Lectures 17-18: Estimation Techniques and Software Metrics

Today

- We present software metrics
  - How they are calculated
  - What they represent
  - Why they are used
  - Their limitations

- We show evaluation techniques
  - How to use them
  - What to deduce from them

Metrics?

- What are the metrics you know?
  - height
  - weight
  - length
  - depth

Definition of Metrics

- A system or standard of measurement
  (Meriam-Webster Dictionary)

-> A metric is a value expressed with units associated to a property of an object, a person, a process...

Software Metrics

- A system or standard of measurement on software

-> A metric is a value expressed with units associated to a software
Why use metrics?

- Metrics give an indication of how the system is
- They are important because they allow people to compare systems and guess what they can achieve

Example
  - Alice is 1.8m tall and Bob is 1.6m tall
  - Alice is taller than Bob

Size-related metrics

- Lines of Code (Locs)
- Number of classes or header files
- Number of methods per class
- Number of attributes per class
- Size of compiled code
- Memory footprint

Lines of Code

- The most known (and used!) metrics to assess the general size of a project. Usually called locs. Can also be called SLOC (Source line of code). Sometimes:
  - 1 klocs = 1000 locs, 1 mlocs = 1000000 locs
- How to count it:
  > Remove all lines devoted to comments and blank lines then count the number of carriage return (\n' or '%N')
- Pitfalls: It is not completely accurate, e.g.
  ```
  print ("Hello World"); Result := 2
  vs
  print ("Hello World");
  Result := 2
  ```

Examples

- STRING_32 has 2766 locs
- EiffelBase has 65416 locs (in 251 classes)
- Apache has around 89000 locs
- EiffelStudio has 3 millions locs
- Windows XP has 40 millions locs

Evolution in time: Windows NT

Source for Data: Wikipedia, SLOC

Evolution in time: vsftpd

Source: Neamtiu, Hicks, Stayle, Oriol, (PLDI 2006)
Evolution in time: sshd

Source: Neamtiu, Hicks, Stoyte, Oriol, (PLDI 2006)

Number of classes or header files

- Another very used one
- How to count it:
  - Count classes (inner classes are not counted)
- Pitfalls:
  - It is very dependent on the programming style
  - E.g. C programmers have more difficulties with cutting application in different pieces with separate headers

Number of methods per class

- Used to devise how big classes are
- How to count it:
  - Count routines in classes
- Pitfalls:
  - The number of routines is more adapted to view the complexity of a class with a client point of view or with a maintenance point of view.
  - Providers tend to consider that a higher number indicates how complete the class is

Number of attributes per class

- Used to devise how complex the data is
- How to count it:
  - Count attributes in classes
  - In C: number of fields per struct
- Pitfalls:
  - People tend to think that a class with a higher number of attributes contains more complex information: this is plain wrong!
  - E.g. a class with 10 integer attributes is easier to understand than a class with 4 linked lists

Examples

- EiffelBase (Estudio 6.1) has 251 classes
- XCode for MacOSX Leopard has 14040 header files
- Java 6 SDK has 7556 classes

Examples

- EiffelBase (Eiffel Studio 6.1) has 58 routines per class on average (14541 routines in total, 251 classes)
- Gobo (in Eiffel Studio 6.1) has 9.6 routine per class on average (19214 routines in total, 1993 classes)
Examples

- EiffelBase has 2.3 attributes per class (578 attributes, 251 classes)
- GOb (in EiffelStudio 6.1) has less than one attribute per class on average (1237 attributes in total, 1993 classes)
- Time cluster (in EiffelStudio 6.1) has 2.18 attribute on average per class (72 attributes over 33 classes)

Evolution in time: additions and deletions of attributes

Source: Neamtiu, Hicks, Foster, (MSR'05)

Size of compiled code

- Used to devise how big the code to load in memory is (can be used to devise startup time)
- How to count it:
  - Use GUI
  - Under Unix: Is –al, size
- Pitfalls:
  - Compiled code is not really representative of the actual size in memory, but it is easier to compute
  - Architectures make these numbers change

Examples

- Firefox is 48.2 MB big (on MacOS X Leopard, including images and all)
- EiffelStudio 6.1 is 72 MB big (on MacOS X Leopard, including images and all)

Memory footprint/Performance

- Used to devise how big the code loaded in memory is, or how fast the code runs
- How to count it:
  - Measure at least 11 times the memory footprint (this can vary)
  - Use GUI tools
  - Under Unix: pmap
- Pitfalls:
  - Taking one measure is clearly NOT acceptable

Examples

- EiffelStudio 6.1: 157 MB

Source: Neamtiu, Hicks, Stoyle, Oriol, (PLDI 2006)
Quality/Complexity metrics

- Cyclomatic Complexity
- Number of states
- Bugs per lines of code
- Coupling metrics
- Inheritance metrics

Cyclomatic Complexity

- The metric measures the number of possible paths in a program (in a subroutine)
- How to count it:
  > M = E - N + 2P
  (M = Cyclomatic Complexity, E number of edges in control-flow graph, N number of nodes of the graph, P number of entries)
- Pitfalls:
  > Difficult to understand for non-specialists
  > What does it really measure?

Example

E = 5, N = 5, P = 1
M = 2

a := 1
b := a + 2
b > 3
Result := b

Yes: Result := a
No: b := a

Distinct states: 2

Number of distinct states

- The metric measures the number of states in a model of the program
- How to count it:
  > Have a model (sic!) and count the number of nodes
- Pitfalls:
  > Model generally non-trivial to make

Example

open
closed

Distinct states: 2

Bugs per line of code

- The metric measures the number of bugs found per line of code
- How to count it:
  > Count the number of bugs divide it by the number of lines of code
- Pitfalls:
  > What if the program is not sufficiently tested?
  > How to compare two different values? Is it relevant?
Examples

- Firefox 1.5: 0.00077 (locs: 2172520, open bugs: 169)
- Apache Http 2: 0.00065 (locs: 89000, open bugs: 58)

Interdependence

- The metric measures the number of interdependencies between two modules
- How to count it:
  - Count all direct references in one class to another
- Pitfalls:
  - How to count dependencies that use inheritance?
  - Are direct calls to be used?

Inheritance Metrics

- These metrics measure how much classes inherit one from another.
- How to count them:
  - Measure the number of parents or descendants
- Pitfalls:
  - How does that relate to the quality of the code?

Inheritance Example: EiffelBase

Examples

- In Eiffel most classes have a couple of parents (max 5 or 6, mostly 2 or 3)
- See new study at ECOOP 2008...

Process metrics

- Failed Builds
- Defects per hour
- Requirements changes
- Programming Time
- Patches after release
Failed Builds
- The metric measures the number of overnight build that failed
- How to count it:
  - Count the number of times the overnight build failed
- Pitfalls:
  - A higher value is not necessarily bad depending on the development model
  - Languages more restrictive on typing are more likely to have higher value

Defects per hour
- The metric measures the number of defects per hour of developer time
- How to count it:
  - Count the number of bugs divide it by the number of hours spent on the project
- Pitfalls:
  - Does not take into account the difficulty of programs

Examples
- Apparently more experienced programmers are responsible for more bugs than rookies.

How to explain that?

Requirements Changes
- The metric measures the number requirements changes
- How to count it:
  - Count the number of requirements that changed
- Pitfalls:
  - Who actually introduced the changes is also relevant!

Programming Time
- The metric measures the number of hours spent by programmers
- How to count it:
  - Ask programmers to fill in some form every week to track their time
- Pitfalls:
  - Is there really a correlation from one project to another

Patches after release
- The metric measures the number of patches distributed after the release
- How to count it:
  - Count patches
- Pitfalls:
  - Can work both ways:
    - Higher = higher maintenance
    - Higher = software more bug-prone
Did we present all Metrics?

**CLEARLY NOT!!!!**

How many???

- M. Xenos, D. Stavrinoudis, K. Zikouli and D. Christodoulaki, "OBJECT-ORIENTED METRICS - A SURVEY" in FESMA 2000
  - studied over 200 software metrics!!

- The question is:
  - what metrics are adapted to what use?

What about using several?

- As an example:
  - Maintainability =
    \[ 171 - 5.2 \ln(V) - 0.23V(G) - 16.2 \ln(L) + 50 \sin(\sqrt{2.4 \ C}) \]

How to avoid stupid things with metrics?

- Treat them like statistics: talk about correlation rather than implication, and then make an interpretation!

- Famous example from statistics:
  - People who wear a hat have more often cancer
    - Deduction: wearing hat provokes cancer

  Real interpretation: it might be that people who wear a hat are more exposed to cancer anyway (e.g. they are bald and thus more exposed to skin cancer)

Probably the most interesting recent work on metrics in the last few years... (1/3)

- Prof. Zeller and his team got access to 5 major projects at Microsoft and they tried 15 - 20 metrics on all projects to predict what modules would be more bug-prone once in production.

  Results: there is no single metric that predicts bugs for all projects!

Probably the most interesting recent work on metrics in the last few years... (2/3)

- Idea: combine metrics to make the One?
  - It does not work either!
Probably the most interesting recent work on metrics in the last few years: (3/3)

- Main result:
  - If a set of metrics is good in one project at one point, it is good to predict failures in future iterations!

The paradox of metrics

- Once a metric expresses a negative property and programmers tend to use it, then it becomes less and less valid to predict errors.
- It behaves like "Goto statements", Dijkstra

What metrics to use?

- Locs
- Number of classes
- Memory footprints
- Cyclomatic complexity for testing
- Number of states
- Programming time

Are all easy to use and do not really imply anything bad for programmers.

What metrics to use? (cont.)

In the end, use the ones that work for you!

How to present results

- Use graphs if possible in addition to raw data in tables
- When aggregating data (like in memory footprints), show confidence intervals
- Show the progression of values: make sure that the intuitive understanding is not contradicted by the actual graphs
- Label correctly all axes and make sure to explain all abbreviations
- Select carefully your data and the way you show them

Example: Present data and Graphs

Source: Neamtiu, Hicks, Stoyte, Oriol, (PLDI 2006)
Example: Confidence Intervals

Example: Intuitive Understanding

Example: Label Axes

Example: Select and present data

In practice: Metrics Tool in Estudio

In practice: Metrics Tool in Estudio
**Estimation Techniques**

- The basic measure: the man-month

**Discussion:**

How would you quantify the time needed to develop a web browser?

---

**Cone of Uncertainty**


**Cloud of Uncertainty**


---

**Narrowing the cloud**


**What makes it so unstable?**

- Forgetting necessary tasks!!!
  - **Missing requirements**
    - Functional and non-functional (e.g., setup, data conversion, glue code, help system, interfaces...)
  - **Missing software development activities**
    - E.g., new team members, creation of test data, coordination with sub-contractors...
  - **Missing other activities**
    - E.g., vacation, training, weekends, holidays, hardware problems

Source: McConnel, *Software Estimation*
First question: estimate the size

- Assumption: If you already coded some program that took around the same amount (up to factor 3) of work to code than the one you are going to make
- What is the size of the project?
  - How to express it?
  - Is it already starting to design?
  - What needs to be written?
  - How can an expert help?

Why estimate the size?

- The cost in terms of man-month is as follows: \( C \) and \( k \) are constants, usually the size is in \( KLoC \)

\[
\text{Cost} = c \times (\text{size})^k
\]

Source: Fundamentals of Software Engineering, Ghezzi, Jazayeri, Mandrioli

Basic COCOMO

- Good for rough and fast estimations
- Somewhat big variability

Intermediate COCOMO?

- \( E = a(KLoC)^b \cdot EAF \)

Software project \( a \), \( b \)
Organic 3.2 1.05
Semi-detached 3.0 1.12
Embedded 2.8 1.20
- EAF?

Effort Adjustment Factor

<table>
<thead>
<tr>
<th>Cost driver</th>
<th>Very Low</th>
<th>Low</th>
<th>Normal</th>
<th>High</th>
<th>Very High</th>
<th>Cost</th>
<th>Ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product attributes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Required software reliability</td>
<td>0.75</td>
<td>0.85</td>
<td>1.00</td>
<td>1.15</td>
<td>1.46</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>Size of application database</td>
<td>0.94</td>
<td>1.00</td>
<td>1.05</td>
<td>1.10</td>
<td>1.18</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>Completeness of project</td>
<td>0.70</td>
<td>0.80</td>
<td>1.00</td>
<td>1.50</td>
<td>2.00</td>
<td>0.70</td>
<td></td>
</tr>
<tr>
<td>Hardware attributes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor performance</td>
<td>1.00</td>
<td>1.11</td>
<td>1.30</td>
<td>1.40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory constraints</td>
<td>1.00</td>
<td>1.05</td>
<td>1.21</td>
<td>1.26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Utility of the virtual machine environment</td>
<td>1.00</td>
<td>1.05</td>
<td>1.15</td>
<td>1.20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Required virtual time</td>
<td>0.97</td>
<td>1.00</td>
<td>1.05</td>
<td>1.10</td>
<td>1.18</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>Personnel attributes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analyst capability</td>
<td>1.40</td>
<td>1.50</td>
<td>1.60</td>
<td>1.75</td>
<td>0.73</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applications experience</td>
<td>1.35</td>
<td>1.45</td>
<td>1.55</td>
<td>1.65</td>
<td>1.80</td>
<td>1.35</td>
<td></td>
</tr>
<tr>
<td>Software engineer capability</td>
<td>1.40</td>
<td>1.50</td>
<td>1.60</td>
<td>1.75</td>
<td>0.70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Utility of the virtual machine experience</td>
<td>1.21</td>
<td>1.30</td>
<td>1.40</td>
<td>1.50</td>
<td>2.00</td>
<td>1.21</td>
<td></td>
</tr>
<tr>
<td>Programming language experience</td>
<td>1.14</td>
<td>1.25</td>
<td>1.35</td>
<td>1.45</td>
<td>2.00</td>
<td>1.14</td>
<td></td>
</tr>
<tr>
<td>Project attributes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User of software tools</td>
<td>1.25</td>
<td>1.40</td>
<td>1.55</td>
<td>1.70</td>
<td>0.70</td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td>Application of software engineering tools</td>
<td>1.25</td>
<td>1.40</td>
<td>1.55</td>
<td>1.70</td>
<td>0.80</td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td>Required development schedule</td>
<td>1.20</td>
<td>1.40</td>
<td>1.60</td>
<td>1.80</td>
<td>1.10</td>
<td>1.20</td>
<td></td>
</tr>
</tbody>
</table>

The product of all these values…
Another Estimation: Function-Points

Idea: replace locs with function points

Types of FP:
- External Inputs: screens, forms, dialog boxes...
- External Outputs: screens, reports, graphs
- External Queries: input/output combination with a query leads to simple output
- Internal Logical Files: major groups of end-user-data
- External Interface Files: files controlled by other programs

Calculation

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Low</th>
<th>Med</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Inputs</td>
<td>x3</td>
<td>x4</td>
<td>x6</td>
</tr>
<tr>
<td>External Outputs</td>
<td>x4</td>
<td>x5</td>
<td>x7</td>
</tr>
<tr>
<td>External Queries</td>
<td>x3</td>
<td>x4</td>
<td>x6</td>
</tr>
<tr>
<td>Internal Logic Files</td>
<td>x4</td>
<td>x10</td>
<td>x15</td>
</tr>
<tr>
<td>External Interface Files</td>
<td>x5</td>
<td>x7</td>
<td>x10</td>
</tr>
</tbody>
</table>

- Count them, then multiply by the factors, then make the addition...

From Function Points to Locs

- Depending on the language it generates a different number of lines of code:
  - E.g.: C: 165.4, C#: 55, C++: 72.8, Cobol: 116, Smalltalk: 24.2, Java: 52.6, Eiffel: 21
  (This is not an advertisement for Eiffel, but...)

Source: http://www.softwareestimator.comIndustryData2.htm#SAIL

Today

- We presented software metrics
  - How they are calculated
  - What they represent
  - Why they are used
  - Their limitations
- We showed evaluation techniques
  - How to use them
  - What to deduce from them

Conclusions

- None of the techniques that were presented here are silver bullet
- They need to be adapted and tested in your environment with your people and to build up some know how
- That’s why being a project manager is implying keeping track of what people do and how projects went in the past