Global Journal of Computer Science and Technology: G Interdisciplinary
Volume 20 Issue 3 Version 1.0 Year 2020
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Design Complexity for Objective Function Points

By Paul Cymerman, Joe Van Dyke & Ian Brown

Abstract- This paper investigates correlating the basic elements of Unified Modeling Language and Cyclomatic Complexity with Function Point Analysis (FPA) principles to develop an automated software functional sizing tool. This concept has been difficult to achieve due to the logical nature of the FPA sizing methodology versus the physical nature of source lines of code (SLOC). In this approach, we examine software complexity from design and maintainability perspectives in order to understand relationships in physical code. Our hypothesis is that this method will "simulate" FPA principles and produce an objective sizing method. This would provide the foundation for an automated tool that scans physical software code to derive "Objective Function Points" (OFPs) functional size measure.

GJCST-G Classification: D.2.9

Strictly as per the compliance and regulations of:

Design Complexity for Objective Function Points

Paul Cymerman [α], Joe Van Dyke [σ] & Ian Brown [ρ]

Abstract- This paper investigates correlating the basic elements of Unified Modeling Language and Cyclomatic Complexity with Function Point Analysis (FPA) principles to develop an automated software functional sizing tool. This concept has been difficult to achieve due to the logical nature of the FPA sizing methodology versus the physical nature of source lines of code (SLOC). In this approach, we examine software complexity from design and maintainability perspectives in order to understand relationships in physical code. Our hypothesis is that this method will "simulate" FPA principles and produce an objective sizing method. This would provide the foundation for an automated tool that scans physical software code to derive "Objective Function Points" (OFPs) functional size measure.

I. Unified Modeling Language Background

We investigated using Unified Modeling Language (UML) [1] to map to Function Points (FPs) [2]. Developed to provide a common language for object- oriented modeling, UML was designed to be extensible in order to satisfy a wide variety of software engineering needs. Like FPs, it was also intended to be independent of any specific programming languages or development methods. [3] Graphical notation represents the UML syntax. UML is defined by the following three categories:

- Static structure diagrams: Describe the structure of a system and include class and object diagrams.
- Behavior diagrams: Describe the behavior /dynamic perspective of a system and include use-case diagrams, interaction diagrams, sequence diagrams, collaborations diagrams, state diagrams and activity diagrams.
- Implementation diagrams: Provide actual source code information including component diagrams and deployment diagrams.

Class diagrams describe the static structure of the model that is objects, classes and relationships between these entities which include generalization and aggregation. They also represent the attributes and operations of the classes.

In order to apply FP concepts in a UML context, we had to translate between the two. To simplify FP terms and definitions into sizing measures that can be easily calculated using a tool, the OFP translation is included in BLUE.

Author α σ: Quaternion Consulting Inc.
e-mails: pcymerman@quaternion-consulting.com, jvandyke@quaternion-consulting.com
Author ρ: Galorath, Inc. e-mail: ibrown@galorath.com

Record Element Type: Most RETs are dependent on a parent – child relationship. In this case, the child information is a superset where a child class/object inherits all attributes and methods of the parent information. In a parent-child structure, there are one- to-many relationships that define the nature of the connection between attributes within entities [4].

RET ~ INHERITANCE

File Type Referenced: Associations between files provide mapping of maintained files by the application [4]

FTR ~ ASSOCIATION

Data Element Type: UML attributes provide a good indication as to what DETs should be counted in FPA [4].

DET ~ ATTRIBUTES

II. What is Cyclomatic Complexity?

Cyclomatic Complexity (CC) is a software metric used as a limiting function for measuring the complexity of routines during program development. When the CC of the module exceeds 10 [5], modules are split into smaller modules.

CC is one measure of complexity in software development. This complexity is specific to the ongoing development of routines during overall program development. McCabe references this as Design Complexity (DC) of the Module. It does not address architectural complexity of software design. That would be called the DC of the architecture. The more interactions between objects and the more associations between classes there are, the higher will be the complexity. Both the abstract level of the class as well as the physical level of the objects are taken into consideration. [6]

The following statements from Richard Seidl captures the following rational behind DC:

"UML Design Complexity metrics can be defined as the relationship of entities to relationships. The size of a set is determined by the number of elements in that set. The complexity of a set is a question of the number of relationships between the elements of that set. The more connections or dependencies there are relative to the number of elements, the greater the complexity." [6]

"The more interactions and associations there are between objects and classes, the greater the dependency of those objects and classes upon one another. This mutual dependency is referred to a

coupling. Classes with a high coupling have greater domain impacts" [6]

III. What is Architecture Design Complexity (DC)?

This DC is a software metric used to understand the Architecture Design – not just for a specific module, but also between modules. This focuses on the Class (a.k.a. Module), Methods (a.k.a. Functions) and Attributes.

A class is a set of objects that have common structure and behavior. A class consists of a collection of states (a.k.a. attributes or properties) and behaviors (a.k.a. methods). A class represents the abstract matrix of an object before it's instantiated, where an object is an instance of a class.

A method is an operation, which can update the value of the certain attributes of an object.

An attribute is an observable property of the objects of a class.

The overall Architecture Design considers the additional relationships:

Association is a relationship between classes which is used to show that instances of classes could be either linked to each other or combined logically or physically through a semantic relationship Inheritance is a form of Association and a feature of object-oriented programming that allows code reusability when a class includes property of another class.

IV. Deriving Design Complexity of the Architecture

The elementary variables in functions above are designated as DET. The functional complexity is estimated as the total number of user-identifiable groups that exists within DETs and is termed as RET in Data Functions and all referenced file types are counted as FTR in Transactions Functions. A corresponding matrix holds the reference function point values for all function types namely the ILF, EIF, EI, EO and EQ, with respect to the range of DET and RET/FTR in each function. The total sum of the high, medium and low count of all operations is the unadjusted function point count.

The goal is to extract the DC from the complexity fundamentally imbedded in these original relationships. This starts with A.J. Albrecht's original Function Point calculations. There are 3 curves, figure 1, that show how the FPs are calculated based on some level of complexity.

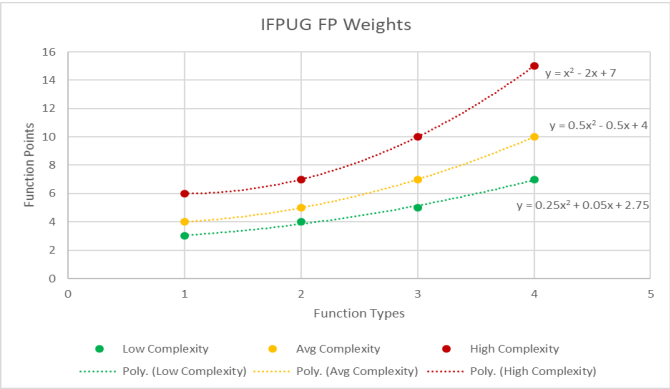

Figure 1: FPA Tables to Curves

Mapping the Function Types to Figure 1, we take the "EI" table and map to the complexity value of "1" on the graph. The "EO and EQ" maps to the complexity value of 2. "EIF" maps to a complexity value of 3 and "ILF" maps to a complexity value of 4.

Hypothesis 1: Since X-axis references function types that have some inherent complexity, assume this complexity is the development complexity that CC tries to capture. We will shortly address the DC that determines the phase shift of the 3 curves.

McCabe used 4 bins for his Cyclomatic Complexities:

1. CC value 1-10
2. CC value 11-20
3. CC value 21-40
4. CC value >40

If we use these bins as our X-axis values, we can determine the appropriate Function Point value.

Thus, the general form equation from the graphs is:

Function Point = "Coefficient A" * CC^2 + "Coefficient B" * CC + "Coefficient C"

Hypothesis 2: Observing the spacing between each of Albrecht's original curves, we can assume that another order of complexity drove these phase shifts. Since the DET, RET and FTR relationships helped build these curves, let's assume that this complexity is the imbedded DC that is used but not specifically referenced. As previously discussed, the DETs, RETs and FTRs are equivalent to ATTRIBUTES, INHERITANCES and ASSOCIATIONS. This would show

how the architecture works together which is the basis for DC. Fitting a quadratic equation:

Function Point = A * CC^2 + B * CC + C

permits DC to be represented by the 3 Coefficients:

A, B, and C.

Next derive the Design Complexity using the values from the 3 curves.

HIGH: $y = x^2 - 2x + 7$

AVG: $y = 0.5x^2 - 0.5x + 4$

LOW: $y = 0.25x^2 + 0.05x + 2.75$

Where x = CC

If we want to model the A Coefficient, we need to look problem in Complexity space where the X-axis is the HIGH, AVG and LOW.

Setting HIGH to 3; AVG to 2; LOW to 1

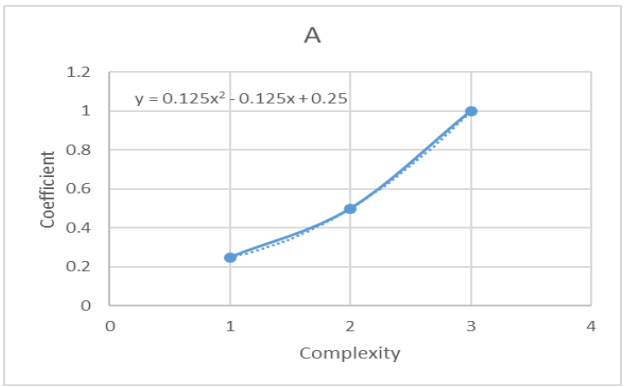

Figure 2: "A" Coefficient by Complexity

Following the same process for Coefficient B, we have:

Setting HIGH to 3; AVG to 2; LOW to 1

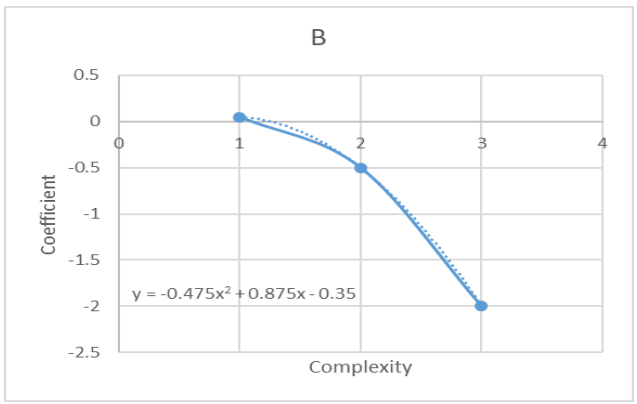

Figure 3: "B" Coefficient by Complexity

Following the same process for Coefficient C, we have:

Setting HIGH to 3; AVG to 2; LOW to 1

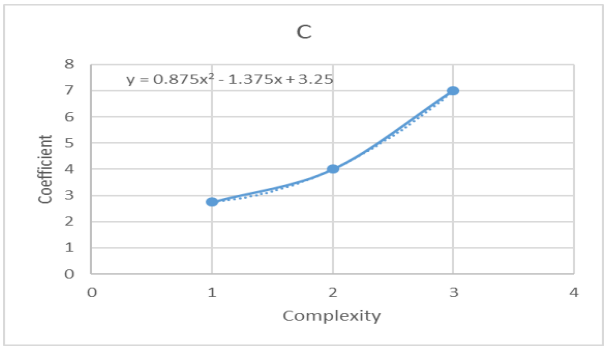

Figure 4: "C" Coefficient by Complexity

The complexity on the X-axis is the DC that we are looking for. We now can calculate the coefficients based on DC.

Coefficient A = 0.125 * DC^2 – 0.125 * DC + 0.25

Coefficient B = -0.475 * DC^2 + 0.875 * DC – 0.35

Coefficient C = 0.875 * DC^2 – 1.375 * DC + 3.25

This now leads to a Function Point equation dependent on CC and DC:

Function Point = (0.125 * DC^2 – 0.125 * DC + 0.25) * CC^2 + (-0.475 * DC^2 + 0.875 * DC – 0.35) * CC + (0.875 * DC^2 – 1.375 * DC + 3.25)

Where DC = 1 for LOW; 2 for AVG; 3 for HIGH

V. Deriving Design Complexity as a Function of Inheritance, Associations and Attributes

Referencing Albrecht's original complexity tables regarding DETs, RETs and FTRs, we can substitute Inheritance for RETs; Associations for FTRs and Attributes for DETs to come up with the following table. To focus on Inheritances, Associations, and Attributes, we are moving from RET, FTR, DET categories to Inheritance, Association, and Attributes categories. For Inheritance and Associations, we need to consider cases where there are values of "0" so we need to adjust the information as follows:

Category	Low	Avg	High
Inheritance	0	1-4	>4
Associations	0-1	2	>2
Attributes	1-19	20-50	>50

The next step is to transform this table into equations. Starting with the Inheritance category, the first row of the table, if we curve fit the values for Inheritance, we will see that the curve, when Inheritance = 0, we intentionally shift the value by 1. Thus, the X-axis is based by Inheritance+1. This technique avoids dealing with a value of 0 which provides a better fit regression curve. When the value on Y-axis is 2 and Inheritance+1 = 1, this translates to LOW complexity.

When Inheritance+1 is ranges 2 to 5, the Y-axis is greater than 2 and less than or equal to 3. This translates to AVG. When X-axis is greater than 5, the Y-axis is greater than 3 which translates into HIGH.

Next we model the Associations category. From Function Point Theory, FTRs are scaled a lot lower than what is seen in today's coding with respect to Associations even though they are similar. One large program shows an average of 2.5 associations, but can range up to 188. This is very common in development and is a result of improved coding practices since 1979 when FPs were first developed. When the value on Y-axis is 1.5 and Association+1 = 1, this translates to LOW complexity. When Association+1 is ranges 2 to 5, the Y-axis is greater than 2 and less than or equal to 3. This translates to AVG. When X-axis is greater than 5, the Y-axis is greater than 3 which translates into HIGH.

Drawing from Function Point mechanics where complexity is the average of [(RET Category DET Category) + (FTR Category DET Category)]

Where:

- (RET Category DET Category) = ∑ (RET Category + DET Category) / 2
- (FTR Category DET Category) = ∑ (FTR Category + DET Category) / 2

Converting RETs, FTRs, and DETS to Inheritances, Associations, and Attributes respectively, we get:

Design Complexity = Average of [(Inheritance Category Attribute Category) + (Association Category Attribute Category)]

Where:

- (Inheritance Category Attribute Category) = ∑ (Inheritance Category + Attribute Category) / 2
- (Association Category Attribute Category) = ∑ (Association Category + Attribute Category) / 2

VI. Determining the Ranges for Low, Avg, and High Design Complexity Values

To understand the response of the DC equation, we calculated every case within a reasonable range.

By producing all these cases, we can isolate when Design Complexities change in value. We observe a pattern that can be expressed through regression. This regression analysis will provide the bounding limits for Low, Avg and High DC.

VII. Determining the Missing Data for Calculating Design Complexity Values

We need to transform the matrix to have Attributes inside, Inheritance going across, and the Associations going down. This produces curves showing Attributes as a function of Inheritances. Each curve is phase- shifted due to their dependence on Associations.

Let's focus on the first Attribute Limit equation where the DC = 2 and the Association = 0:

- Attribute_Limit = 27.9 * (Inheritance + 1) ^ - 0.701
  - When Inheritance + 1 = 1, the Attribute_Limit = 28.0
  - When Inheritance + 1 = 2, the Attribute_Limit = 17.0
  - When Inheritance + 1 = 3, the Attribute_Limit = 13.0

Note that 27.9 is the First Term and -0.701 is the Second Term.

We now need to estimate the First and Second Terms as a function of DC using regression

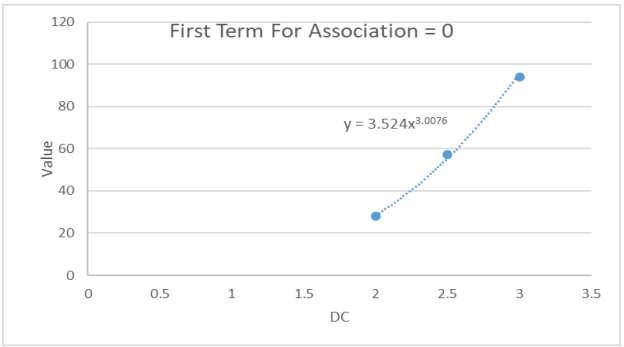

Figure 5: First Term Relationship to DC

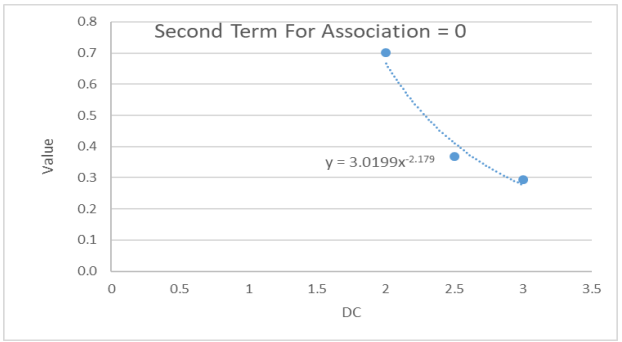

Figure 6: Second Term Relationship to DC

Performing this exercise for additional Associations, we get the following values for the First Term:

- First Term (Association = 0) = 3.524
- First Term (Association = 1) = 1.7403
- First Term (Association = 2) = 1.2486
- And so on …

Next is the Second Term:

- Second Term (Association = 0) = 3.0199
- Second Term (Association = 1) = 5.8571
- Second Term (Association = 2) = 8.9756
- And so on …

We now can perform regression to estimate the First and Second Terms as a function of Associations and DC.

VIII. Transformation to Design Complexity Space

Next step we perform regression based on the previous analysis.

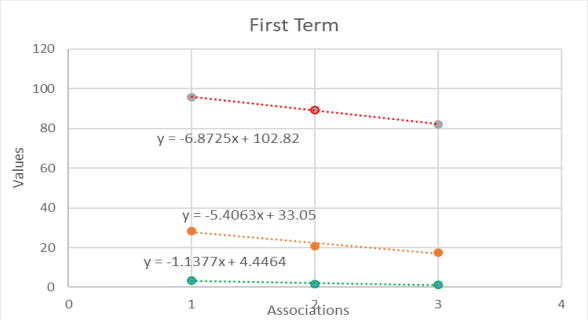

Figure 7: First Term Relationship to Associations
Red=HIGH DC; Yellow=AVG DC; Green=LOW DC

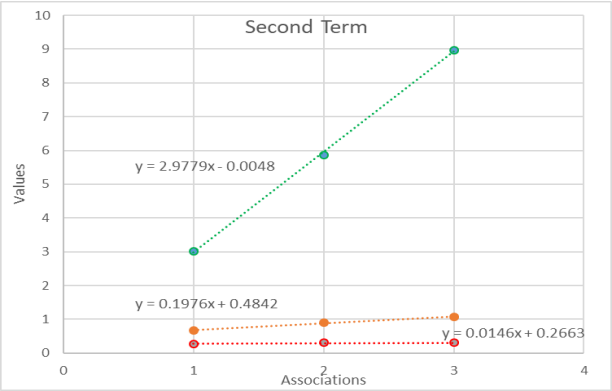

Figure 8: Second Term Relationship to Associations
Red=HIGH DC; Yellow=AVG DC; Green=LOW DC

We now can use this information and the original Attribute Limit formula:

- Attribute_Limit = First Term * (Inheritance + 1) ^ -(Second Term)

When DC = 1

- Attribute_Limit_1 = (-1.1377 * Association + 4.4464) * (Inheritance + 1) ^ - (2.9779 * Association + 0.0048)

When DC = 2

- Attribute_Limit_2 = (-5.4063 * Association + 33.05) * (Inheritance + 1) ^ -(0.1976 * Association + 0.4842)

When DC = 3

- Attribute_Limit_3 = (-6.8725 * Association + 102.82) * (Inheritance + 1) ^ - (0.0146 * Association + 0.2663)

We will know the Class DC after we enter the known Inheritance, Associations and Attributes for the specific Class.

- If Attributes < Attribute_Limit_1
  - then DC = 0
- If Attribute_Limit_1 ≤ Attributes < Attribute_Limit_2
  - then DC = 1
- If Attribute_Limit_2 ≤ Attributes < Attribute_Limit_3
  - then DC = 2
- If Attribute_Limit_3 ≤ Attributes
  - then DC = 3

For example:

- Inheritances =1
- Associations = 2
- Attributes = 15

Thus,

- Attribute_Limit_1 = 0.03
- Attribute_Limit_2 = 12.09
- Attribute_Limit_3 = 72.58

Since Attributes = 15, the DC = 2 since Attributes are between 12.09 and 72.58.

This calculates DC for a combination of Inheritances, Associations, and Attributes.

IX. Using Design Complexity and Cyclomatic Complexity to Calculate Ofps

Going back to previous section where we solved the following equation:

Function Point = (0.125 * DC ^ 2 – 0.125 * DC + 0.25)* CC ^ 2 + (-0.475 * DC ^ 2 + 0.875 * DC – 0.35) * CC +(0.875 * DC ^ 2 – 1.375 * DC + 3.25)

Where:

DC = 1 for LOW; 2 for AVG; 3 for HIGH CC values fall into 4 bins:

1. CC value 1-10
2. CC value 11-20
3. CC value 21-40
4. CC value >40

We now can simplify to a table that provides the OFPs in a simple form:

OFP	CC_BIN1	CC_BIN2	CC_BIN3	CC_BIN4
DC=0	1	2	3	4
DC=1	3	4	5	7
DC=2	4	5	7	10
DC=3	6	7	10	15

For example:

- If CC = 7,
  - then CC bin = 1;

- If Inheritances =1; Associations = 2; Attributes = 15
  - then DC = 2,
- Then the OFP = 4

Note that for DC = 0, we needed to minimize the weighting to reflect cases where the design is simplistic in nature. It made little sense to apply a weighting of 3 to a design that had zero Inheritance, zero Associations and zero Attributes. To account for someone thinking of implementing this design, we choose a value of 1 Function Point and went from there using CC.

X. Summary

This methodology successfully creates a new and simple OFP table that is dependent on CC and DC. We extracted a DC that captures interface relationships based on inheritances, associations and attributes in the actual code. This DC is based on Albrecht's original analysis where DC was a factor but never exclusively identified. This new table is independent of transactional and database qualifiers. Next steps are to incorporate this methodology into an automated Function Point counter that reads actual source code to extract UML definition such as inheritances, associations and attributes to derive the OFPs. This effort is being implemented into the Objective Function Point counter that will reside in the Unified Code Counter Govt (UCC-G) version and the University of Southern California (USC) Unified Code Counter Java version (UCC-J).

References Références Referencias

1. Rational, UML, 1.1 Notation Guide, Rational Software, 1997.
2. A.J. Albrecht. Function Point Analysis. Encyclopedia of Software Engineering, 1: John Wiley & Sons, 1994.
3. T. Uemura, S. Kusumoto, and K. Inoue, Function Point Measurement Tool for UML Design Specification, Osaka Japan, Nov 1999.
4. I. Brown, Using Entity Relationship Diagrams to Count Data Functions, 2007.
5. McCabe T., "NIST Special Publication 500-235", 1996.
6. Richard Seidl "Modeling Metrics for UML Diagrams", 2010.