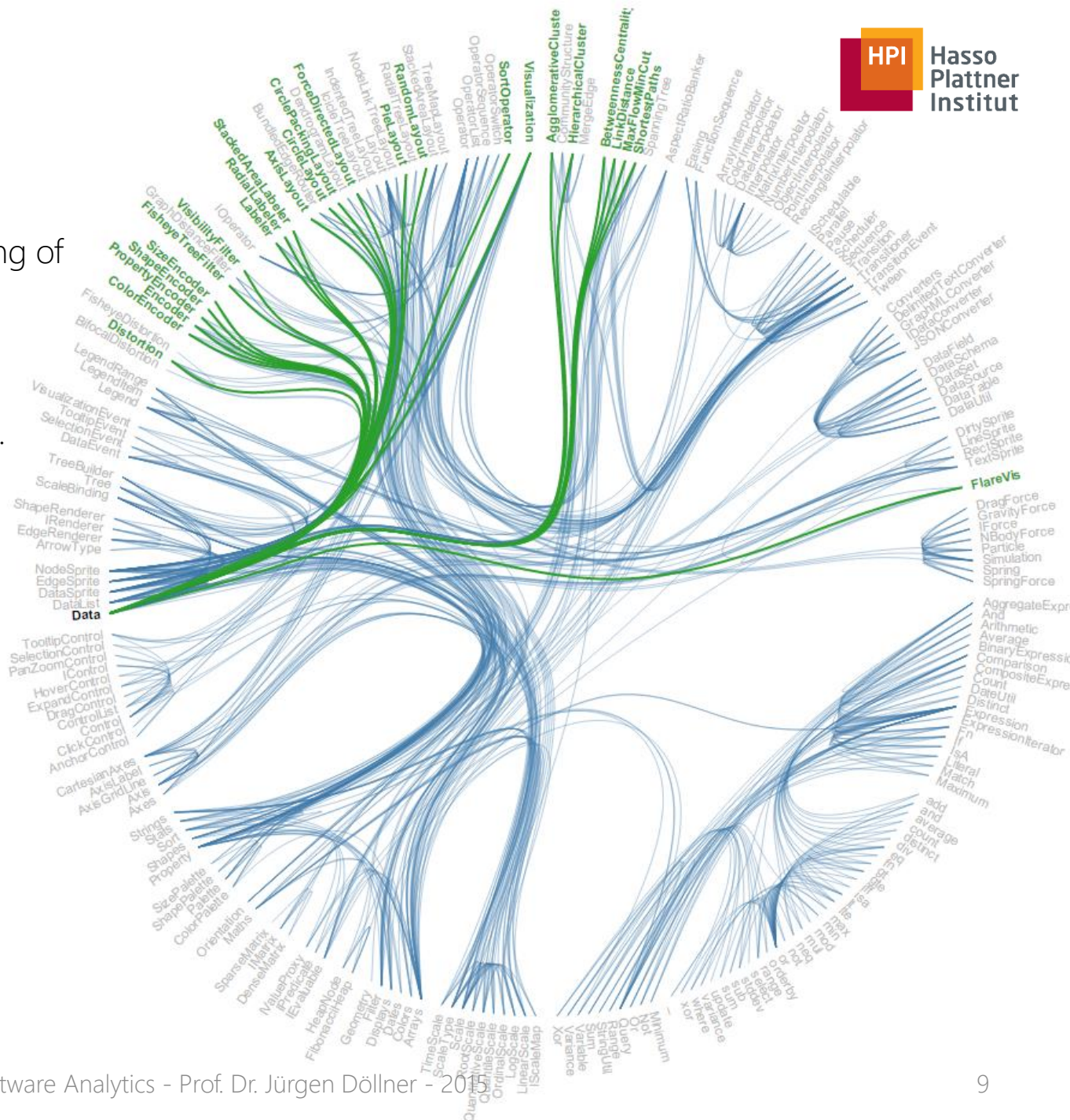
Example: Efferent coupling for
the module "Data"
(red: all modules that
"Data" imports/includes)



Coupling

Example: Afferent coupling of module "Data".

All modules that use,
import, or include "Data".



Coupling Metric:

$$C = 1 - \frac{k}{d_i + a c_i + d_o + b c_o + g_d + c g_c + w + r} \text{ with } a, b, c = 2; k = 1 \text{ for approximation.}$$

– Roger S. Pressman: "Software Engineering: A Practitioner's Approach", 2009

For data and control flow coupling:

- d_i number of input data parameters
- c_i number of input control parameters
- d_o number of output data parameters
- c_o number of output control parameters

For global coupling:

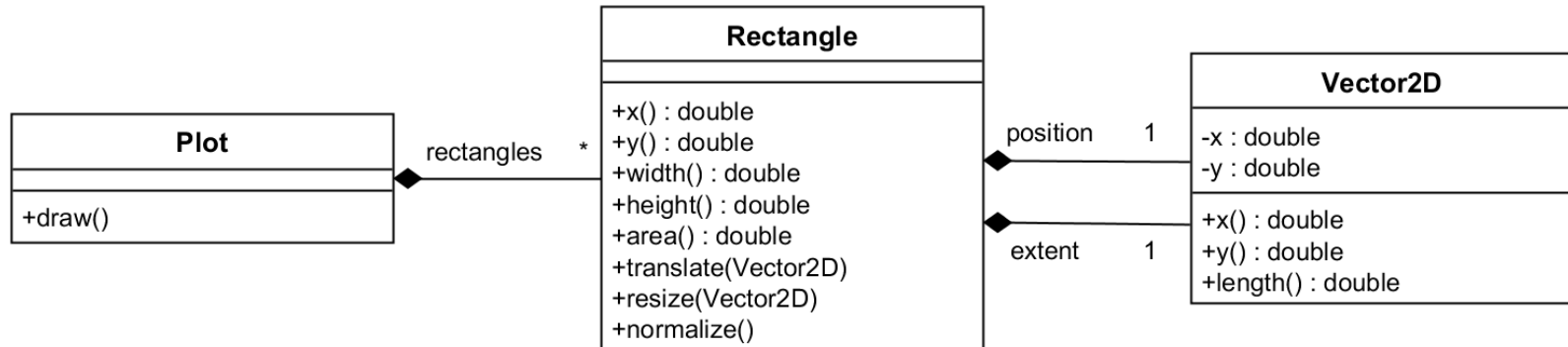
- g_d number of global variables used as data
- g_c number of global variables used as control

For environmental coupling:

- w number of modules called (fan-out)
- r number of modules calling the module under consideration (fan-in)

$C \rightarrow 0$: weak coupling, $C \rightarrow 1$: strong coupling

– Shyam .R. Chidamber, Chris. F. Kemerer: "Towards a metrics suite for object-oriented design", 1991



Metrics for Rectangle Values

Number of input parameters d_i 2 (Vector2D in translate and resize)

Fan-out w 1 (to Vector2D)

Fan-in r 1 (from Plot)

Coupling C 0.75 $= 1 - 1 / (2 + 1 + 1)$

- *Efferent cohesion*: the number of dependencies within a module
- *Afferent cohesion*: the number of responsibilities within a module

Undesired



Desired

- **Coincidental**: arbitrarily grouped
- **Logical**: different nature but similar category
- **Temporal**: grouping by execution time similarity
- **Procedural**: grouping by execution order
- **Communicational**: operate on the same data
- **Sequential**: shared inputs and outputs/pipeline
- **Functional**: contribution to a single, well defined task

Indications

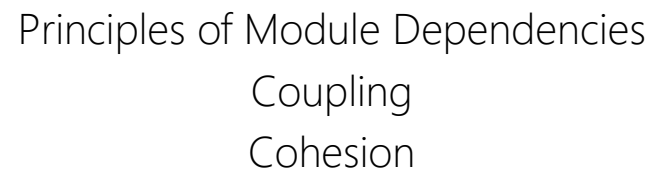
- A **high degree of efferent coupling** can indicate that a module is unfocused. It may also indicate brittleness since the module depends on the stability of many coupled modules.
- A **high degree of coupling** can indicate higher costs for reuse and maintenance of the module.
- A **low degree of cohesion** can indicate that some module parts are not related to each other. Possibly these unrelated parts should be refactored, redistributed, or redesigned.
- A **high degree of cohesion** can indicate that the module's parts are interrelated and, hence, form a "strong" module (e.g., tight bundling of attributes and methods in a class).

The implementation of a complex computer-based system is generally characterized by modularization. **The dependency network graph** reveals the dependencies among modules in a formal way. Typically, dependencies are determined by static code analysis.

Inter-module and intra-module dependency measures reflect to different aspects

- coupling indicates the quality of modularization, whereas cohesion gives indications about the composition of a module.

In software analytics, both aspects support **program understanding and comprehension**. Dependency network graphs can be used for prediction analytics, for example, to assess the likelihood for bug propagation, efforts for redesign or refactoring, etc.



Dependency

A dependency denotes a relationship between two modules inferred from data dependencies, call dependencies, or inheritance dependencies.

Coupling

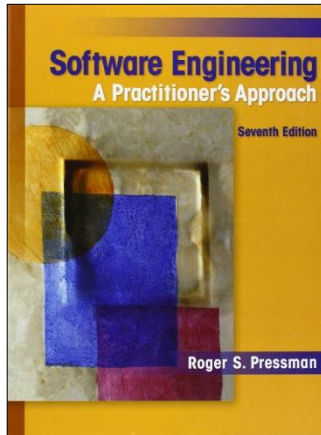
"The manner and degree of interdependence between software modules."

Cohesion

"The manner and degree to which the tasks performed by a single software module are related to one another."

Sources: IEEE Standard Glossary of Software Engineering Terminology

Further Reading



Roger S. Pressman:

"Software Engineering: A Practitioner's Approach", 2009



Shyam .R. Chidamber, Chris. F. Kemerer:

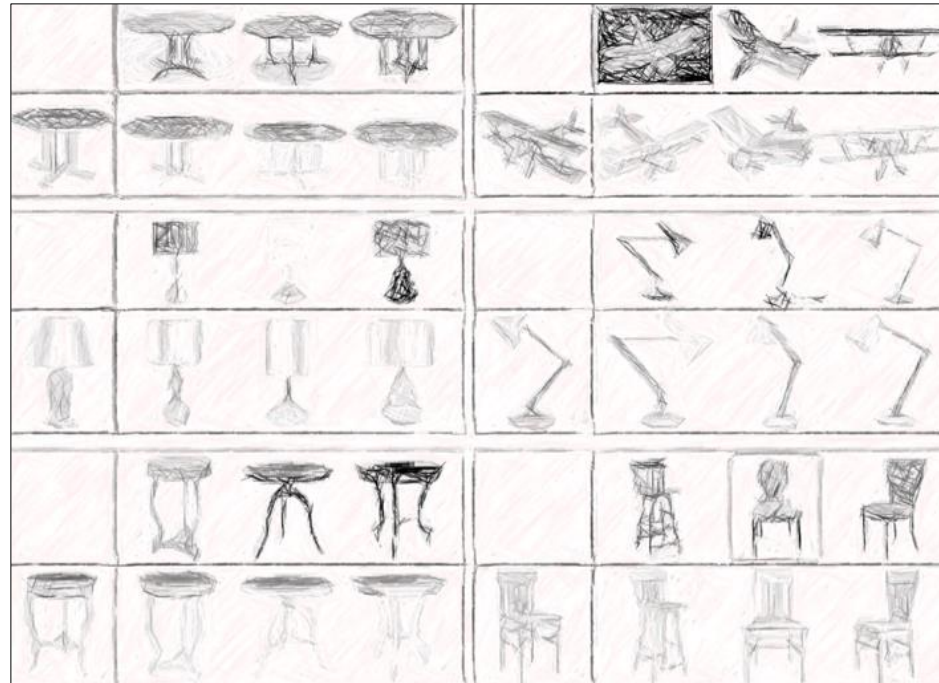
"Towards a metrics suite for object-oriented design", 1991

The actual dependencies may be more complex compared to those detected by static source code analysis. For example, pure virtual methods or calls by function pointers are resolved at run-time, that is, they are not detectable by static analysis. To detect those dependencies, run-time behavior has to be analyzed (e.g., trace analysis, function usage).

Unexpected cliques of modules often indicate and suggest abstraction, for example, to build a new package, class, or library.

Dependencies to standard modules (system libraries etc.) frequently can be excluded for analysis because these modules can be considered to be part of the programming environment. Dependency metrics reflect source code complexity that is induced by the use of other modules (calls, data, inheritance). For that reason, it is a key indicator for redesign, refactoring, repackaging and provides base information for software analytics.

Object-Oriented Metrics



Definition of Object-Oriented Metrics

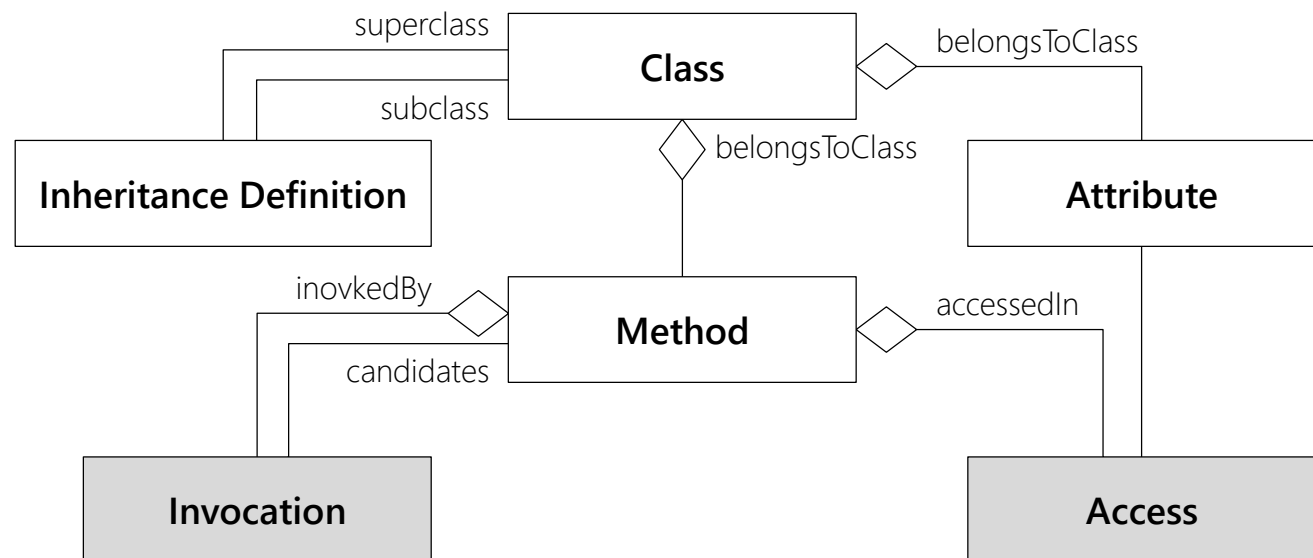
CK Metrics

MOOD and QMOOD

- **Object-oriented (OO) analysis and design** are widely adopted in software development; the implementation of object-oriented systems is supported by a number of
 - OO programming languages (C#, C++, Java, ...) and
 - OO modeling languages (e.g., UML).
- OO design principles include **encapsulation, information hiding, and inheritance** that aim at better management of system complexity and evolution
- OO development, in general, offers advantages regarding software maintainability, reusability, and reengineering, etc.
- OO development requires additional **OO specific metrics** compared to traditional functional decomposition and data flow based approaches, reflecting quality aspects of the object-oriented design.

Object-oriented Software Metric

- An *object-oriented software metric* is a software metric that reflects specific aspects of the object-oriented design.
- Meta model for main OO concepts:



– M. Lanza, S. Ducasse: "Beyond Language Independent ObjectOriented Metrics: Model Independent Metrics"
In: 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering. (2002)

Objectives

- OO metrics facilitate to “monitor” OO principles in design and to support stakeholders to achieve better, consistent OO design.
- OO metrics can be used to predict efforts related to fault-proneness, reliability, maintenance, and testing of OO systems.
- OO metrics evaluate methods, attributes, classes, coupling, and inheritance.

CK Metrics (Chidamber and Kemerer, also: MOOSE “Metrics for OO SW Engineering”)

- Metrics:
 - Weighted Methods per Class (WMC)
 - Depth of Inheritance Tree (DIT)
 - Number of Children (NOC)
 - Coupling Between Objects (CBO)
 - Response for a Class (RFC)
 - Lack of Cohesion in Methods (LCOM)
- CK metrics measure design complexity in relation to impact on external quality attributes such as maintainability, reusability, theoretically and empirically validated.

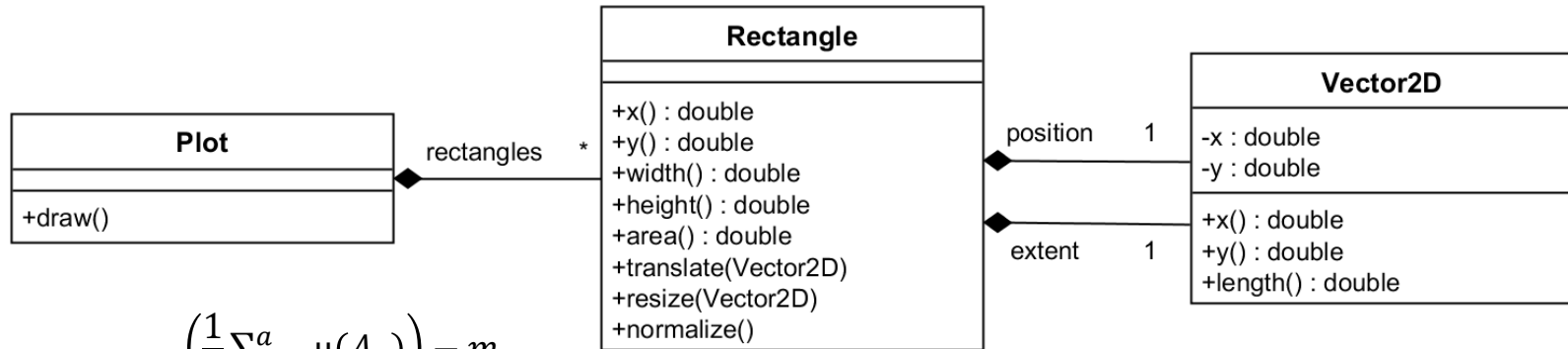
Lack of Cohesion in Methods

- $M = \{M_1, M_2, M_3, \dots, M_m\}$ Set of methods of a class
- $A = \{A_1, A_2, A_3, \dots, A_a\}$ Set of attributes of a class
- $\mu(A_k)$ results the number of methods in M accessing A_k

$$LCOM^* = \frac{\left(\frac{1}{a} \sum_{k=1}^a \mu(A_k)\right) - m}{1 - m}$$

- Assumption that $\mu(A_k) \geq 1$
- Properties:
 - $LCOM^* = 0$, desired situation (...all methods accessing every attribute)
 - $LCOM^* = 1$, undesired situation (...only one accessing method per attribute)

– Lionel C. Briand, John W. Daly, Jürgen Wüst: „A Unified Framework for Cohesion Measurement in Object-Oriented Systems“, 1998



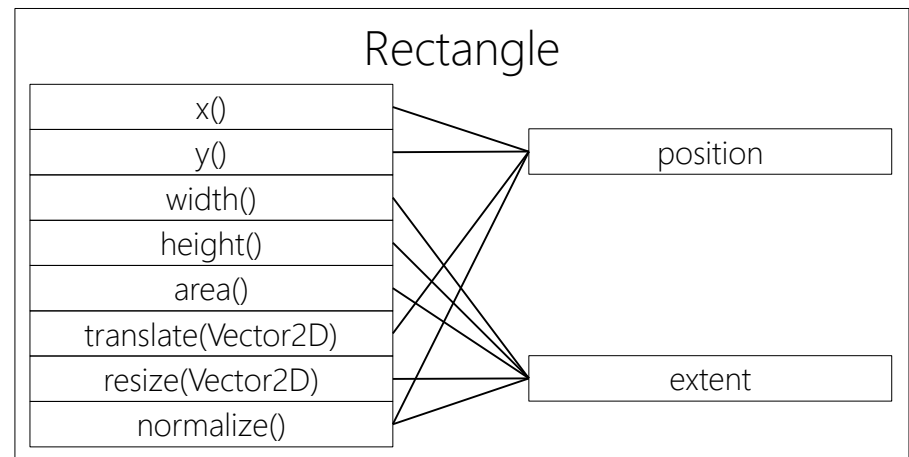
$$LCOM^* = \frac{\left(\frac{1}{a} \sum_{k=1}^a \mu(A_k)\right) - m}{1 - m}$$

$$LCOM^* = \frac{\left(\frac{1}{2} \sum_{k=1}^2 \mu(A_k)\right) - 8}{1 - 8}$$

$$LCOM^* = \frac{\left(\frac{1}{2} (4 + 5)\right) - 8}{1 - 8}$$

$$LCOM^* = \frac{4.5 - 8}{1 - 8}$$

$$LCOM^* = 0.5$$



Applications of CK Metrics for Software Analytics Objectives

- CK metrics “appear to be useful to predict class fault proneness during the early phases of the life cycle.”

– Basili, Briand, Melo: “A Validation Of Object-Oriented Design Metrics As Quality Indicators”. IEEE Transactions on Software Engineering. 1995

- “higher values of the coupling and cohesion [CK] metrics ... were associated with reduced productivity and increased rework/design effort.”

– S.R. Chidamber, D.D., Kemerer, C.: Managerial use of metrics for object oriented software: An exploratory analysis. IEEE Trans. Software Eng. 24 (1998) 629–639

- In a study of “a medium-size telecommunication system, ... [DIT and NOC] were found to influence defect density.”

– Cartwright, M., Shepperd, M.: An empirical investigation of an object-oriented software system. IEEE Trans. Software Eng. 26(7) (2000) 786–796

MOOD (Metrics for OO Design; Abreu 1995)

- MOOD metrics are system-wide measurements
- Each metric is a quotient of the use of a particular mechanism in the system being measured (numerator) and the maximum possible use of the same mechanism (denominator). (Reminder: **numerator/denominator**)
- Metrics:
 - Method Hiding Factor (MHF)
 - Attribute Hiding Factor (AHF)
 - Method Inheritance Factor (MIF)
 - Attribute Inheritance Factor (AIF)
 - Polymorphism Factor (PF)
 - Coupling Factor (CF)

MOOD metrics are expressed as percentages, ranging from 0% (no use) to 100% (maximum use) and are dimensionless.

- If all methods are private, Method Hiding Factor (MHF) is 100%.
If all methods are public, Method Hiding Factor (MHF) is 0%.
- If all attributes are private, Attribute Hiding Factor (AHF) is 100%.
If all methods or attributes are public, Attribute Hiding Factor (AHF) is 0%.
- If there is no reusability, Method Inheritance Factor (MIF) is 0%.
- If there is no reusability, Attribute Inheritance Factor (AIF) is 0%.
- If all methods are overridden in all derived classes, Polymorphism Factor (PF) is 100%.
- If no classes are coupled, Coupling Factor (CF) is 0%.

Example: comparison of MOOD measurements for complex software systems:

System	MFC	GNU	ET+	Motif
Method Hiding Factor (MHF)	24.6%	13.3%	9.6%	39.2%
Attribute Hiding Factor (AHF)	68.4%	84.1%	69.4%	100.0%
Method Inheritance Factor (MIF)	83.2%	63.1%	83.9%	64.3%
Attribute Inheritance Factor (AIF)	59.6%	62.6%	51.8%	50.4%
Polymorphism Factor (PF)	2.7%	3.5%	4.5%	9.8%
Coupling Factor (CF)	9.0%	2.8%	7.7%	7.6%

– originally taken from dec.bournemouth.ac.uk/staff/swebster/OODesignEvaluation/sld014.htm

Applications of QMOOD

Quality model for object oriented design metrics:

Design Property		Derived Design Metric	
Design Size	DSC	Design Size in Classes	
Hierarchies	NOH	Number of Hierarchies	
Abstraction	ANA	Average Number of Ancestors	
Encapsulation	DAM	Data Access Metric	
Coupling	DCC	Direct Class Coupling	
Cohesion	CAM	Cohesion Among Methods in Class	
Composition	MOA	Measure of Aggregation	
Inheritance	MFA	Measure of Functional Abstraction	
Polymorphism	NOP	Number of Polymorphic Methods	
Messaging	CIS	Class Interface Size	
Complexity	NOM	Number of Methods	

– Bansiya J. and C. G. Davis (2002): A Hierarchical Model for Object-Oriented Design Quality Assessment, IEEE Transactions on Software Engineering, pp. 4-17, 2002.

Applications of QMOOD

Define relations between quality attributes and design properties by equations:

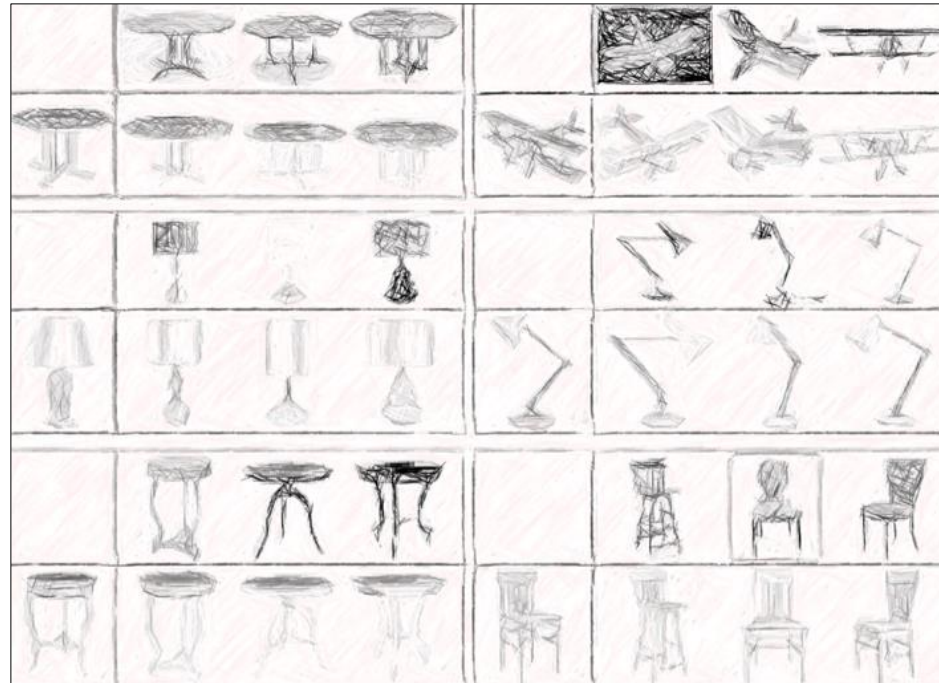
Quality Attribute	Index Computation Equation
Reusability	- 1/4 Coupling + 1/4 Cohesion + 1/2 Messaging + 1/2 Design Size
Flexibility	- 1/4 Coupling + 1/4 Encapsulation + 1/2 Composition + 1/2 Polymorphism
Understandability	- 1/3 Abstraction + 1/3 Encapsulation - 1/3 Coupling + 1/3 Cohesion - 1/3 Polymorphism - 1/3 Complexity - 1/3 Design Size
Functionality	+ 6/50 Cohesion + 11/50 Polymorphism + 11/50 Messaging + 11/50 Design Size + 11/50 Hierarchies
Extensibility	+ 1/2 Abstraction - 1/2 Coupling + 1/2 Inheritance + 1/2 Polymorphism
Effectiveness	+ 1/5 Abstraction + 1/5 Encapsulation + 1/5 Composition + 1/5 Inheritance + 1/5 Polymorphism

Object-oriented metrics allow for monitoring key characteristics of object-oriented design in a software system's implementation. OO metrics are based on a static analysis of source code, analyzing class features within a given system, and they are automatically computed.

CK metrics and **MOOD/QMOOD** represent two approaches for OO metrics; both are used in practice. In a number of studies, it is shown that these metrics generally facilitate control and monitoring of software quality related to the OO design.

OO metrics represent a key source of information about the OO aspects of a complex, long-term software system. **Software analytics** measures and monitors OO metrics to provide key performance indicators regarding reusability, flexibility, understandability, functionality, extendibility, effectiveness of OO systems.

Object-Oriented Metrics



Definition of Object-Oriented Metrics

CK Metrics

MOOD and QMOOD

OO Metric

An *object-oriented metric* is a software metric that reflects object-oriented design characteristics of object-oriented systems.

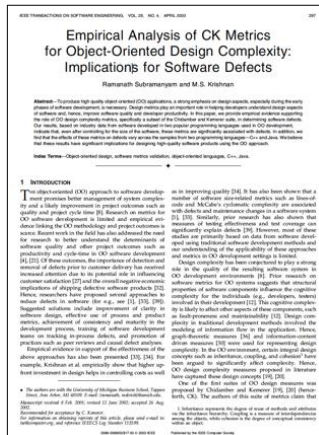
CK Metrics

CK metrics are a set of OO metrics capturing concepts of inheritance, coupling and cohesion in object-oriented systems.

QMOOD

Quality model for object-oriented design metrics are a set of OO metrics capturing concepts of inheritance, coupling and cohesion in object-oriented systems.

Further Reading



R. Subramanyam, M. S. Krishnan: "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects". IEEE Transactions on Software Engineering, 29(4), April, 2003.



S. Chawla: "Review of MOOD and QMOOD metric sets". International Journal of Advanced Research in Computer Science and Software Engineering, 2013, www.ijarcscse.com

If object-orientation is used in design and implementation of a software system, OO metrics should be monitored over time as a key software quality check.

OO metrics can be compared, on the one hand, among subsystems and components within the one project; on the other hand, OO metrics of other projects (e.g., similar in type or size) could be used for comparison.

Software analytics uses OO metrics, monitored over time, to identify correlations between different OO characteristics and different software quality changes or software development activities.