Software Metrics (PA1407)

Lecture 5

Internal Product Attributes: Size and Structure

Software Size

Software size

- Each product of software development is expressed in a concrete form and can be treated in a manner similar to physical entities.
  - Software products can be described in terms of their size
- Size is a very useful internal product attribute. It is used in predicting/ measuring attributes such as effort, productivity, defect density, cost etc.
  - However, Size alone can not directly indicate these external attributes.
- We use our intuition about the size of things in the physical world to develop measures of the size of software entities.
  - Analogy with measures of human size

Software size

- Properties for valid measures of software size
  - Non-negativity
  - Null value
  - Additivity
- Req. specs size can predict the size and complexity of a design
- Design size can predict code size.
- Design and code sizes can determine the required effort for testing, as well as the effort required to add features.
- Size measures should satisfy the representation condition, discussed earlier with representation theory of measurement.

Code Size

- The most commonly used measure of source code program length is the number of lines of code (LOC).
  - *NCLOC:* non-commented source line of code or effective lines of code (ELOC).
  - *CLOC:* commented source line of code.
  - By measuring NCLOC and CLOC separately we can define:
    - *total length (LOC) = NCLOC + CLOC*
- The ratio: *CLOC/LOC* measures the density of comments in a program.

Code Size

- We must take great care to clarify what we are counting and how we are counting it. It should be clarified how following are handled:
  - Blank lines
  - Comment lines
  - Data declarations
  - And lines that contain several separate instructions, for example
- Definition of code size is influenced by its intended use
  - Some companies use size to compare different projects
  - Others measure size only within a project team.

Code size

- Alternative code size measures
  - Source memory size: Measuring length in terms of number of bytes of computer storage required for the program text. Excludes library code.
  - Char size: Measuring length in terms of number of characters (CHAR) in program text.
  - Object memory size: Measuring length in terms of an object (executable or binary) file. Includes library code.
- Dealing with non-textual or external code
  - Code generators, visual programming tools and externally constructed components (e.g. COTS)
  - The LOC size measurement needs to be replaced by some other size measures, such as object points, reused portion of code etc.

Design size

- Design size can be measured in a manner similar to the used to measure code size
  - Count design elements rather than LOCs
- Design elements
  - At lowest level of abstraction: Number of procedures and arguments from APIs of a system.
  - At higher level: Number of packages and subsystems
- OO systems
  - Packages: number of packages/sub-packages, number of classes, interfaces or absract classes
  - Design patterns
  - Classes, interfaces or abstract classes
  - Methods or operations

Requirement specifications size

- Requirements specifications document can consist of a mixture of text and diagrams.
  - It is difficult to generate a single size measure
- Number of pages measures length for an arbitrary type of requirements documentation, and is frequently used
- Other options?
  - Number of use cases or requirements or features, number of actors or goals
    - Categories of these into simple, average or complex

Functional size measures

- Many software engineers argue that length is misleading, and that the amount of functionality inherent in a product paints a better picture of product size.
- Effort and duration estimates from early development products often prefer to estimate functionality rather than physical size
- There have been several serious attempts to measure functionality of software products
  - Function points
  - Object points
  - Use case points
- In agile software development, concept of story points is used now a days.

Function points

- First developed by Albrecht (1979~1983) who suggested a productivity measurement approach called the **Function Point (FP)** method.
- Function Point (FP) is a weighted measure of software functionality based upon the *system specification*.
- FP is computed in two steps:
  - Calculating ***Unadjusted Function point Count*** **(*UFC*)**.
  - Multiplying the UFC by a ***Technical Complexity Factor (TCF)***
  - The final (adjusted) Function Point is: $FP = UFC \times TCF$

Function points

External inputs: those items provided by the user that describe distinct application-oriented data (such as file names and menu selections). These items do not include inquiries, which are counted separately.

External outputs: those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these).

External inquiries: interactive inputs requiring a response.

External files: machine-readable interfaces to other systems.

Internal files: logical master files in the system.

Function Points

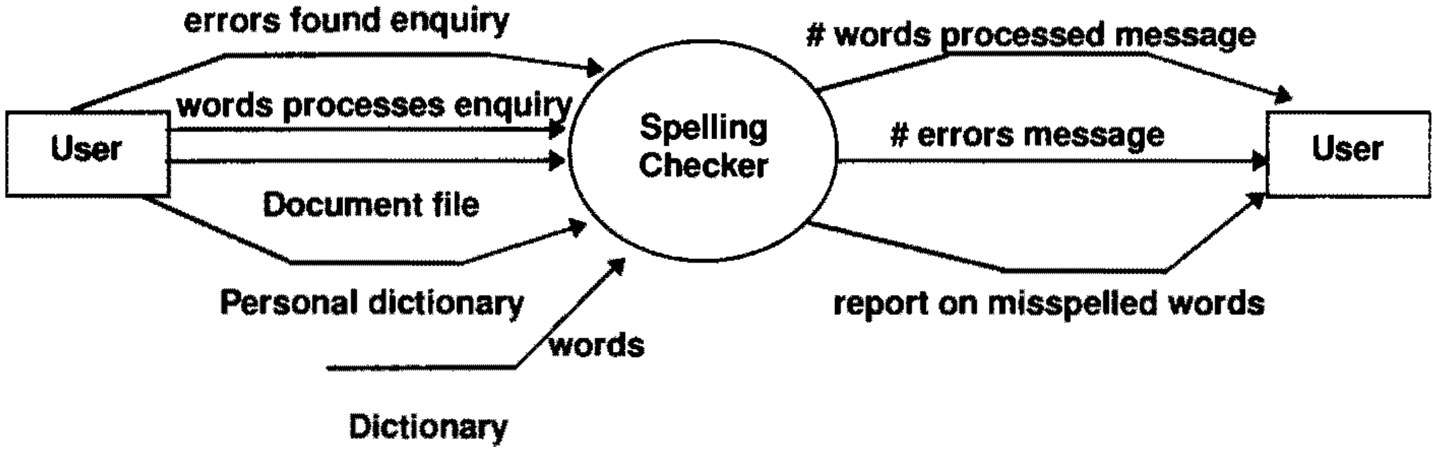
Complexity weights

Item	Simple	Weighting factor Average	Complex
External inputs	3	4	6
External outputs	4	5	7
External inquiries	3	4	6
External files	7	10	15
Internal files	5	7	10

Technical complexity factors:
Each factor is rated from 0 to 5, where 0 means it is not relevant and 5 means it is essential.

F_1 Reliable back-up and recovery	F_2 Data communications
F_3 Distributed functions	F_4 Performance
F_5 Heavily used configuration	F_6 Online data entry
F_7 Operational ease	F_8 Online update
F_9 Complex interface	F_{10} Complex processing
F_{11} Reusability	F_{12} Installation ease
F_{13} Multiple sites	F_{14} Facilitate change

FP example: Spell checker



A = # external inputs = 2, B =# external outputs = 3, C = # inquiries = 2,
D = # external files = 2, E = # internal files = 1

FP example: Spell checker

- Compute UFC using these counts and relevant weighting
- Next compute TCF
  - Assign weights to each component factor
  - Compute TCF using formula:

$$\mathrm{TCF} = 0.65 + 0.01\sum_{i=1}^{14} F_i$$

- Finally, FP = UFC x TCF
- Studies have attempted to relate LOC and FR metrics (Jones, 96)
  - http://www.qsm.com/resources/function-point-languages-table

FP limitations

- Problem with subjectivity in the technology factor
- Problem with double counting
- Problem with counterintuitive values
- Problems with accuracy
- Problems with measurement theory

Applications of size measures

- Using size to normalize other measurements
  - KLOC are often used to normalize many measures e.g. errors, faults, cost
- Size based reuse measurement
  - Measurement of size must also include some method of counting reused products.
  - It is difficult to define formally what is meant by reused code.
  - Extent of reuse is defined as follows
    - Reused verbatim: code in the unit was used without any changes
    - Slightly modified: fewer than 25% of the LOCs in the unit were modified
    - Extensively modified: 25% or more of the LOCs were modified
    - New: none of the code comes from a previously developed unit.
- Size based software testing measurement
  - Size of test suite in terms of SLOC of testing code or number of test cases

Problem/Solution size and complexity

- Size of the problem -> reflected in specifications -> used in functional size measures
- A problem can have more than one solution
  - Complexity or size of a solution can be considered as a separate component of size
- Complexity of a problem can be informally defined as the amount of resources required for an optimal solution to the problem
- Complexity of solution can then be regarded in terms of resources needed to implement a particular solution.
- Solution complexity have at least two aspects (both are size measures)
  - Time complexity
  - Space complexity

Software Structure

Aspects of structural measures

- We would like to assume that a large module takes longer to implement and test than a small one
  - However, we know such an assumption is unrealistic
  - Product structure also contributes to the development and maintenance effort.
- A software module or design can be viewed from several perspectives
  - Abstraction level
  - The way the module or design is described (syntax and semantics)
  - The specific attribute to be measured.
- Structure can be viewed from two perspectives
  - Control flow structure
  - Data flow structure

Product structure

- Structural complexity
  - Captures the complicatedness of the connections between elements in a system model.
- Coupling
  - It is an attribute of an individual module and depends on a module's links to and from elements that are external to the module
- Cohesion
  - Cohesion is an attribute of an individual module and depends on the extent that related elements are contained in a module.
  - A module with many connections between its internal elements will have greater cohesion than a module that contains unrelated elements

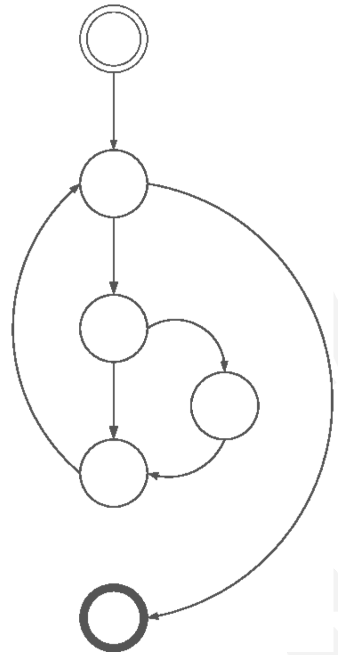
Mcabe's cyclomatic complexity measure

- A program's complexity can be measured by the cyclomatic number of the program flow graph.
- For a program with the program flow graph *G*, the cyclomatic complexity *v(G)* is measured as:
  1. $v(G) = e - n + 2p$
     - *e* : number of edges, *n* : number of nodes representing block of sequence code, *p* : number of connected components
  2. $v(G) = 1 + d$
     - *d* : number of predicate nodes (i.e., nodes with out-degree other than 1)

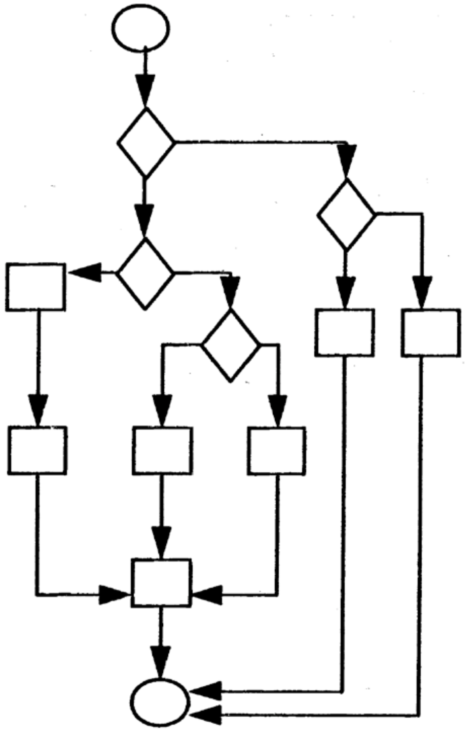
Cyclomatic complexity: example

- $v(G) = e - n + 2p = 7 - 6 + 2 \times 1$
- $v(G) = 3$

- *Or*
- $v(G) = 1 + d = 1 + 2 = 3$



Another example



$v(G) = 16 – 13 + 2 = 5$

or

$v(G) = 4 + 1 = 5$

Mcabe's cyclomatic complexity measure

- A useful indicator of how difficult a program or module will to test and maintain
- Limitations
  - Requires complete design or code visibility
  - Can only be used at the component level
  - Two programs having same CC number may need different programming effort.

Object Oriented structural attributes and measures

- OO concepts
  - Class and object
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

Coupling properties

- Non-negativity: Module coupling cannot be negative
- Null Value: A module with no links to external elements has zero coupling
- Monotonicity: Adding links between module does not decrease coupling
- Merging modules: Merging two modules creates a new module that has coupling that is at most the sum of the coupling of the two modules.
- Disjoint module additivity: Merging disjoint modules without links between them creates a new module with coupling that is the sum of the coupling of the original modules.

Coupling in OO systems

- Most of the coupling measures quantify between classes, not objects.
- Type of associations in OO systems?
- Object oriented metrics suite (Chidamber and Kemerer 1994)
  - CBO (Coupling Between Object classes), RFC (Response For Class)
- MPC (Message Passing Coupling) was introduced by Li and Henry. The MPC value of a class is a count of the number of static invocations of methods that are external to the class.
- Package level coupling
  - Afferent coupling (Ca is really a fan-out of a package)
    - The No. of classes from other packages that depends on the classes within the subject package. Only class relationships are counted, such as inheritance and association.
  - Efferent coupling (Ce is really a fan-in of a package)
    - The No. of classes in other packages that the classes in the subject package depend on, via class relationships.
    - Instability metric = Ce/(Ca + Ce)

Cohesion properties

- Non-negativity and normalization: Module cohesion is normalized so that it is between zero and one.
- Null Value: A module whose elements have no links between them has zero cohesion
- Monotonicity: Adding links between elements in a module cannot decrease cohesion of the module
- Merging modules: Merging two unrelated modules creates a new module with a maximum cohesion no greater than that of the original module with the greatest cohesion

Cohesion in OO systems

- Class cohesion is an intra-class attribute
  - It reflects the degree to which the parts of a class – methods, method calls, fields, and attributes belong together.
- Example class cohesion measures
  - Lack of Cohesion Metric (LCOM)
    - How closely are the local methods related to the local instance variables in the class
    - It is an inverse cohesion measure i.e. higher value implies lower cohesion.
  - Tight class cohesion (TCC) and loose class cohesion (LCC)
    - Methods have **direct connection** if they read/write to same instance variables; and **indirect connection** if one method uses one or more instance variables directly, while the other uses them indirectly by calling another method.
    - TCC is based on the relative number of direct connections: TCC(C) = NDC(C)/NP(C)
      - NDC(C) is no of direct connections; NP(C) is maximum no. of possible connections
    - LCC is based on no of direct and indirect connections: LCC(C) = (NDC(C) + NIC(C)) /NP(C)
      - NIC(C) is no. of indirect connections.

Cohesion in OO systems

- Package Cohesion
  - It concerns the degree to which the elements of a package – classes and interfaces – belong together.
  - Robert C. Martin defines package cohesion as
    - Package relational cohesion RC(P) = (R(P) + 1) / N (P)
      - R(P) is the No. of relations between classes and interfaces in a package; N(P) is the No. classes and interfaces in a package.
      - A 1 is added so that a package with one class will have H = 1.
    - Revised package relational cohesion is defined as (R(P) + 1) / NP(P)
      - NP(P) is the no. of possible relations between classes and interfaces in a package.
  - This measure may not reflect the desired structuring for a package.
    - In structuring a package, we may really seek logical cohesion.

Acknowledgement

- Lecture notes are prepared from following sources:
  - T1: Software Metrics - A Rigorous & Practical Approach, 2nd edition, Authors: N. E. Fenton, S. L. Pfleeger, Publishers: International Thomson Computer Press, 1996, ISBN: 1-85032-275-9.