

Software Metrics

Metrics are tools that are used to estimate the cost and resource requirements of a project.



Successful Software Projects



- In order to conduct a successful software project we must understand
 - the scope of work to be done
 - the risks incurred
 - the resources required
 - the tasks to be accomplished
 - the milestones to be tracked
 - the cost
 - the schedule to be followed

Project management



- Before a project can be planned
 - objectives and scope should be established
 - alternative solutions should be considered
 - technical and management constraints should be identified
- This information is required to estimate costs, tasks, and a schedule.

Metrics



- Metrics help us understand the technical process that is used to develop a product.
 - The process is measured to improve it and the product is measured to increase quality.

Metrics



- Measuring software projects is still controversial.
 - It is not yet clear which are the appropriate metrics for a software project or whether people, processes, or products can be compared using metrics.

Project Planning



- Estimates for project cost and time requirements must be derived during the planning stage of a project.
- Experience is often the only guide used to derive these estimates, but it may be insufficient if the project breaks new ground.

Estimation Techniques



- A number of estimation techniques exist for software development.
- These techniques consist of
 - establishing project scope using software metrics based upon past experience to generate estimates
 - dividing the project into smaller pieces which are estimated individually

Why Use Metrics?



- Without measurements there is no real way to determine if the process is improving.
- They allow the establishment of meaningful goals for improvement.
 - A baseline from which improvements can be measured can be established.

Why Use Metrics?



- Metrics allow an organization to identify the causes of defects that have the greatest effect on software development.

Applying Metrics



- When metrics are applied to a product they help identify:
 - which user requirements are likely to change
 - which modules are most error prone
 - how much testing should be planned for each module

A Baseline



- A baseline consists of data collected from past software development projects.
- It consists of size or function oriented measures and may also contain quality metrics.
- The baseline provides information for the strategic, project, and technical level.

A Baseline



- It provides a series of measures that can be used to help predict the time and costs of software development activities.
- All measures in a baseline should be reasonably accurate, collected from as many projects as possible, use consistent measures, and should be derived from similar types of app's.

Metrics for Productivity and Quality



- Productivity and quality are measures of the output as a function of effort and fitness of use of the output respectively.
- For planning and estimation, historical data are used to aid in predicting more accurately.

Software is measured to



- indicate the quality of the product
- assess the productivity of the people who produce the product
- assess the benefits derived from new software engineering tools and methods
- form a baseline for estimation
- help justify requests for new tools or training

Direct Measurements



- Measurements can be either direct or indirect.
- Direct measures are taken from a feature of an item (e.g. length).
 - Direct measures in a product include lines of code (LOC), execution speed, memory size, and defects reported.

Indirect Measurements



- Indirect measures associate a measure to a feature of the object being measured (e.g. quality is based upon counting rejects).
 - Indirect measures include functionality, quality, complexity, efficiency, reliability, and maintainability.

Direct and Indirect Measurements



- Direct measures are generally easier to collect than indirect measures.
- Size-oriented metrics are used to collect direct measures of software engineering output and quality.
- Function-oriented metrics provide indirect measures.

Size-Oriented Metrics



- Size-oriented metrics are a direct measure of software and the development process.
- These metrics can include:
 - effort (time)
 - money spent
 - KLOC (1000s lines of code)
 - pages of documentation created
 - errors
 - people on the project

Size-Oriented Metrics



- From this data some simple size-oriented metrics can be generated.
 - Productivity = KLOC / person-month
 - Quality = defects / KLOC
 - Cost = Cost / KLOC
 - Documentation = pages of documentation / LOC

The Traditional Measure: Lines of Code



- The most widely used measure of source code length is the number of lines of code (LOC).
- But...not all lines are created equal!
 - Blank line (makes code more readable)
 - Comment line (improves understandability)
 - Data declarations
 - Lines containing several instructions

Definition of a LOC



- Any line of program text that is not a comment or a blank line, regardless of the number of statements or fragments of statements on the line (Conte, Dunsmore, and Shen, 1986)
- A non-commented source statement – any statement in the program except for comments and blank lines (Grady and Caswell, 1987)

Drawing Conclusions from the Definition



- If length is related to effort then
 - Comments are not effort
- The total length of a program (LOC) can be redefined to be
 - $NCLOC + CLOC$
- Non-commented plus commented LOC.
- This leads to a useful indirect measure of $CLOC/LOC$
 - density of comments (self-documentation)

Executable Statements



- Some LOC are not executable
 - Data declarations
 - Header statements
- For testing purposes, it might be more useful to know the length of the program in terms of executable statements (ES).
 - Ignores comments, data declarations, headers and counts separate statements on one physical line as distinct.

Delivered Source Instructions



- The amount of code delivered can be significantly different than the actual amount of code developed for the project.
 - Drivers, stubs, scaffolding, prototypes
- DSI distinguishes between LOC written and LOC delivered to the customer.

Size-Oriented Metrics



- Size-oriented metrics are not universally accepted.
- The use of LOC as a key measure is the center of the conflict.

Proponents of the LOC measure claim



- It is an artifact of all software engineering processes which can easily be counted.
- Many existing metrics exist which use LOC as an input.
- A large body of literature and data exist which is predicated on LOC.

Opponents of the LOC measure claim



- That it is language dependent.
- Well designed short programs are penalized.
- They do not work well with non-procedural languages.
- Their use in planning is difficult because the planner must estimate LOC before the design is completed.

Halstead's Software Science



- In 1976, Maurice Halstead attempted to develop a methodology for capturing the size and complexity of programs.
- His work has had a lasting impact even though it provides confused and inadequate software metrics.

Halstead's Theory of Software Science



- Probably the best known and most thoroughly studied composite measures of software complexity.
- It proposes the first analytical laws for computer software.
- These laws have generated substantial controversy - there is not universal agreement that these laws are correct.

Physics Envy



- Software science assigns quantitative laws to the development of computer software.
- A set of primitive measures are used which may be derived after code has been generated or estimated once design is complete.
- Unlike other metrics, it is amenable to experimental verification.

The Primitive Measures



- λ_1
 - the number of distinct operators
- λ_2
 - the number of distinct operands
- N_1
 - the total number of operator occurrences
- N_2
 - the total number of operand occurrences

Use of the Primitive Measures



- These measures are used to develop expressions for
 - the overall program length
 - the potential minimum volume for an algorithm
 - the actual volume (number of bits required to specify a program)
 - the program level (a measure of software complexity)

Program Length and Volume

- The length N can be estimated by

$$N = \lambda_1 * \log_2 \lambda_1 + \lambda_2 * \log_2 \lambda_2$$

- Program volume may be defined by

$$V = N * \log_2 (\lambda_1 + \lambda_2)$$

- V will vary with programming language.
- It represents the volume of information (in bits) required to specify a program.
- Theoretically, a minimum volume must exist for a particular algorithm.



Volume Ratio

- Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program.

- L must always be less than one.

- The volume ratio may be expressed as

$$L = (2 / \lambda_1) * (\lambda_2 / N_2)$$



Language Level

- Halstead proposed that each language may be categorized by language level, l , which will vary among languages.
- Halstead thought that language level was constant for a given language, but other research indicates that language level is a function of both language and programmer.
- Language level implies a level of abstraction in the specification of procedure.



Halstead's Language Levels

- The following language levels (*Mean/l*) were derived empirically:

- English prose 2.16
- PL/1 1.53
- Algol/68 2.12
- Fortran 1.14
- Assembler 0.88



Measuring Functionality



- The amount of functionality that is inherent in software is often a better measure than its size.
- Most functionality metrics measure the functionality of specification documents but can be applied later in the life-cycle.

Function-Oriented Metrics



- Function-oriented metrics are indirect measures of software which focus on functionality and utility.
- The first function-oriented metric was proposed by Albrecht (IBM, 1979) who suggested a productivity measurement approach called the function point method.

Function Points



- FPs are derived from countable measures and assessments of software complexity.
- An unadjusted function point count or UFC is calculated based on five characteristics.

The Five Characteristics



1. number of user inputs
2. number of user outputs
3. number of user inquiries (on-line inputs)
4. number of files
5. number of external interfaces (tape, disk)

Calculating the UFC



- Each item is assigned a subjective complexity rating and weighted accordingly:

	Simple	Avg	Complex
# of user inputs	3	4	6
# of user outputs	4	5	7
# of user inquiries	3	4	6
# of files	7	10	15
# of ext interfaces	5	7	10

Calculating the TCF



- Next a technical complexity factor or TCF is calculated.
- There are 14 contributing factors for the TCF.

The Contributing Factors in the Form of Questions



- Does the system require reliable backup and recovery?
- Are data communications required?
- Are there distributed processing functions?
- Is performance critical?
- Will the system run in an existing, heavily utilized operational environment?

The Questions



- Does the system require on-line data entry?
- Does the on-line data entry require the input transaction to be built over multiple screens or operations?
- Are the master files updated on-line?
- Are the inputs, outputs, files or inquiries complex?
- Is the internal processing complex?

The Questions

- Is the code designed to be reusable?
- Are conversion and installation included in the design?
- Is the system designed for multiple installations in different organizations?
- Is the application designed to facilitate change and ease of use by the user?

Calculating TCF

- Each component is rated from 0 to 5 where
 - 0 is No Influence
 - 1 is Incidental
 - 2 is Moderate
 - 3 is Average
 - 4 is Significant
 - 5 is Essential

Calculating TCF

- The 14 ratings are combined to give:
$$TCF = 0.65 + 0.01 * \sum Fi$$
where Fi are the 14 complexity adjustment values.
- The final calculation multiplies UFC and TCF to give

$$FP = UFC * TCF$$

Using Function Points

- Once calculated, FPs may be used in place of LOC as a measure of
 - Productivity
 - Quality
 - Cost
 - Documentation
 - Other attributes

Pros and Cons



- Proponents of FPs claim that they are language independent and are based upon data that are more likely to be known early in a project.
- Opponents claim that the method is too subjective and that FPs have no direct physical meaning, it's just a number.

Problems with FPs



- Subjectivity in the technical factor.
- Double-counting
- Counter-intuitive values
- Accuracy
 - particularly with TCF
- Early life-cycle use
 - requires a full specification

Problems with FPs



- Changing requirements
- Differentiating specified items
- Technology dependence
- Application domains
 - used mostly in data-processing apps, not real-time or scientific apps
- Subjective weighting
- Measurement theory

Measurement Theory



- FP calculations combine measures from different scales.
 - weights and TCF ratings are on an ordinal scale while the counts are on a ratio scale
 - linear combinations in the formula are meaningless
- FPs should probably be viewed as a vector of several aspects of functionality
 - not a single number!

Feature Points



- Function points were originally designed to be applied to business information systems.
- Extensions have been suggested called feature points which may enable this measure to be applied to other software engineering applications.
- Developed in 1986 by Software Productivity Research, Inc.

Feature Points



- Feature points accommodate applications in which the algorithmic complexity is high such as real-time and embedded software.
- The counts generated in FPs appear to be misleading for software that is high in algorithmic complexity, but sparse in inputs and outputs.

Feature Points



- Feature points are calculated similarly to function points with the addition of an additional software characteristic, algorithms, which is given a weight of 3.

What is an Algorithm?



- An algorithm is a bounded computational problem such as inverting a matrix.
- The Feature Point treatment of algorithms assumes a range of perhaps 10 to 1 for algorithmic difficulty.
 - Algorithms requiring only basic arithmetic operations or a few simple rules are assigned a minimum value of 1.

What is an Algorithm?



- Algorithms requiring complex equations, matrix operations, and difficult mathematical and logical processing could be assigned a weight of 10.
- The default weight for a normal algorithm using ordinary mathematics would be 3.

What is Algorithmic Complexity?



- There is no available taxonomy for classifying algorithms other than purely ad hoc methods.
- The basis for the Feature Points weights for algorithms is:
 - The number of calculation steps or rules required by the algorithm;
 - The number of factors or data elements required by the algorithm.

Feature Points



- Feature points are calculated using:

• Number of user inputs	4
• Number of user outputs	5
• Number of user inquiries	4
• Number of files	7
• Number of external interfaces	7
• Algorithms	3

Feature Points



- The Feature Point method reduces the empirical weights for number of files from the average value of 10 down to an average value of 7.
 - This reflects the somewhat reduced significance of files for systems software as compared to information systems.
- The sum of these values is used in the function point calculation to calculate the feature points.

Function and Feature Points



- Both function points and feature points represent the functionality or utility of a program.
- For conventional software and information systems they produce the same results.
- For complex systems, feature points often produce counts which are 20 to 35 percent higher than function points.

Metric Comparisons



- The relationship between LOC and FP depend on the programming language being used.
- Rough estimates for the number of lines of coded needed for one function point reveal the following:

LOC and FP



Language	LOC/FP (Average)
Assembly language	300
COBOL	100
FORTRAN	100
Pascal	90
ADA	70
OO languages	30
4GLs	20
Code generators	15

Productivity Measures



- The FP and LOC measures should not be used for comparison between different developers.
- The use of LOC/person-month or FP/person-month are too easily influenced by other factors to make them appropriate for comparing productivity between different developers.

Productivity Measures



- Factors which can influence software productivity include
 - size and expertise of the development organization
 - complexity of the problem
 - analysis and design techniques
 - programming language
 - reliability of the computer system
 - availability of hardware and software tools

Object-Oriented Metrics



Object-oriented metrics measure the structure or behaviour of an object-oriented system.

References



- Applying and Interpreting Object Oriented Metrics by Dr. Linda H. Rosenberg
 - http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html
 - [http://ourworld.compuserve.com/homepages/qualazur/\\$swmesu2.htm](http://ourworld.compuserve.com/homepages/qualazur/$swmesu2.htm)
- Object-Oriented Metrics
 - <http://irb.cs.uni-magdeburg.de/sw-eng/us/oo/morris.shtml>

Review of Object Oriented Design



- The system is viewed as a collection of objects rather than as functions with messages passed from object to object.
 - Each object has its own set of associated operations.
- The new (?) approach to software design.

Review of Object Oriented Design



- Object-oriented design is based on the idea of information hiding and modularization of both data and processing which was first put forward by Parnas (1972).
- Best when neither data structure nor processing operations are well defined ahead of time.

Three Pillars of Object Oriented Design



- **Abstraction**
 - identifies common operations between different classes
 - this allows classes to be used interchangeably
 - allows methods relying on one of the interchangeable classes to be applied to all
 - fewer protocol specifications

Three Pillars of Object Oriented Design



- **Information Hiding**
 - details of implementation are hidden
 - details of implementation can be changed without affecting other components
 - ease of maintenance

Three Pillars of Object Oriented Design



- **Modularity**
 - data and processes are packaged together
 - easy re-use
 - breaks down problem

Why Object-Oriented Metrics?



- Some researchers contend that traditional metrics are inappropriate for object oriented systems.
 - But there are valid reasons for applying traditional metrics if possible.
 - The traditional metrics have been widely used.
 - They are well understood.
 - Their relationships to software quality attributes have been validated.

The Use of Traditional Metrics



- Some traditional metrics can be adapted to work within the O-O domain:
 - LOC
 - Percentage of comments
 - Cyclomatic complexity
- These metrics are applied within methods.

Cyclomatic Complexity



- Cyclomatic complexity is used to evaluate the complexity of an algorithm in a method.
- It is a count of the number of test cases that are needed to test the method comprehensively.

Cyclomatic Complexity



- For a sequence where there is only one path, no choices or option, only one test case is needed.
- An IF loop has two choices: if the condition is true, one path is tested; if the condition is false, an alternative path is tested.

New Object-Oriented Metrics



- Weighted methods per class (WMC)
- Response for a class (RFC)
- Lack of cohesion of methods (LCOM)
- Coupling between objects (CBO)
- Depth of inheritance tree (DIT)
- Number of children (NOC)

Weighted Methods per Class (WMC)



- WMC is a count of the methods implemented within a class or the sum of the complexities of the methods.
 - method complexity is measured by cyclomatic complexity
- The number of methods and their complexity is a predictor of how much time and effort is required to develop and maintain the class.

Weighted Methods per Class (WMC)



- The larger the number of methods in a class, the greater the potential impact on children; children inherit all of the methods defined in the parent class.
- Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Response for a Class (RFC)



- The RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class.
 - This includes all methods accessible within the class hierarchy.

Response for a Class (RFC)



- This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes.
- The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class.

Response for a Class (RFC)



- If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester.

Lack of Cohesion (LCOM)



- Lack of Cohesion (LCOM) measures the dissimilarity of methods in a class by instance variable or attributes.
- A highly cohesive module should stand alone.
 - High cohesion indicates
 - good class subdivision
 - simplicity and high reusability
 - good class subdivision

Lack of Cohesion (LCOM)



- Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.
- Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion.

Coupling Between Object Classes (CBO)



- CBO is a count of the number of other classes to which a class is coupled.
 - measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends
- Excessive coupling is detrimental to modular design and prevents reuse.
 - The more independent a class is, the easier it is reuse in another application.

Coupling Between Object Classes (CBO)



- The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult.
- Strong coupling complicates a system since a class is harder to understand, change or correct.

Coupling Between Object Classes (CBO)



- Complexity can be reduced by designing systems with the weakest possible coupling between classes.
 - This improves modularity and promotes encapsulation.

Depth of Inheritance Tree (DIT)



- The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes.
- The deeper a class is within the hierarchy, the greater the number methods it is likely to inherit making it more complex to predict its behavior.

Depth of Inheritance Tree (DIT)



- Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods.

Number of Children (NOC)



- The number of children is the number of immediate subclasses subordinate to a class in the hierarchy.
- If a class has a large number of children, it may require more testing of the methods of that class, thus increasing the testing time.

Number of Children (NOC)



- NOC is an indicator of the potential influence a class can have on the system.
 - The greater the number of children, the greater the likelihood of improper abstraction of the parent.
 - But the greater the number of children, the greater the reuse since inheritance is a form of reuse.