Software Testing

Metrics and Measurement

What have we seen so far?

- Testing techniques (white box, black box etc)
- Coverage measures
- Reviews and inspections
- A little bit on measurement
- Static analysis
- Plans and strategies
- Technology specific testing
- Testing in practice

What will we see next?

- Measurement and metrics
  - What is it?
  - Some very basic measurement theory
  - Product metrics
    - E.g., external metrics like defects
    - E.g., internal metrics like size and complexity
- Won't get to process metrics (e.g., COCOMO)

Software Quality Metrics

- Applying measurement to software
- *Software metrics* are measurable properties of software systems, their development, and their use.
- Wide assortment:
  - Properties of the product itself.
  - The process of producing and maintaining it
  - Its source code, designs, tests etc.

Example Metrics

- Number of failures
- Time to build
- Number of lines of code (boring)
- Number of failures per 1000 lines of code (more interesting)
- Number of lines of code per programmer per month (should be fired out of a cannon)
- Number of decisions per 1000 lines of code (more interesting)

Why do we care?

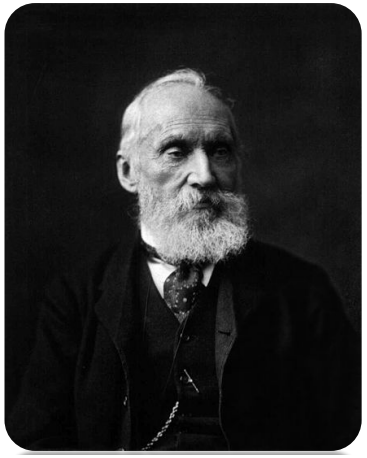
- *Reliability and quality control*
  - Prediction and control of quality
  - Example: by measuring the relative effectiveness of defect detection when dropping different testing/inspection methods we can choose the best one for our products.
- *Cost estimation and productivity improvement*
  - They help us predict effort for producing or maintaining software, and to improve scheduling
  - Example: by measuring code production using different languages/IDEs we choose those that give better results.
- *Quality improvement*
  - They can help improve code quality and maintainability
  - Example: by measuring complexity of code we can identify those parts most likely to fail, or to be difficult to maintain

Kinds of Metrics

- ***Product metrics:*** those that describe the internal and external characteristics of the product itself.
  - Examples: size, complexity, features, performance, reliability, quality level
  - Most common metrics are of this kind.
- ***Process metrics***: measure process of software dev and maintenance so as to improve it
  - Examples: effectiveness of defect removal during development; pattern of defect arrival during testing; response time for fixes
- ***Project metrics:*** describe project characteristics
  - Examples: number of devs, dev cost, schedule, productivity

Measurement Basics

- **If you want to know, measure**
  - "When you can measure what you are speaking about and can express it in numbers, you know something about it. But when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre kind."
    - William Thomson (Lord Kelvin)



Measurement Basics

- **… but make sure you know what you are measuring**
  - "In truth, a good case could be made that if your knowledge is meagre and unsatisfactory, the last thing in the world you should do is make measurements. The chance is negligible that you will measure the right things accidentally."
    - George A. Miller, Psychologist

Measurement Basics

- Definition: *measurement* is the process of empirical objective assignment of numbers to entities, in order to characterize an attribute.
- What does this mean?
  - An *entity* is an object (e.g., program) or event.
  - An *attribute* is a feature or property of an entity, such as its size.
  - *Objective* means measurement must be defined on a well-defined rule whose results are repeatable (e.g., counting LOCs)
- In other words:
  - Each entity is given a number which tells you about its attribute
  - Example: each source program has a line count, which tells you something about its size.

Example Measurements

ENTITY	ATTRIBUTE	MEASURE
Person	Age	Years at last birthday
Person	Age	Months since birth
Source code	Length	# Lines of Code (LOC)
Source code	Length	# Executable statements
Testing process	duration	Time in hours from start to finish
Tester	efficiency	Number of faults found per KLOC
Testing process	fault frequency	Number of faults found per KLOC
Source code	quality	Number of faults found per KLOC
Operating system	reliability	Mean Time to failure rate of occurrence of failures

Common Mistakes in Measurement

- It's super easy to make mistakes in choosing what or how to measure software characteristics
- To avoid mistakes, stick to the definition of measurement.

1. You must specify both an entity and an attribute, not one or the other.
   - Example: you're not measuring a program but a property of a program
   - Example: you're not measuring the size of the software, say what artifact of the software you're measuring (e.g., code, design models)
2. Define the entity precisely.
   - Example: not just "program" but "program source code" (confusion with tests, scaffolding, config files…)

Common Mistakes in Measurement

3. You need a good intuitive understanding of the attribute before you propose a measure for it.
   - Example: it's pretty obvious that size is related to number of source lines
- It's a mistake to propose a measure if there's no consensus on what attribute it's characterizing.
  - Example: number of defects per KLOC characterizes quality of code or quality of testing?
- It's a mistake to redefine an attribute to fit an existing measure.
  - Example: if we've measure defects found this month, don't mistake that as an indicator of quality!

Kinds/Uses of Measurement

- Kinds of measurement: direct and indirect.
- Uses of measurement.
  - Assessment: evaluating how things are now.
  - Prediction: attempting to assess how things will be in the future.
  - Measurement for prediction requires a prediction system.

Direct Measurement

- Direct measures are numbers that can be derived directly from an entity without any other information.
- Examples:
  - Length of source code (number of lines)
  - Duration of test process (number of elapsed hours)
  - Number of defects discovered during the testing process (measured by counting defects)
  - Effort of a programmer on a project (measured by person-months worked)

Indirect Measurement

- Indirect measures are numbers that are derived by combining two or more direct measures to characterize an attribute.
- Examples:
  - Programmer productivity = LOC produced/PM of effort
  - Program defect density = Num defects/Length of source code
  - Requirements stability = Original number of requirements/total number of requirements
  - Test effectiveness ratio = Number of items covered / total number of items

Predictive Measurement

- A prediction system is required
- A prediction system consists of:
  - A mathematical model
    - Example: $E = a*S^b$, where E is effort to be predicted, S is the estimated size in LOC, and a and b are constants
  - A procedure for determining the model parameters.
    - Example: determine a and b from past project data.
  - A procedure for interpreting results.
    - Example: use Bayesian probability analysis to determine the likelihood that our prediction is accurate within 10%

Product Metrics

- 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:
  - # failures/unit of time
  - availability rate (% of time system is “up”)
  - defect rate (#defects/LOC)

Reliability

- *Reliability* is the probability that a system will execute without failure in a given environment over a given period of time (e.g., a year)
- 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
- Definition of failure?
  - Formal view:
    Any deviation from specified behaviour
  - Engineering view:
    Any deviation from required, specified or expected behaviour

Defect Density Metric

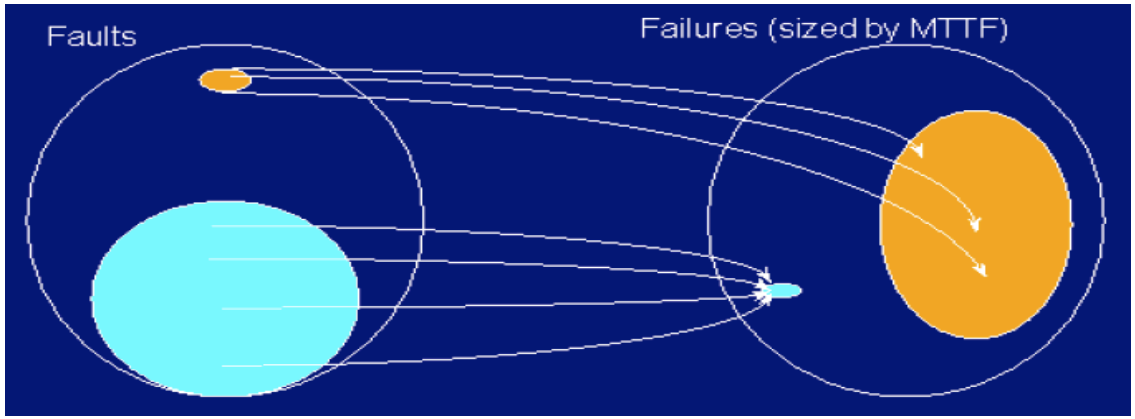
- A defect is usually defined as a fault or a failure: Defects = Faults + Failures
  - (or sometimes just Faults or just Failures)
- Defect density is a standard reliability metric:

  - Defect Density = $\frac{\text{Number of defects found}}{\text{System Size}}$

- Size is normally measured in KLOC (1000's 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

- Unfortunately, faults are not a good predictor of failures, and vice versa (Adams 1984)



- 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

Defects and Process

- Defect statistics can also be used for evaluating and improving software process

Testing Type	Defects found per hour
Regular use	0.21
Black box	0.282
White box	0.322
Reading/Inspections	1.057

Source: Fenton, Agena Corp. 2000

- Grady (1992) used defect metrics to show effectiveness of inspection vs. testing

Internal Product Metrics

- The vast majority of metrics in practice are internal product metrics, measures of the software code, design or functionality itself, 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)
- It is used in a number of other indirect measures, such as
  - productivity (LOC / effort)
  - effort / cost estimation (Effort = f(LOC)
  - quality assessment / 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

- LOC really measures length of 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, reuse (e.g., **XXXX** Bank - 500 MLOC, only about 100 MLOC unique)
- Cannot be compared across different programming
- languages
- Can really only be usefully measured at end of development cycle

Better Size Measures

- *Length* - the physical size of the software (LOC will do as measure)
- *Functionality* - the capabilities provided to the user by the software
  - (how big / rich is the set of functions provided)
- *Complexity* - how complex is this software?
  - *Problem complexity* - measures the complexity of the underlying problem
  - *Algorithmic complexity* - measures the complexity / 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 Measures

- Realization that we need something better approaching cognitive complexity led to work on complexity metrics for code
- Early explorations measured characteristics such as:
  - number / density of decision (if) statements
  - number / depth of blocks / loops
  - number / average length of methods / classes
  - and many more...
- Best known and accepted source code complexity measures are
  - Halstead's "Software Science" metrics
  - McCabe's "Cyclomatic Complexity" and "Data Complexity" metrics

Example: McCabe Complexity

- 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

  $e = 16$ $n = 13$

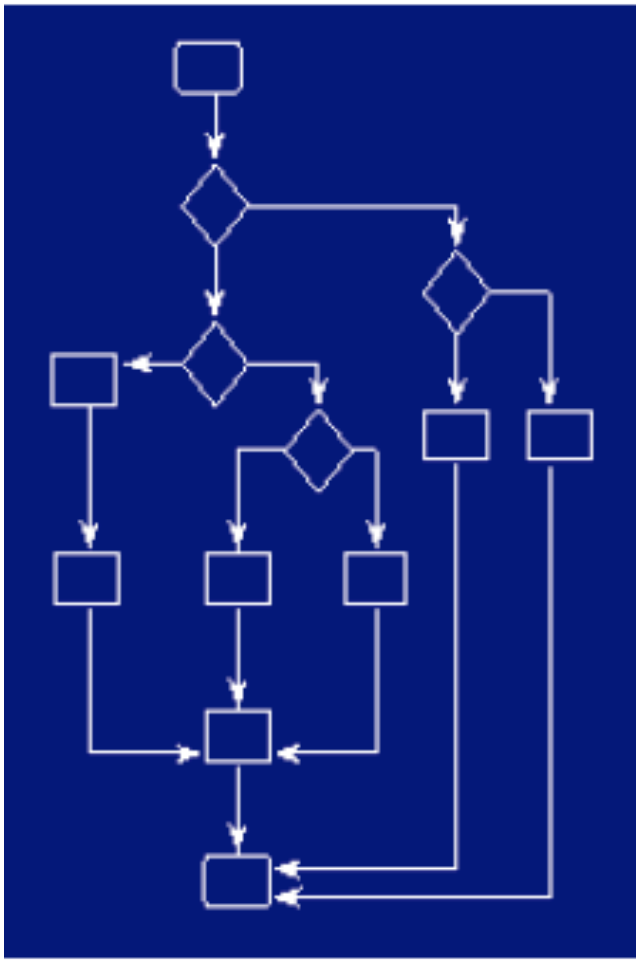
  $v(P) = 16-13+2 = 5$

- More simply, if **d** is the number of decision nodes in **G** then

  $v(P) = d + 1$

- McCabe proposed that for each module **P**

  $v(P) < 10$



A little about project measurement

- Much of project measurement is about cost estimation.
  - $$$, time, resources
- Project measurement provides a link between economics and software engineering.
- Software cost estimation techniques can help in providing better software project management.

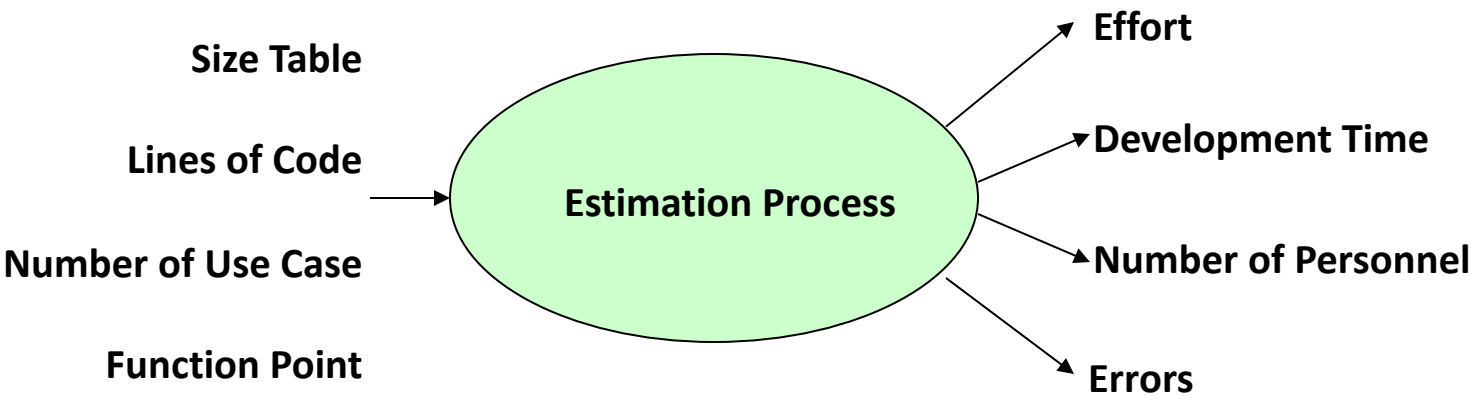
Cost of a project

- The cost in a project is due to:
  - the requirements for software, hardware and human resources
  - the cost of software development is due to the human resources needed
  - most cost estimates are measured in ***person-months (PM)***

Cost of a project (…)

- the cost of the project depends on the nature and characteristics of the project,
- at any point, the accuracy of the estimate will depend on the amount of reliable information we have about the final product.
  - If we have not been working in a domain for a while we may not trust our data (or data may simply be unavailable)

Cost Estimation Process



Project Size - Metrics

1. Number of functional requirements
2. Cumulative number of functional and non-functional requirements
3. Number of Customer Test Cases
4. Number of ‘typical sized’ use cases
5. Number of inquiries
6. Number of files accessed (external, internal, master)
7. Total number of components (subsystems, modules, procedures, routines, classes, methods)
8. Total number of interfaces
9. Number of System Integration Test Cases
10. Number of input and output parameters (summed over each interface)
11. Number of Designer Unit Test Cases
12. Number of decisions (if, case statements) summed over each routine or method
13. Lines of Code, summed over each routine or method

Introduction to COCOMO models

- The **COstructive COst Model** (COCOMO) is the most widely used software estimation model.
- The COCOMO model predicts the **effort** and **duration** of a project based on inputs relating to the size of the resulting systems and a number of "**cost drivers**" that affect productivity.

COCOMO Models

- COCOMO is defined in terms of three different models:
  - the **Basic model**,
  - the **Intermediate model**, and
  - the **Detailed model**.
- The more complex models account for more factors that influence software projects, and (in theory) make more accurate estimates.

The Development mode

- The most important factors contributing to a project's duration and cost is the Development Mode
  - **Organic Mode:** The project is developed in a familiar, stable environment, and the product is similar to previously developed products. The product is relatively small, and requires little innovation.
  - **Semidetached Mode:** The project's characteristics are intermediate between Organic and Embedded.
  - **Embedded Mode:** The project is characterized by tight, inflexible constraints and interface requirements. An embedded mode project will require a great deal of innovation.

Effort Computation

- The **Basic COCOMO model** computes effort as a function of program size. The Basic COCOMO equation is:
  - ***E = a*KLOC^b***
- Effort for three modes of Basic COCOMO.

Mode	**a**	**b**
Organic	2.4	1.05
Semi-detached	3.0	1.12
Embedded	3.6	1.20

Example

Mode	Effort Formula
Organic	$E = 2.4 * S^{1.05}$
Semidetached	$E = 3.0 * S^{1.12}$
Embedded	$E = 3.6 * S^{1.20}$

- Suppose S = 200 KLOC
- Organic: $E = 2.4 * 200^{1.05} = 626$ PM
- Embedded: $E = 3.6 * 200^{1.20} = 2077$ PM

Effort Computation

- The **intermediate COCOMO model** computes effort as a function of program size and a set of cost drivers. The Intermediate COCOMO equation is:
  - ***E = a*KLOC^b*EAF***
- Effort for three modes of intermediate COCOMO.

Mode	a	b
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

Effort Adjustment Factors

Cost Driver	Very Low	Low	Nominal	High	Very High	Extra High
Required Reliability	.75	.88	1.00	1.15	1.40	1.40
Database Size	.94	.94	1.00	1.08	1.16	1.16
Product Complexity	.70	.85	1.00	1.15	1.30	1.65
Execution Time Constraint	1.00	1.00	1.00	1.11	1.30	1.66
Main Storage Constraint	1.00	1.00	1.00	1.06	1.21	1.56
Virtual Machine Volatility	.87	.87	1.00	1.15	1.30	1.30
Comp Turn Around Time	.87	.87	1.00	1.07	1.15	1.15
Analyst Capability	1.46	1.19	1.00	.86	.71	.71
Application Experience	1.29	1.13	1.00	.91	.82	.82
Programmers Capability	1.42	1.17	1.00	.86	.70	.70
Virtual machine Experience	1.21	1.10	1.00	.90	.90	.90
Language Experience	1.14	1.07	1.00	.95	.95	.95
Modern Prog Practices	1.24	1.10	1.00	.91	.82	.82
SW Tools	1.24	1.10	1.00	.91	.83	.83
Required Dev Schedule	1.23	1.08	1.00	1.04	1.10	1,10

Summary

- Measurement is about characterizing the attributes of entities
  - can be direct or indirect
  - can be for either assessment or prediction
- Any framework for software measurement is based on:
  - classifying entities as products, processes and resources
  - classifying attributes as internal and external
  - determining whether we want assessment or prediction
- Product metrics include both external metrics (e.g., defect) and internal metrics (e.g., size, complexity)

- Any questions?

  Now

  By email