

Chapter 23

Software Metrics and Analytics

McCall's Triangle of Quality

Maintainability

Flexibility

Testability

Portability

Reusability

Interoperability

PRODUCT REVISION

PRODUCT TRANSITION

PRODUCT OPERATION

Correctness

Usability

Efficiency

Reliability

Integrity

ISO 9126 Quality Factors

- Functionality, reliability, usability, efficiency, maintainability, portability

Measures, Metrics and Indicators

- A SW engineer collects **measures** and develops **metrics** so that **indicators** will be obtained
 - A **measure** provides a **quantitative** indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
 - The IEEE defines a **metric** as “a quantitative measure of the degree to which a system, component, or process possesses a given **attribute**. ”
 - IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)
 - An **indicator** is a metric or combination of metrics that provide **insight** into the software process, a software project, or the product itself
- Ex. Moonzoo Kim
 - Measure: height=170cm, weight=65 kg
 - Metric: fat metric= 0.38 (=weight/height)
 - Indicator: normal health condition (since fat metric < 0.5)

Measurement Principles

- The **objectives** of measurement should be established before data collection begins
 - Ex. It is useless for black-box testers to measure a # of words in a C file.
 - Ex. It is useful for C compiler developers to measure a # of words in a C file.
- Each technical metric should be defined in an **unambiguous** manner
 - Ex. For measuring a total line number of a C program
 - Including comments? Including empty lines?
- Metrics should be derived based on a **theory** that is valid for the domain of application
 - Metrics for design should draw upon **basic design concepts and principles** and attempt to provide an **indication** of the presence of a desirable attribute
 - Metrics should be **tailored** to best accommodate specific products and processes

Measurement Process

■ *Formulation.*

The derivation of software measures and metrics appropriate for the representation of the software that is being considered.

■ *Collection*

The mechanism used to accumulate data required to derive the formulated metrics.

■ *Analysis.*

The computation of metrics and the application of mathematical tools.

■ *Interpretation.*

The evaluation of metrics results in an effort to gain insight into the quality of the representation.

■ *Feedback.*

Recommendations derived from the interpretation of product metrics transmitted to the software team.

■ *Example of Formulation*

To check whether a give software is *hot-spotted* (i.e. has intensive loops)

■ *Example of Collection*

Instrument a source program/binary to count how many time a given statement is executed in one second

■ *Example of Analysis.*

Using Excel/MatLab to get average numbers of executions of statements

■ *Example of Interpretation.*

If there exist statements which were executed more than 10^8 , on a 3 Ghz machine, then the program is hot-spotted

■ *Example of Feedback.*

Try to optimize those hot-spotted statements. Or those hot-spotted statement might have logical flaws

Goal-Oriented Software Measurement

- The Goal/Question/Metric Paradigm
 - establish an explicit measurement *goal*
 - define a set of *questions* that must be answered to achieve the goal
 - identify well-formulated *metrics* that help to answer these questions.
- Goal definition template
 - **Analyze**
{the name of activity or attribute to be measured}
 - **for the purpose of**
{the overall objective of the analysis}
 - **with respect to**
{the aspect of the activity or attribute that is considered}
 - **from the viewpoint of**
{the people who have an interest in the measurement}
 - **in the context of**
{the environment in which the measurement takes place}.

Ex> Goal definition for SafeHome

- **Analyze** the Safehome SW architecture
- **for the purpose of** evaluating architectural components
- **with respect to** the ability to make Safehome more extensible
- **from the viewpoint of** the SW engineers performing the work
- **in the context of** enhancement over the next 3 years
- Questions
 - Q1: Are architectural components characterized in a manner that compartmentalizes function and related data?
 - Answer: 0 ... 10
 - Q2: Is the complexity of each component within bounds that will facilitate modification and extension?
 - Answer: 0 ... 1

Metrics Attributes

- *Simple and computable.*

It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time

- *Empirically and intuitively persuasive.*

The metric should satisfy the engineer's intuitive notions about the product attribute under consideration

- *Consistent and objective.*

The metric should always yield results that are unambiguous.

- *Consistent in its use of units and dimensions.*

The mathematical computation of the metric should use measures that do not lead to bizarre combinations of unit.

ex. MZ measure of a software complexity: kg x m⁴

- *An effective mechanism for quality feedback.*

That is, the metric should provide a software engineer with information that can lead to a higher quality end product

Collection and Analysis Principles

- Whenever possible, data collection and analysis should be **automated**
- Valid statistical techniques should be applied to establish relationship between internal product attributes and external quality characteristics
- Interpretative guidelines and recommendations should be established for each metric
 - Ex. Fat metric greater than 0.5 indicates obesity. A person who has more than 0.7 fat metric should consult a doctor.

Overview of Ch23. Product Metrics

- 23.3.1 Metrics for the Requirement (Analysis) Model
 - Function point metrics
- 23.3.2 Design Metrics for Conventional Software
- 23.3.3 Design Metrics for OO Software
 - Class-oriented metrics
 - Component-level design metrics
 - Operation oriented metrics
- 23.3.4 User Interface Design Metrics
- 23.3.5 Metrics for Source Code
- 23.4 Metrics for Testing
- 23.5 Metrics for Maintenance

Metrics for the Analysis Model

- These metrics examine the analysis model with the intent of predicting the “**size**” of the resultant system
- Size can be an indicator of **design complexity**
- Size can be an indicator of **increased coding, integration, and testing efforts**
- Example
 - Function-based metrics
 - Metrics for specification quality

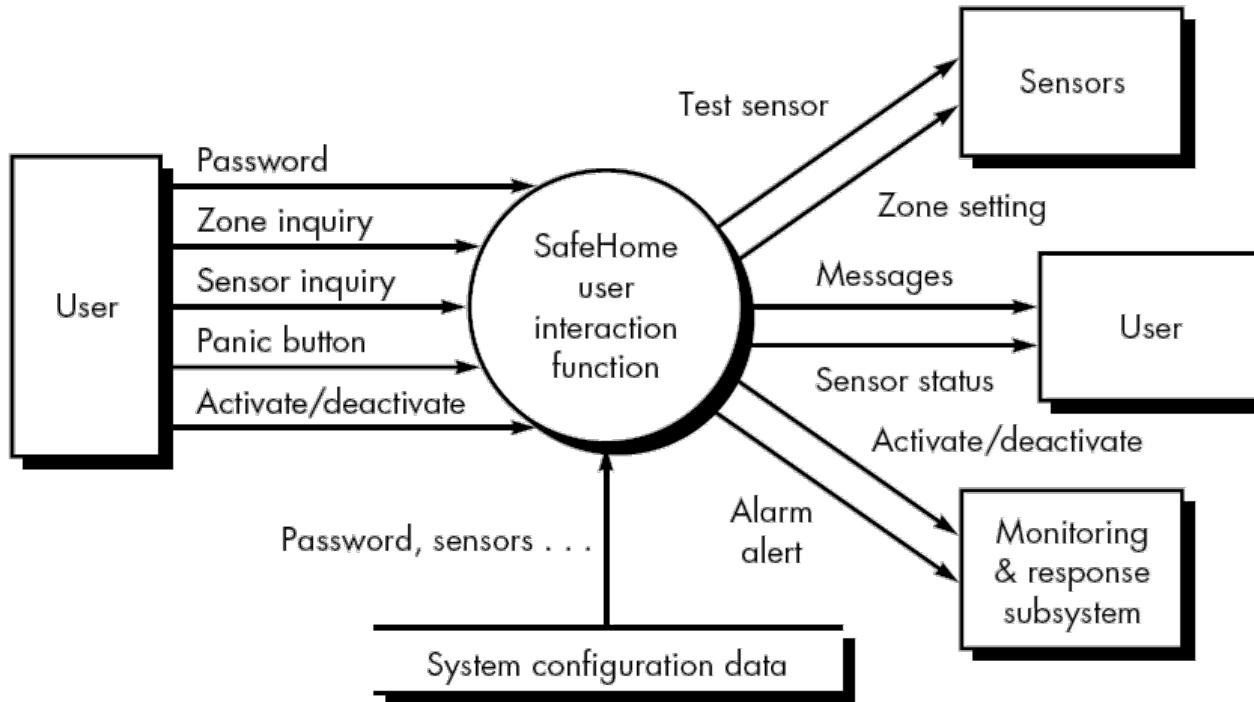
Function-Based Metrics

- The ***function point metric (FP)***, first proposed by Albrecht [ALB79], can be used effectively as a means for measuring the functionality delivered by a system.
- Function points are derived using an empirical relationship based on countable (direct) measures of software's **information domain** and assessments of **software complexity**
- Information domain values are defined in the following manner:
 - # of external inputs (EIs)
 - often used to update internal logical files
 - # of external outputs (EOs)
 - # of external inquiries (EQs)
 - # of internal logical files (ILFs)
 - # of external interface files (EIFs)

Function Points

Information Domain Value	Count	Weighting factor			=	
		simple	average	complex		
External Inputs (Els)	<input type="text"/>	3	3	4	6	<input type="text"/>
External Outputs (EOss)	<input type="text"/>	3	4	5	7	<input type="text"/>
External Inquiries (EQs)	<input type="text"/>	3	3	4	6	<input type="text"/>
Internal Logical Files (ILFs)	<input type="text"/>	3	7	10	15	<input type="text"/>
External Interface Files (EIFs)	<input type="text"/>	3	5	7	10	<input type="text"/>
Count total						<input type="text"/>

FP = count total x (0.65 + 0.01 x $\sum(F_i)$)
where F_i 's are value adjustment factors based on
responses to the 14 questions



Weighting Factor

Measurement parameter	Count	Simple	Average	Complex	=	
Number of user inputs	3	x	3	4	6	= 9
Number of user outputs	2	x	4	5	7	= 8
Number of user inquiries	2	x	3	4	6	= 6
Number of files	1	x	7	10	15	= 7
Number of external interfaces	4	x	5	7	10	= 20
Count total						50

Value Adjustment Factors (F_i)

- Following questions should be answered using a scale that ranges from 0 (not important) to 5 (absolutely essential)
 - Does the system require reliable backup and recovery?
 - Are specialized data communications required to transfer information to or from the application?
 - Are there distributed processing functions?
 - Is performance critical?
 - Will the system run in an existing, heavily utilized operational environment?
 - 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?

Usage of Function Points

- Assume that
 - past data indicates that one FP translates into 60 lines of code
 - 12 FPs are produced for each person-month of effort
 - Past projects have found an average of 3 errors per FP during analysis and design reviews
 - 4 errors per FP during unit and integration testing
- These data can help SW engineers assess the completeness of their review and testing activities.
- Suppose that Safehome has 56 FPs
 - $56 = 50 \times [0.65 + 0.01 \times \sum(F_i)] (= 46)$
- Safehome will be
 - Expected size: $60 \text{ lines} * 56 = 3360 \text{ lines}$
 - Expected required man-month: $1/12 \text{ MM} * 56 = 4.7 \text{ MM}$
 - Total analysis/design errors expected: $3 * 56 = 168 \text{ errors}$
 - Total testing errors expected: $4 * 56 = 224 \text{ errors}$

Metrics for the Design Model

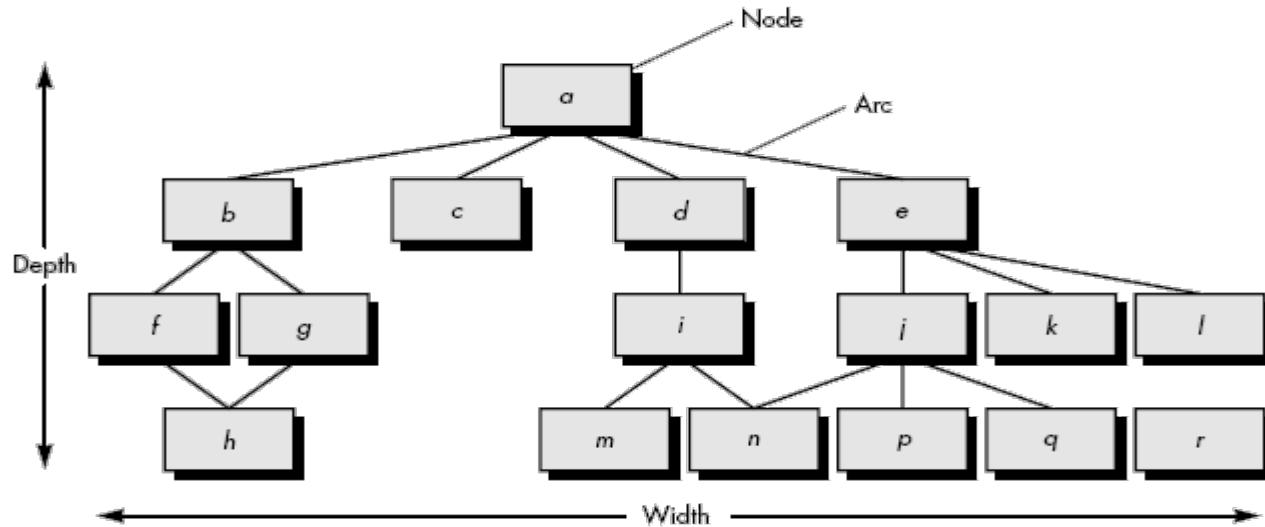
- The design of engineering products (i.e. a new aircraft, a new computer chip, or a new building) is conducted with well-defined design metrics for various design qualities
 - Ex 1. Quality does matter, see AMD's success in 2000~2006.
 - Ex 2. Pentium X should have
 - Heat dispense ratio < 100 Kcal/s
 - Should operate 99.99% time correctly at 10 Ghz
 - Should consume less than 100 watts/h electric power
- The design of complex software, however, often proceeds with virtually no metric measurement
 - Although design metric is **not** perfect, design without metric is **not** acceptable.

Architectural Design Metrics

- Architectural design metrics put emphasis on the effectiveness of modules or components within the architecture
 - These metrics are “black box”
- Architectural design metrics
 - Structural complexity of a module $m = (\# \text{ of fan-out of module } m)^2$
 - Fan-out is the number of modules immediately subordinate to the module
 - i.e. the # of modules that are directly invoked by the module
 - Data complexity = (# of input & output variables)/ (fan-out+1)
 - System complexity = structural complexity + data complexity

Morphology Metrics

- **Morphology metrics:** a function of the number of modules and the number of interfaces between modules
 - Size = $n + a$
 - Depth = the longest path from the root node to a leaf node
 - Width = maximum # of nodes at any one level of the architecture
 - Arc-to-node ratio



Design Structure Quality Index (DSQI)

- Developed by U.S. Air Force Systems Command
- It rates how well a SW design is 1) structured, 2) modularized, and 3) maintainable, producing a score from 0 (worst) to 1 (best) from the following factors:

$$DSQI = 1 - \frac{\text{Design Defects}}{\text{Total Design Elements}}$$

Evaluation Factor	Meaning
# of modules	Design modularity and separation of concerns
Interfaces between modules	Dependency complexity
Data structures used	Data clarity and organization
Global/shared data usage	Risk of side-effects / tight coupling
Component reuse	Efficiency and maintainability
Known design flaws	Problems found during review

Metrics for OO Design-I

- Whitmire [WHI97] describes nine distinct and measurable characteristics of an OO design:
 - **Size**
 - Size is defined in terms of the following four views:
 - Population: a static count of OO entities such as classes
 - Volume: a dynamic count of OO entities such as objects
 - Length: a measure of a chain of interconnected design elements
 - Functionality: value delivered to the customer
 - **Complexity**
 - How classes of an OO design are interrelated to one another
 - **Coupling**
 - The physical connections between elements of the OO design
 - The # of collaborations between classes
 - **Sufficiency**
 - “the degree to which an abstraction possesses the features required of it, ... from the point of view of the current application.”
 - Whether the abstraction (class) possesses the features required of it

Metrics for OO Design-II

- **Completeness**
 - An indirect implication about the degree to which the abstraction or design component can be reused
- **Cohesion**
 - The degree to which all operations working together to achieve a single, well-defined purpose
- **Primitiveness**
 - Applied to both operations and classes, the degree to which an operation is atomic
- **Similarity**
 - The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose
- **Volatility**
 - Measures the likelihood that a change will occur

Distinguishing Characteristics

Berard [BER95] argues that the following characteristics require that special OO metrics be developed:

- **Encapsulation**
the packaging of data and processing
- **Information hiding**
the way in which information about operational details is hidden by a secure interface
- **Inheritance**
the manner in which the responsibilities of one class are propagated to another
- **Abstraction**
the mechanism that allows a design to focus on essential details
- **Localization**
the way in which information is concentrated in a program

Class-Oriented Metrics

Proposed by Chidamber and Kemerer (CK metrics suite):

■ Weighted Methods per Class (WMC)

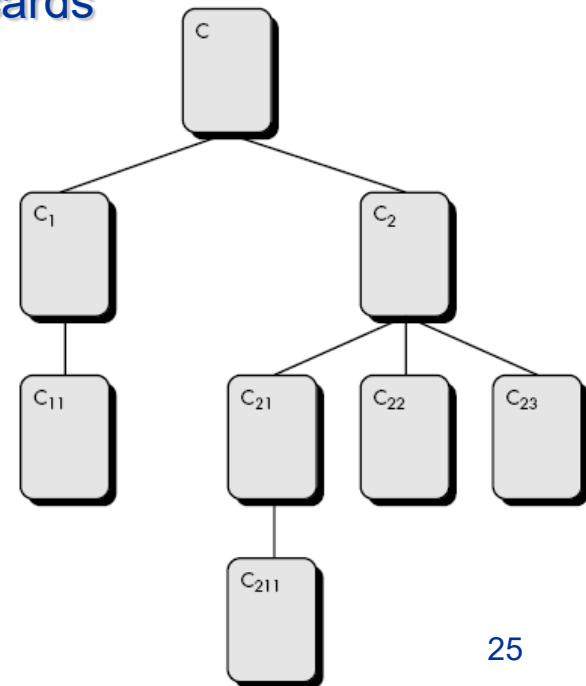
- Weighted methods per class $\sum C_i$ where C_i is a normalized complexity 0~1 for method i (e.g., cyclometric complexity)
- The # of methods and their complexity are reasonable indicators of the amount of effort required to implement and test a class
- As the # of methods grows for a given class, it is likely to become more application specific -> less reusability
- Counting the # of methods is not trivial

■ Depth of the inheritance tree (DIT)

- As DIT grow, potential difficulties when attempting to predict the behavior of a class

Class-Oriented Metrics

- Number of children/subclasses (NOC)
 - As NOC grows, more reuse, but the abstraction of the parent class is diluted
 - As NOC grows, the amount of testing will also increase
- Coupling between object classes (CBO)
 - CBO is the # of collaborations listed on CRC index cards
 - As CBO increases, reusability decreases
- Response for a class (RFC)
 - A set of methods that can be executed in response to a request
 - As RFC increases, test sequence grows
- Lack of cohesion in methods (LCOM)
 - A # of methods that access same attributes



Applying CK Metrics

- **The scene:**
 - Vinod's cubicle.
- **The players:**
 - **Vinod, Jamie, Shakira, Ed** members of the *SafeHome* software engineering team, who are continuing work on component-level design and test case design.
- **The conversation:**
- **Vinod:** Did you guys get a chance to read the description of the CK metrics suite I sent you on Wednesday and make those measurements?
- **Shakira:** Wasn't too complicated. I went back to my UML class and sequence diagrams, like you suggested, and got rough counts for DIT, RFC, and LCOM. I couldn't find the CRC model, so I didn't count CBO.
- **Jamie (smiling):** You couldn't find the CRC model because I had it.
- **Shakira:** That's what I love about this team, superb communication.
- **Vinod:** I did my counts . . . did you guys develop numbers for the CK metrics?

- (Jamie and Ed nod in the affirmative.)
- **Jamie:** Since I had the CRC cards, I took a look at CBO, and it looked pretty uniform across most of the classes. There was one exception, which I noted.
- **Ed:** There are a few classes where RFC is pretty high, compared with the averages . . . maybe we should take a look at simplifying them.
- **Jamie:** Maybe yes, maybe no. I'm still concerned about time, and I don't want to fix stuff that isn't really broken.
- **Vinod:** I agree with that. Maybe we

should look for classes that have bad numbers in at least two or more of the CK metrics. Kind of two strikes and you're modified.

- **Shakira (looking over Ed's list of classes with high RFC):** Look, see this class? It's got a high LCOM as well as a high RFC. Two strikes?

- **Vinod:** Yeah I think so . . . it'll be difficult to implement because of complexity and difficult to test for the same reason. Probably worth designing two separate classes to achieve the same behavior.

- **Jamie:** You think modifying it'll save us time?

- **Vinod:** Over the long haul, yes.

Class-Oriented Metrics

The MOOD Metrics Suite

- Method inheritance factor (MIF) $MIF = \sum M_i(C_i) / \sum M_a(C_i)$
 - $M_i(C_i)$ = the # of methods inherited (and not overridden) in C_i
 - $M_d(C_i)$ = the # of methods declared in the class C_i
 - $M_a(C_i) = M_d(C_i) + M_i(C_i)$
- Coupling factor $CF = \sum \sum is_client(C_i, C_j) / (T_c^2 - T_c)$
 - $is_client = 1$ if and only if a relationship exists between the client class C_c and C_s ($C_c \neq C_s$)
 - High CF makes trouble to understandability, maintainability and reusability.

Component-Level Design Metrics

- **Cohesion metrics**
 - a function of data objects and the locus of their definition
- **Coupling metrics**
 - a function of input and output parameters, global variables, and modules called
- **Complexity metrics**
 - hundreds have been proposed (e.g., cyclomatic complexity)

Operation-Oriented Metrics

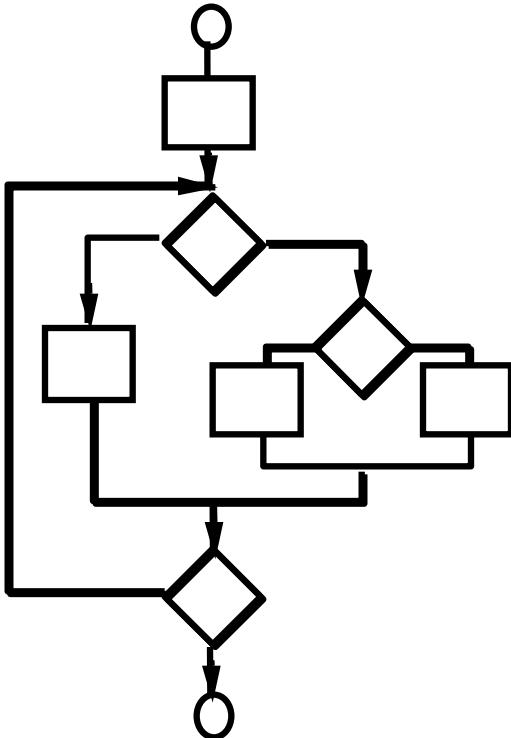
Proposed by Lorenz and Kidd [LOR94]:

- average operation size
of messages sent by the operation
- operation complexity
- average number of parameters per operation

Metrics for Source Code

- **Halstead's Software Science:** a comprehensive collection of metrics based on the number (count and occurrence) of operators and operands within a component or program
 - n_1 : # of distinct operators that appears in a program
 - n_2 : # of distinct operands that appears in a program
 - N_1 : # of operator occurrences
 - N_2 : # of operand occurrences
- Program length $N = n_1 \log_2 n_1 + n_2 \log_2 n_2$
- Program volume $V= N \log_2 (n_1 + n_2)$
- And many more metrics

Cyclometric Complexity



- A quantitative measure of the logical complexity
- Cyclomatic complexity defines the # of independent paths to test for complete statement/branch coverage
 - number of simple decisions + 1
 - number of edge – number of node +2p
 - number of enclosed areas + 1
 - In this case, $V(G) = 4$

Metrics for Testing

- Testing effort can also be estimated using metrics derived from Halstead measures
- Binder [BIN94] suggests a broad array of design metrics that have a direct influence on the “testability” of an OO system.
 - Lack of cohesion in methods (LCOM).
 - Percent public and protected (PAP).
 - Public access to data members (PAD).
 - Number of root classes (NOR).
 - Fan-in (FIN).
 - Number of children (NOC) and depth of the inheritance tree (DIT).

Metrics for Maintenance

■ IEEE Std 982.1-1998 Software Maturity Index (SMI)

- $SMI = [M_t - (F_a + F_c + F_d)]/M_t$
 - M_t = # of modules in the current release
 - F_c = # of modules in the current release that have been changed
 - F_a = # of modules in the current release that have been added
 - F_d = # of modules from the preceding release that were deleted in the current release

