

CISC327 - Software Quality Assurance

Lecture 26

Software Product Metrics

Software Product Metrics

- Today's topic is **product metrics**, the measurements we can make of the actual software product itself
 - **External** product metrics
 - Defect metrics
 - **Internal** product metrics
 - Size metrics
 - Complexity metrics

External Product Metrics

- **Measures of the Software in its Environment**
 - **External** metrics are those we can apply only by observing the software product in its environment (e.g., by running it)
 - Includes many measures, but particularly:
 - **Failure** rate (number of failures per unit time)
 - **Availability** rate (percent of time system is "up")
 - **Defect** rate (number of defects per size of code)

Reliability

- **Definition of Reliability**
 - Reliability is the **probability** that a system will execute without failure in a given environment over a given period of time
 - **Implications:**
 - No single number for a given program
 - Depends on how the program is **used** (its environment)
 - Use **probability** to model our uncertainty
 - **Time** dependent

Reliability Metrics (recall)

- **Probability of failure on demand (POFOD)**
 - The **probability** that a demand for service from a system will result in a system failure
 - POFOD = 0.001 means that there is a 1/1000 chance that a failure will occur when a demand is made
- **Rate of occurrence of failures (ROCOF)**
 - The probable **number** of failures likely to be observed in a certain time period (e.g., 1 hour)
 - Reciprocal of ROCOF is the mean time to failure
 - ROCOF of 2 failures/h, mean time to failure = 30 min

Reliability: Definition of Failure

- **Formal** view:
 - Any deviation from **specified** behaviour
- **Engineering** view:
 - Any deviation from **required**, specified, or **expected** behaviour
 - Required (by environment)
 - Expected (by user)

Errors, Faults, and Failures

- **Definitions**

- a (human) error is a mistake or oversight on the part of a designer or programmer, which **may** cause...
- a fault, which is a mistake in the software, which in turn **may** cause...
- a failure when the program is run in certain situations

- **Defect:** usually defined as a fault or a failure

- **Defects = Faults + Failures**
- but sometimes **only Faults**, or **only Failures**

Defect Density Metric

- Defect Density

- a standard reliability metric

$$DD = (\text{Number of defects found}) / (\text{System size})$$

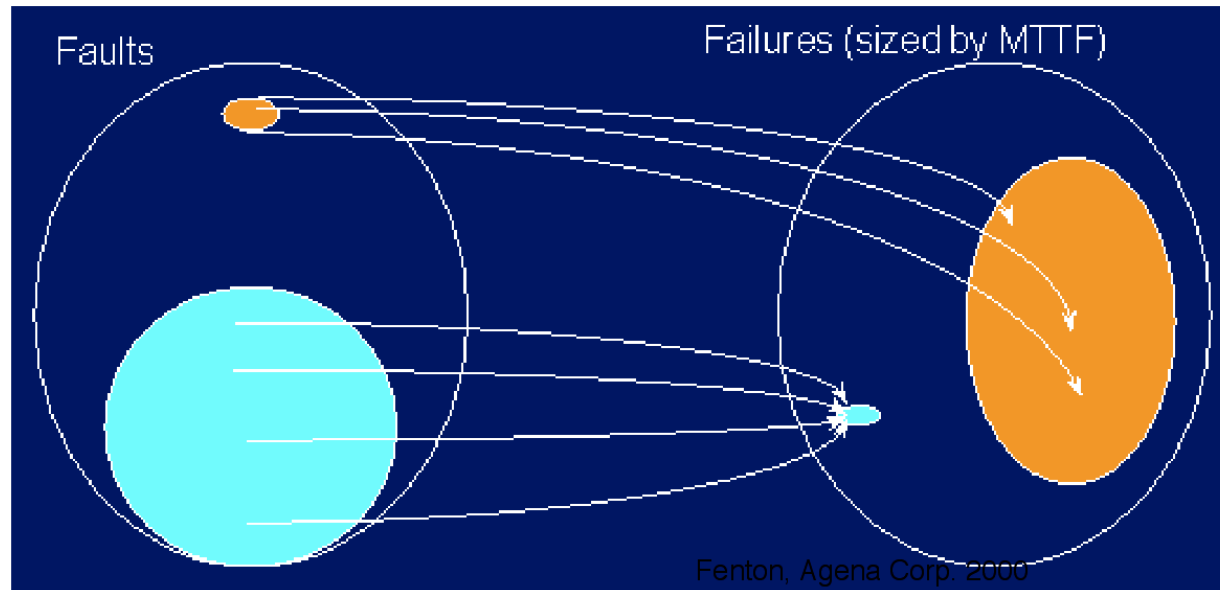
- Size is often measured in **KLOC** (1000s of lines of code), so units of defect density are defects found per 1000 lines
- Widely used as an indicator of software **quality**

Predictive Power of Defect Density

- However..
 - Unfortunately, **faults** are not a good predictor of **failures**, and vice versa
 - **35%** of faults cause only about **1%** of failures, and **35%** of failures are caused by only about **2%** of faults

Predictive Power of Defect Density

- 35% of faults cause only about 1% of failures, and 35% of failures are caused by only about 2% of faults



- This finding makes historical **defect density** look like not such a good **predictor** of quality

A Process Metric Using Defects

- **Effectiveness of Defect Detection**
 - Defect statistics can also be used for evaluating and improving software **process**
 - For example, defect metrics have been used to show the effectiveness of **inspection** vs. **testing**

Testing Type	Defects found per hour
Regular use	0.210
Black box	0.282
White box	0.322
Reading/Inspections	1.057

Internal Product Metrics

- **Measures of the Product Itself**
 - The vast majority of metrics in practice are **internal** product metrics, measures of the software **code**, **design**, or **functionality**, independent of its environment
 - The U.S. military lists literally **hundreds** of measures of code alone
 - These measures are easy to make and easy to automate, but it's not always clear which **attributes** of the program they characterize (if any)

Code Metrics

- **Software Size**

- The simplest and most enduring product metric is the **size** of the product, measured using a count of the number of lines of source code (**LOC**), most often quoted in 1000's (**KLOC**)
- Used in a number of other **indirect** measures, such as:
 - **Productivity** (LOC / effort)
 - Effort or **cost** estimation (Effort = $f(\text{LOC})$)
 - **Quality** assessment or estimation (defects / LOC)
- Many similar measures are also used:
 - **KDSI** (1000's of delivered source instructions)
 - **NLOC** (non-comment lines of code)
 - Number of **characters** of source or **bytes** of object code

Problems with LOC Measures

- **What Attribute is Measured?**
 - LOC really measures **length** of a program (a physical characteristic), not **size** (a logical characteristic)
 - Mistakenly used as a surrogate for measures of what we're really interested in
 - **Effort, complexity, functionality**
 - Does not take into account **redundancy** or **reuse**
 - Cannot be compared across different programming **languages**
 - Can only be measured at **end** of development cycle

Problems with LOC Measures

- How many lines of code are there?
- How many statement lines of code?

```
for (i=0; i<100; ++i) printf("hello %d\n", i);
```

```
printf("hello 0\n");  
printf("hello 1\n");  
printf("hello 2\n");  
printf("hello 3\n");  
... 96 more lines ...
```

```
printf("hello 0\nhello 1\nhello 2\nhello 3\n    ...99\n");
```

Better Size Measures

- **Fundamental Size Attributes of Software**
 - Length: The physical size of the software
 - LOC will do as measure
 - Functionality: The capabilities provided to the user by the software
 - How big or rich is the set of functions provided?
 - Somewhat tricky unless we are comparing subsets and supersets—but sometimes we are:
 - version 1.0: features 1 through 50
 - version 1.1: features 1 through 56
 - version 2.0: features 1 through 100
 - version 2.0 \supseteq version 1.1 \supseteq version 1.0
 - Complexity: How complex is this software?

“Lines of code metrics distort the true economic case by so much that their use for economic studies involving more than one programming language might be classified as professional malpractice.

“The only situation where LOC metrics behave reasonably well is when two projects utilize the same programming language.”

—Capers Jones,

Applied Software Measurement (2008)

Better Size Measures

- **Complexity**: How complex is this software?
 - **Problem complexity**: Measures the complexity of the underlying problem
 - **Algorithmic complexity**: Measures the complexity or efficiency of the solution implemented by the software
 - **Structural complexity**: Measures the structure of the program used to implement the algorithm (includes control structure, modular structure, data flow structure, and architectural structure)
 - **Cognitive complexity**: Measures the effort to understand the software

Code Complexity Metrics

- **Better Measures of Source Code**

- Need something closer to **cognitive** complexity
⇒ work on **complexity metrics** for code

- Early work measured characteristics such as:

- **Number/density** of decision (**if**) statements
- **Number/depth** of blocks/loops
- **Number/average length** of methods/classes

- Best-known and accepted source code complexity measures are:

- Halstead's "**Software Science**" metrics
- McCabe's "**Cyclomatic Complexity**" and "**Data Complexity**"

"Software Science" Metrics (Halstead 1977)

- **Operators and Operands**

- **Operators**: Reserved language words and language operators

- Examples: **if**, **return**, **this**, **+**, **!=**, **>>**, etc..

- **Operands**: Identifiers, type names, and character, numeric, or string constants

- Examples: **int**, **bool**, **void**, **x**, **y**, **1**, **"Hello"**, etc..

"Software Science" Metrics (Halstead 1977)

- Operators and Operands

- Program **source code** considered as a sequence of “**tokens**”, each of which is either an **operator** or an **operand**

- η_1 = number of unique (different) operators
- η_2 = number of unique (different) operands
- N_1 = total number of operator uses
- N_2 = total number of operand uses

- **Length** of program $N = N_1 + N_2$

- **Vocabulary** of program $\eta = \eta_1 + \eta_2$

"Software Science" Metrics (Halstead 1977)

- The **Software Science Predictive Theory**
 - Using η_1 , η_2 , N_1 , and N_2 as a basis, Halstead formulated a theory of software complexity and effort
 - Theory 1: An **estimate** of program length N is

$$N \approx \eta_1 \log \eta_1 + \eta_2 \log \eta_2$$

- Theory 2: Effort E required to create program P is

$$E = (\eta_1 N_2 N \log N) / (2 \eta_2) = \text{Volume} * \text{Difficulty}$$

- Theory 3: Time T required to create program P is

$$T = E / 18 \text{ seconds}$$

McCabe's "Cyclomatic Complexity" Metric

- Flow Graphs Again

- If the control flow graph G of program P has e edges and n nodes, then the **cyclomatic complexity** v of P is:

- $v(P) = e - n + 2$

- $v(P)$ is the number of linearly independent **paths** in G

- Example: $e = 16$, $n = 13$ $v(P) = 16 - 13 + 2 = 5$

- McCabe **proposed** that for each module of code (e.g., method) P , $v(P) < 10$

McCabe's "Cyclomatic Complexity" Metric

- **Flow Graphs Again**

- If all decisions are **two-way** (no switch statements), can simplify
- Let n_{Dec} be the number of **decision nodes**, such as an *if*, an *else-if*, or the start (or end) of a *while* loop, with 2 out-edges
- Let n_{NoDec} be the number of **non-decision nodes**, such as the body of an *if-then* with no *else*, with 1 out-edge (except the last node, which has 0 out-edges)
- Every node is either a decision or a non-decision node, so

$$n = n_{Dec} + n_{NoDec}$$

- Then McCabe's formula $e - n + 2$ becomes

$$\begin{aligned} & e - (n_{Dec} + n_{NoDec}) + 2 \\ &= (2 n_{Dec} + 1 n_{NoDec} - 1) - (n_{Dec} + n_{NoDec}) + 2 \\ &= 2 n_{Dec} + 1 n_{NoDec} - 1 - n_{Dec} - n_{NoDec} + 2 \\ &= 1 n_{Dec} + 1 \cdot 0 - 1 + 2 \\ &= n_{Dec} - 1 + 2 \\ &= n_{Dec} + 1 \end{aligned}$$

Other Flowgraph Metrics

- **Flowgraph Complexity of Software**
 - **McCabe** is just one of many **flowgraph-based** complexity metrics, all with the advantage that they are independent of programming language
 - Others measure things like:
 - Maximum **path length**
 - Number/interaction of **cycles**
 - Maximum number of alternative paths (a.k.a. **width** or "**fan-out**")
 - All can be **automatically** computed once the flowgraph is known (and it can be automatically computed too)

Other Flowgraph Metrics

- **Flowgraph Complexity of Software**
 - Flowgraph **decomposition**, which **partitions** flowgraphs into independent **one-node-in / one-node-out** subgraphs, provides a rigorous general theory of **structured programming**
 - And it can **also** be automatically computed!

Summary

- **Software Product Metrics**
 - Failures and faults are the basis of **defect density** metrics, widely used as an indicator of software quality
 - **Code metrics** began with simple size (**LOC**), but have evolved to try to encompass the notion of cognitive **code complexity**
 - **Flow graphs** form the basis of many code complexity metrics
- **References**
 - Kan ch. 6.1–6.3 Defect Removal Effectiveness
 - Kan ch. 10 Complexity Metrics and Models

- Next time
 - Software Process Metrics