A SURROGATE METRIC FOR ABSTRACTION COMPLEXITY IN OO SOFTWAREDESIGN:THEORETICAL VALIDATION

Bajeh,Amos Orenyi
member, *IEEE Computer Society*
(UniversitiTeknologi PETRONAS, Seri Iskandar, Perak, Malaysia
bajeh_amos@yahoo.com)
Shuib Basri
(UniversitiTeknologi PETRONAS, Seri Iskandar, Perak, Malaysia
Shuib_basri@petronas.com.my)
Low Tan Jung
(UniversitiTeknologi PETRONAS, Seri Iskandar, Perak, Malaysia
lowtanjung@petronas.com.my)

ABSTRACT: *The measurement of the quality attributes of software designs is crucial to the development of quality software. Design refinements, in order to enhance the quality of software products, become harder and costlier after implementation. Quality indicators that can be measured from High Level Designs (HLD) provide early information that can guide developers in building quality into software product at the design phase before the implementation of the design. This paper presents the identification and theoretical validation of an OO metrics that can be taken as surrogates for the measurement of the complexity due to the level of abstraction in OO software HLD. The metric can be used as part of measurement models for external quality attributes such as maintainability. It can serve as early indicator of complexity level in design and thus aid in developing maintainable products which are crucial to the success of development processes, such as the agile software development process, that tends to manage complexity by using iterative and incremental approach to software development.*

Keywords:OO metrics, Validation, Abstraction, Complexity, Agile development, Retrospectives

1. INTRODUCTION

The significance of measurement in any engineering discipline is succinctly captured by the statement: “You can’t control what you can’t measure”-Tom DeMarco. Measurement plays an integral part in the development of engineering artifact such as software designs in software engineering. Measuring the attributes of software designs informs developers on the quality of their products and how to control complexity in order to develop software products with desirable qualities [1-5]. Measurement of the internal attributes of software serves as surrogates for the estimation of the external attributes such as usability, maintainability, changeability, analyzability, etc. This is depicted by Briand et al.’s description of a causal chain model shown in Figure 1 [6]. It shows that structural properties or attributes have impact on the cognitive complexity of a design and this in turn affects the external quality attributes of the design. It implies that in order to estimate the external quality attributes of software designs, the internal quality attributes must be measured accurately as much as possible. In Object-

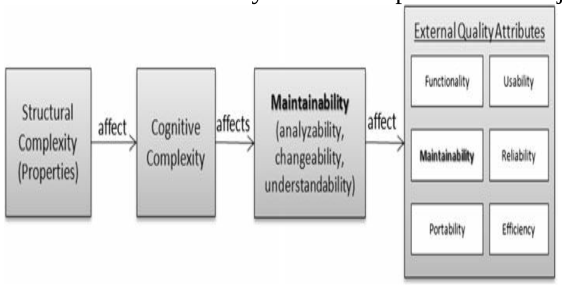

Structural Complexity (Properties)
affect
Cognitive Complexity
affects
Maintainability (analyzability, changeability, understandability)
affect
External Quality Attributes
Functionality
Usability
Maintainability
Reliability
Portability
Efficiency

Fig 1: Briand et al.’s Causal chain model

Oriented (OO) software development research literature, several OO metrics [7-11] have been proposed to measure the internal attributes of OO software design, and several quality models [12-19] have been proposed to estimate the external quality attributes of OO software products.

Complexity as a concept is multifaceted and only specific dimension of it can be measured at a time; attempting to measure complexity as a single value has been seen to be an impossible task; Fenton [20] liken such attempt to the a search for the impossible holy grail. Thus, to measure the complexity of an OO software design, specific viewpoint or dimension should be considered. The OO mechanisms: abstraction, encapsulation and information hiding; inheritance; and polymorphism can be considered as specific dimensions from which design complexity can be measured. This paper focuses on the identification and theoretical validation of an OO metric that can serve as surrogate measure of the complexity due to the level of abstraction in an OO HLD. The identification and theoretical validation of OO abstraction metric is to ensure that the metric measures the concept of complexity [21], and conforms to the theory of measurement [22]. Consequently, this will ensure that the metric properly discriminates between different alternative OO software designs. Metrics that properly discriminating between designs will ensure that the same value is not assigned to different designs having different levels of the measured attribute (e.g. abstraction).

In the literature, the theoretical validation of OO metrics is seldom carried out before the proposal of some of the existing external quality measurement models that use the metrics [4, 16, 17, 18]. Consequently this often renders the models inherently invalid theoretically. The empirical validation of such models can be nothing but a premature task since some/all of their constituent metrics are theoretically invalid.

To aid in early quality management of OO software, this study identifies and validates OO abstraction metric that can be collected/measured from OO HLD. The HLD is the UML class diagram -the de facto design notation for OO software- which consists of classes, their attributes and methods declarations, and class interrelationships. The declarations specify the attributes' data types and visibility; methods' visibility, return data types and signatures.

The other parts of this paper are organized as follows: Section II presents the approach that will be used to validate the OO metric that is identified in section III. The theoretical validation of the metric is presented in section IV while the applicability of the metric is discussed in section V. Section VI concludes the paper and states the future work.

2. METHODOLOGY

Existing OO abstraction measurement metrics that can be measured from HLD are identified from the literature and discussed. The promising metric, among the identified metrics, is further validated using existing metric validation properties and axioms. The theoretical validation of the metric is carried out using two approaches: the property-based approach and the measurement theory approach. The property-based approach provides the necessary but not sufficient properties that a metric must satisfy. The properties only indicate the quality concept that a metric measures; quality concept such as complexity, size and length. On the other hand, the measurement theory approach provides sufficient properties that must be satisfied to show that a metric is theoretically valid but do not indicate the quality concept the metric measures. The complexity properties proposed by Briand et al. [21] are used in this study for the property-based validation while the axioms for distance function, used by Poels n Dedene [22] to develop the DISTANCE framework, are used for the measurement theory-base validation. The satisfaction of the complexity properties will show that the identified metric measures complexity, while the satisfaction of the distance function axioms will show that the metric conforms to the theory of measurement.

2.1 Briand et al. Properties for Complexity Metric

In [21], Briand et al. defined the properties of five quality concepts. The properties of complexity metrics was defined thus:

A system, S is defined as a pair $S=<E,R>$ where E is the elements of the system and R is the binary relations on the elements of the system (i.e. $R \subseteq E \times E$), representing the relationships between the elements of the system. Base on this definition of a system, the following are the five Briand et al.'s properties of a valid complexity metric (in this paper, the properties are designated as P1, P2, P3, P4 and P5):

P1. Nonnegativity: The complexity of a system $S=<E,R>$ is nonnegative, i.e. $Complexity(s) \geq 0$.

P2. Null Value: The complexity of a system $S=<E,R>$ is null if R is empty, i.e. $R=\phi \Rightarrow Complexity(s)=0$.

P3. Symmetry: The complexity of a system does not depend on the convention chosen to represent the relationships between its elements, i.e. given that $S=<E,R>$ and $S^{-1}=<E,R^{-1}>$, $\Rightarrow Complexity(S)=Complexity(S^{-1})$.

P4. Module Monotonicity: The complexity of a system $S=<E,R>$ is no less than the sum of the complexities of any two of its modules with no relationships in common $S=<E,R>$ and $m_1=<E_{m1},R_{m1}>$ and $m_2=<E_{m2},R_{m2}>$ and $m_1 \cup m_2 \subseteq S$ and $R_{m1} \cap R_{m2}=\phi$,

$\Rightarrow Complexity(S) \geq Complexity(m_1)+Complexity(m_2)$

P5. Disjoint Module Additivity: The complexity of a system $S=<E,R>$, composed of two disjoint modules m_1 and m_2, is equal to the sum of the complexities of the two modules, i.e. $S=<E,R>$, $S=m_1 \cup m_2$ and $m_1 \cap m_2=\phi$,

$\Rightarrow Complexity(S)=Complexity(m_1)+Complexity(m_2)$

2.2 Distance Function (Measurement Theory) Axiom

The distance function axioms for the measurement theory-based validation approach are as follows [22] (in this paper, the axioms are designated as A1, A2, A3 and A4):

Let A be a set. The function δ: A x A $\rightarrow \Re$ is a metric if and only if:

A1: Non-negativity: $\forall$ a, b $\in$ A: δ(a, b) ≥ 0

A2: Identity: $\forall$ a, b $\in$ A: δ(a, b) = 0 $\Leftrightarrow$ a = b

A3: Symmetry: $\forall$ a, b $\in$ A: δ(a, b) = δ(b, a)

A4: The Triangle inequality:

$\forall$ a, b, c $\in$ A: δ(a, b) $\leq$ δ(a, c) + δ(c, b)

3. Identification of OO Abstraction Metrics

The appropriate use of the OO mechanisms enhances the quality of OO software products [23]. A properly abstracted design with good encapsulation and hiding of data and operation details reduces complexity and consequently enhances the external quality attributes of OO software. Measuring the level of use of the OO mechanisms in a design can serve as a surrogate for estimating the (external) quality of the design. This Section presents the identification of the OO metric for the measurement of OO abstraction. The identified metric will be validated in the Section IV.

Abstraction, in computer programming, is the concept used to reduce details and allow a programmer (designer) to focus on one design perspective at a time [2], thereby reducing complexity. It is a concept employed in software development to manage complexity by keeping away implementation details and allowing the designer to focus on one design perspective at a time. The interfaces of the abstracted components/parts (i.e. hidden implementation details) are subsequently used in other design components.

In the literature, the measurement of the abstraction of OO software design has relatively received minimal attention compare to the measurement of other mechanisms (e.g. information hiding, inheritance and polymorphism). Very few OO metrics have been proposed for the measurement of the level of abstraction in OO designs. Some of the metrics for abstraction identified from the literature are the Measurement of Functional Abstraction (MFA) proposed by Bansiya and Davis [10] and the Abstractness (A) metrics defined in the Borland Together Tool [24].

The MFA metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. This metric is an inheritance based metrics that is only measurable from inheritance hierarchies. The Abstractness (A) metric is “the ratio of the number of the abstract classes and interfaces in a package to the total number of classes and interfaces in the same package”. It simply measures the percentage of the total number of classes and interfaces that are abstract in a package.

These metrics, MFA and A, are the ratios of two numbers and thus do not completely discriminate between different designs. For instance, if a design, P has a total of 20 classes out of which 10 are interfaces and abstract classes, and another design Q has 40 classes out of which 20 are interfaces and abstract classes; the Abstractness (A) metric will yield the same value of 0.5 for the two designs P and Q implying that they have the same level of abstraction. This deficiency is also noticed with MFA since it is also the ratio of two numbers. Thus, MFA and A do not satisfy the measurement theory Identity axiom (A2 above): using the instance given above, the distance between the metric values of P and Q is zero and yet they are not the same in terms of abstraction.

To measure the complexity due to the level of abstraction of a design, first, a metric that measures abstraction as defined above must be used and such metric must be theoretically valid by satisfying the properties (P1-P5) and the axioms (A1-A4) stated in section 2. The metric that is proposed for the measurement of abstraction in this study, measures the total number of points where Abstract Data Types (ADT) or User-defined Data Type (UDT) are used in a design. This number of points signifies the number of places in a design where details are kept away (abstracted) in other to facilitate the understanding of one aspect of the design at a time, and thus enhance the external quality (e.g. analyzability) of the design. Fig. 2 depicts a model of this metric, where two data items of class X are of the ADT/UDT Person and Dog. Here, the arrows link class X to Person and Dog showing that the implementation details of the two data elements are in classes Person and Dog.

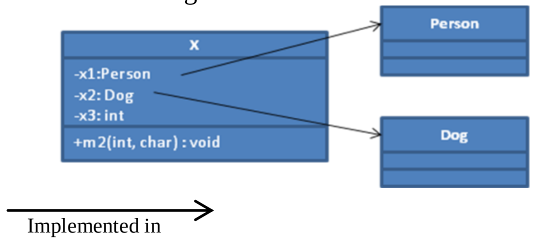


Fig 2: Number of Abstraction Points (NAP)

In this paper, this metrics is named Number of Abstraction Points (NAP) and its theoretical validity is presented in the next section. The metric will serve as a surrogate for the measurement of complexity due to the level of abstraction in OO design. It can be used as one of the measures in developing external quality measurement models.

4. Metric Validation

The designs in Fig 3, 4 and 5 will be used to validate NAP. Fig. 3 depicts the model for measuring NAP, Fig. 4 is the design of some selected classes in junit-a java based testing framework, and Fig. 5 is a design collected from the literature.

4.1 Property-based Validation

In other to use the Briand et al.’s complexity properties to validate the proposed metric NAP, the system (S), elements (E) and the binary relations (R) have to be defined. Let an OO UML class diagram design be a system S , E is the set of the classes in the system design, and R is the set of data (attributes and parameters) that are ADT or UDT in the classes. The data items of types ADT/UDT represents a logical linkage between the classes in which the data items are declared and the ADT/UDT definition which specifies the implementation of the data type. This logical relation is what is depicted in Fig. 2.

The property-based validation of NAP follows:

P1.Nonnegativity: obviously this property is satisfied by NAP since for any given design the total number of data elements of type ADT cannot be a negative number. It is zero when there is no any data/attribute of the type ADT, and it is greater than zero when there are data items of type ADT. Thus, $NAP(S) \geq 0$; the *nonnegativity* property is satisfied.

P2. Null Value: the NAP for a system in which none of its data elements is of ADT will be zero i.e. $R = \phi$ thus $NAP(S) = 0$.

P3. Symmetry: using system S_1 in Fig. 3 (a); the inverse of this system is S_2 , depicted in Fig. 3 (b), in which the direction of the relation is reversed (i.e. $S_2 = S_1^{-1}$). For the two systems, $NAP(S_1) = NAP(S_2) = 2$ thus, the *symmetry* property is satisfied.

P4. Module Monotonicity: Using the UML diagram of junit (version 3.8.1) shown in Fig. 4 (a) as a system S; Fig 4(b) is a representation of S using the defined model in Fig 2. Modules $m1$ and $m2$ in Fig 4(c) can be taken as two modules of S that do not have any relationship (i.e. $m1 \cup m2 \subseteq S$ and $R_{m1} \cap R_{m2} = \phi$). Using the metric NAP to measure the system and the two modules we have, $NAP(m1) = 5$ and $NAP(m2) = 3$; which show that $NAP(S) \geq NAP(m1) + NAP(m2)$. This expression is also true for any two of the modules in Fig 4(c). Thus, the *module monotonicity* property is satisfied.

P5. Disjoint Module Additivity: this property is also implied by module monotonicity property P4 above, but with the condition that $S = m1 \cup m2$ and $m1 \cap m2 = \phi$. Using Fig 4(c), assuming module m1 consist of classes TestCase, TestResult, Test, AssertionFailedError, Protectable and TestListener; and that S is made up of m1 and m2, then NAP of S is the sum of the NAP of the two modules i.e $NAP(S) = NAP(m1) + NAP(m2)$. Thus, the *disjoint moduleadditivity* property is satisfied

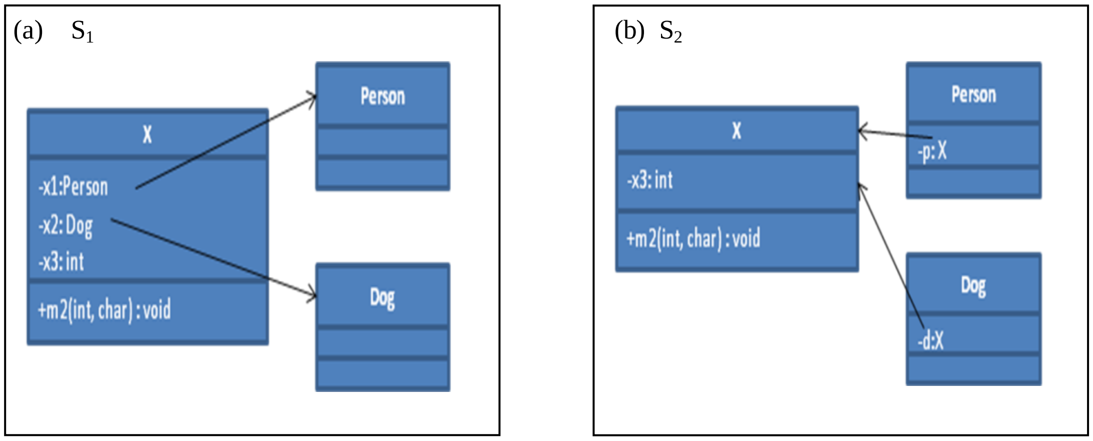


Fig 3: Abstraction model

4.2 Measurement Theory-based Validation

The distance between any two points/values is the absolute difference between the two points/values. The designs in Fig. 4 and 5 will be used to show that the metric satisfies the A2, A3 and A4 axioms stated in section II. The NAP values for some of the classes in the figures are: BasicComponent=1, Application=4, UIComponent=3, BaseTestRunner=3 and Clock=3.

A1.Nonnegativity: this axiom is obviously satisfied since the number of abstraction points in any given design can either be zero or greater than zero i.e. a design can either have one or more variables of type ADT(s) or no variable of type ADT. Thus the value of NAP is always nonnegative (≥ 0).

A2. Identity: by the definition of the metric, NAP, any two OO HLD with equally number of variables of ADT are intuitively having the same level of abstraction. This implies that the value of NAP will be the same for the two designs and thus the distance between the two designs in terms of NAP will be zero (i.e. δ (a, b) = 0 where a and b represents the two designs). For instance, the BaseTestRunner class and the Clock class in Fig 4(b) and 5(b) respectively are of the same level of abstraction; thus, δ(BaseTestRunner, Clock) = $|3-3|=0$. This can also be seen in classes TestResult and Application in Fig 4(b) and 5(b) respectively. Thus, the identity property is satisfied.

A3. Symmetry: for any two of the designs in Fig. 5, the absolute distance between them is symmetric. For instance, BasicComponent and Clock: δ (BasicComponent, Clock) = δ (Clock, BasicComponent) i.e. $|1-3|=|3-1|$. This is also true for any combination of any two designs since the absolute difference between them is the same irrespective of which the minuend or subtrahend. Table 1 explicitly shows this for some of the designs in Fig 4 and Fig 5. Thus, the symmetry axiom is satisfied.

A4. The triangle inequality: also, for any three designs, the distance between the NAP of any two (of the three) designs will be less than the sum of the distance between each of the two designs and the third one. For instance, BasicComponent, Application and UIComponent having the metric values 1, 4 and 3 respectively:

δ(BasicComponent, Application) $\leq$
δ(BasicComponent, UIComponent) + δ (UI Component, application)

i.e. $|1-4|\leq|1-3|+|3-4|$ is true.

Also, Table 2(a) show that this axiom is satisfied for some other possible combination of any three of the designs in Fig 5 while Table 2(b) shows how some possible permutation of three of the classes in Fig 5 satisfy the axiom. Thus the triangle inequality axiom is satisfied.

5. Metric Applicability

Having shown that the metrics *NAP* for abstraction is a valid metrics based on measurement theory axioms, and that the metrics satisfy the Briand et al.'s complexity properties, it can be used as surrogates for measuring the complexity due to the level of abstraction in OO software design. The effect of complexity on external quality attributes is the same [25].For instance, external quality attribute such as analyzability and testabilitycan be expressed as the inverse of complexity: the more complex a system becomes, the less analyzable and testable the system will be; conversely, the less complex a system becomes, the more analyzable and testable the system will be. This type of relationship between quality concepts (e.g. complexity) and external attributes (e.g. analyzability) is what is depicted in the causal chain model (Fig. 1) described by Briand et al. Thus, the complexity metric *NAP* can be used as indicator of complexity from abstraction viewpoint.

Theoretically valid metrics such as *NAP* can be used as quality indicators for OO HLD either as standalone metric or part of quality models for external quality attributes such as analyzability, testability, maintainability, etc. Design complexities need to be managed right from the early stage of a software process in order to build software products that are maintainable. Agile software development strongly advocates for tasks such as 'responding to change requests' over 'following a rigid plan' which is perceived to cause delay in software delivery. Thus, controlling the complexity of designs will ensure the development of products that can be easily maintained in order to incorporate change requests. Agile software development methodologies uphold the principles of frequently delivering, to product owners, working software with best architectural designs [26]. The Agile approach is incremental and iterative [26 - 30]. Business functionalities are grouped into iterations. At the end of iteration, a new version of the working software is delivered to client who in turn provides feedback to the development team. The next iteration, in addition to incorporating the feedback from the client, involves the implementation of additional functionality. Thus the software product incrementally grows in size and functionality. The iterative approach itself helps in managing design complexities in a divide-and-conquer manner; however, developers need to keep tab on how the complexity of the design grows with iterations and measurement can serve this purpose. This is much more so during the task of *agile retrospectives* when the development team reviews the activities carried out during an iteration and identify what worked well, what has not been done well, how best to resolve impediments and also improve on the software design. A typical retrospective agenda [27, 29]includes, among others, *gathering data*, *generating insights from data* and *deciding what to do to improve the software design*. *NAP* becomes handy here for the estimation of the level of complexity due to abstraction in the design. This measurement can be collected as data to give insight on how the complexity of the design is growing over time through the iterations.This can lead the team to decide how to manage the growing complexity of the software design by refining the design in order to minimize complexity.

6. CONCLUSION AND FUTURE WORK

This paper has identified and theoretically validated a metric, *Number of Abstraction Points* (*NAP*), which can serve as surrogates for measuring the complexity due to the level of abstraction in OO software high-level designs. The measurement theory distance function axioms are used to validate the metric in order to show that the metric complies with the theory of measurement, while the complexity metric properties proposed by Briand et al. are used to show that the metric measures the concept of complexity.

The applicability of the validated metric, *NAP* was presented and it promises to be useful quality indicator in terms of the design complexityof OO software in software process such as the agile process that requires highly maintainable software designs for easy incorporation of changes as software requirements evolves and designs are refined to improve quality. During agile retrospectives, *NAP* measurement can be used to gather information on how the complexity of the software increases in the iteration under review. This information will give insight into the design complexity and thus triggers the agile team into deciding on how to refine the design to keep complexity in check if it is growing unfavorably.

The identification and validation of complexity metrics using other OO mechanisms (e.g. inheritance and polymorphism) as specific viewpoints will be carried out as future works. Also, the empirical validation and analysis showing practical utilization of the metric in an agile development process (e.g. scrum) will also be conducted as future work

.

Table 1: Symmetry axiom satisfied

a	b	$\delta(a,b)=\delta(b,a)$	
Clock	TestResult	$\lvert 3-4\rvert = \lvert 4-3\rvert$	true
BasicComponent	Application	$\lvert 1-4\rvert = \lvert 4-1\rvert$	true
UIComponent	BasicComponent	$\lvert 3-1\rvert = \lvert 1-3\rvert$	true
TestCase	UIComponent	$\lvert 1-3\rvert = \lvert 3-1\rvert$	true
Application	BaseTestRunner	$\lvert 4-3\rvert = \lvert 3-4\rvert$	true

Table 2: The Triangle inequality axiom satisfied

X: BasicComponent , Y: UIComponent , Z: Application and W: Clock

(a) Possible combination of 3 of the classes in Fig 4

(b) (b) Possible permutation of 3 of the classes in Fig 4

a	b	c	$\delta(a,b)\le\delta(a,c)+\delta(c,b)$	Valid?
X	Y	Z	$\lvert 1-3\rvert \le \lvert 1-4\rvert + \lvert 4-1\rvert$	true
X	Y	W	$\lvert 1-3\rvert \le \lvert 1-3\rvert + \lvert 3-3\rvert$	true
X	Z	W	$\lvert 1-4\rvert \le \lvert 1-3\rvert + \lvert 3-4\rvert$	true
Y	Z	W	$\lvert 3-4\rvert \le \lvert 3-3\rvert + \lvert 3-4\rvert$	true

a	b	c	$\delta(a,b)\le\delta(a,c)+\delta(c,b)$	Valid?
X	Y	Z	$\lvert 1-3\rvert \le \lvert 1-4\rvert + \lvert 4-3\rvert$	true
X	Z	Y	$\lvert 1-4\rvert \le \lvert 1-3\rvert + \lvert 3-4\rvert$	true
Y	X	Z	$\lvert 3-1\rvert \le \lvert 3-4\rvert + \lvert 4-1\rvert$	true
Y	Z	X	$\lvert 3-4\rvert \le \lvert 3-1\rvert + \lvert 1-4\rvert$	true
Z	X	Y	$\lvert 4-1\rvert \le \lvert 4-3\rvert + \lvert 3-1\rvert$	true
Z	Y	X	$\lvert 4-3\rvert \le \lvert 4-1\rvert + \lvert 1-3\rvert$	true
Z	W	Y	$\lvert 4-3\rvert \le \lvert 4-3\rvert + \lvert 3-3\rvert$	true

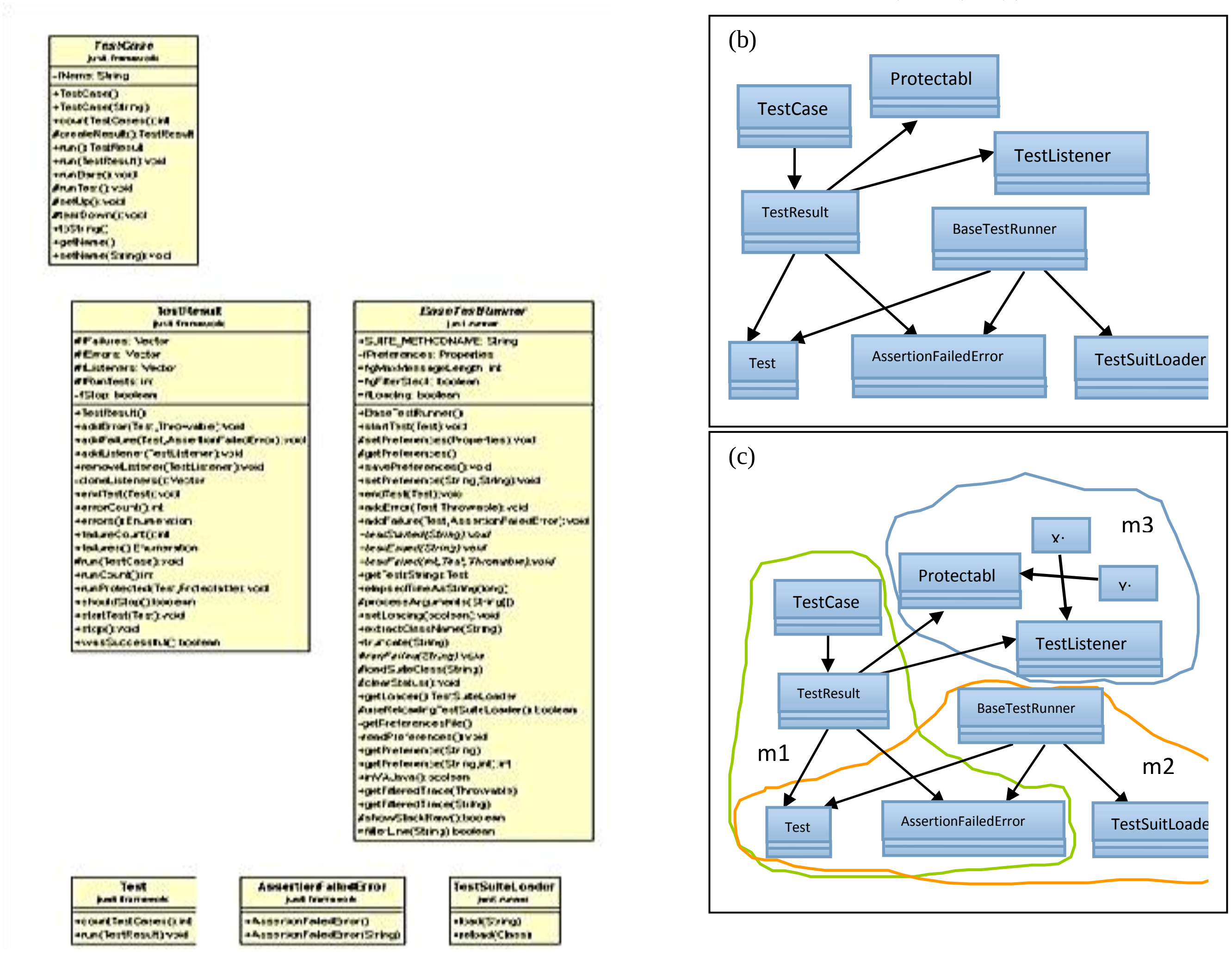


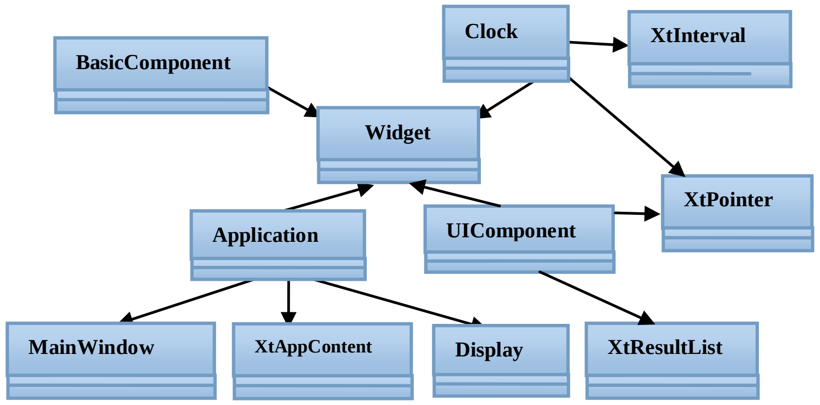
Fig 4: UML class diagram (showing some classes only) for junit- a java based testing framework

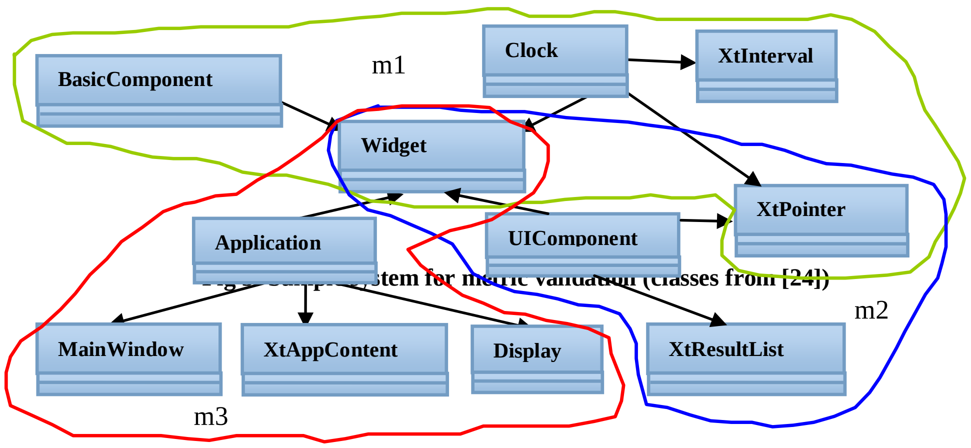
Application
#_display : Display #_appContext : XtAppContext #_applicationClass: char #_windows : MainWindow #_numWindows: int
-registerWindow(MainWindow) : void -unregisteredWindow(MainWindow):void #initializa (int*, char*): void #initialize (unsigned int*, char**) void #handleEvents(): void +Application(Widget, char*, int, int) +~Application() +unmanage(): void+iconify(): void+display(): Display +appContent():XtAppContent +applicationClass(): const char +className():const char*

Clock
-_data : int-_id : XtIntervalId
-timeout() : void -speedChanged(int):void -timeoutCallback(XtPointer, XtIntervalId): void -speedchangedCallback(Widget, XtPointer, XtPointer): void #tick(): void +clock(Widget, char*, int, int) +~clock()+stop(): void +pulse(): void +start(): void +className(): const char

BasicComponent
#name : char #w : Widget
#BasicComponent(const char*) +BasicComponent() +manage(): void +unmanage(): void +baseWidget(): Widget

UIComponent
-widgetDestroyedCallback(Widget, XtPointer, XtPointer) : void #UIComponent(const char*) #installDestroyHandler (): void #widgetDestroyed () : void #setDefaultResources(const Widget, const String): void #getResources(const XtResourceList, const String*): void +UIComponent ()+manage(): void+className():const char*





REFERENCES

[1] F.T.Sheldon, K. Jerath, and H. Chung, Metrics for maintainability of class inheritance hierarchies, *Journal of Software Maintenance and Evolution: Researchand Practice*, vol. **14**, pp. 147-160, 2002.

[2] R.S. Pressman, Software Engineering: *A practitioner's approach*,McGraw Hill ,New York, USA, 2005.

[3] L. Badri, M. Badri and A. B. Gueye, Revisiting Class Cohesion: An empirical investigation on several systems, *Journal of Object Technology*, vol. **7**, pp. 55-75, 2008.

[4] M. Genero, M. Piattini, E. Manso and G. Cantone, "Building UML class diagram maintainability prediction models based on early metrics, *In Software Metrics Symposium, 2003.Proceedings. Ninth International*, pp. 263-275, 2003.

[5] K. Aggarwal, Y. Singh, P. Chandra and M. Puri, Measurement of software maintainability using a fuzzy model, *Journal of Computer Sciences*, vol. **1**, pp. 538-542, 2005.

[6] L.C. Briand, J. Wust, S.V. Ikonomovski, and H. Lounis, Investigating quality factors in object-oriented designs: An industrial case study, In ICSE'99: *Proceedings of the 21st International Conference on Software engineering*, New York, NY: ACM, pp. 345-354, 1999.

[7] S. R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design, *Software Engineering, IEEE Transactions on*, vol. **20**, pp. 476-493, 1994.

[8] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, 1994.

[9] W. Li and S. Henry, Object-oriented metrics that predict maintainability, *J. Syst. Software*, vol. **23**, pp. 111-122, 1993.

[10] J. Bansiya and C.G. Davis, A hierarchical model for object-oriented design quality assessment, *Software Engineering, IEEE Transactions on*, vol. **28**, pp. 4-17, 2002.

[11] M. Genero, M. Piattini and C. Calero, A survey of metrics for UML class diagrams, Journal of Object Technology, vol. 4, pp. 59-92, 2005.

[12] International Organization for Standardization/International Electrotechnical Commission. 2001. ISO/IEC 9126-1 Standard, Software Engineering, Product Quality, Part 1: Quality Model," Author, Geneva.

[13] J.A. McCall, P.K. Richards, G.F. Walters, Rome Air Development Center and United States. Air Force Systems Command. Electronic Systems Division. 1977, Factors in Software Quality, Rome Air Development Center, Air Force Systems Command.

[14] R. G. Dromey, A model for software product quality, *IEEE Transactions onSoftware Engineering*, vol. **21**, pp. 146-162, 1995.

[15] B. W. Boehm, J.R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod and M.J. Merrit, *Characteristics of software quality*, 1978.

[16] S. Rizvi and R. Khan, Maintainability Estimation Model for Object-Oriented Software in Design Phase (MEMOOD), Journal of Computing, vol. 2, pp. 26-32, 2010.

[17] P. Antonellis, D. Antoniou, Y. Kanellopoulos, C. Makris, E. Theodoridis, C. Tjortjis and N. Tsirakis, A data mining methodology for evaluating maintainability according to ISO/IEC-9126 software engineering–product quality standard,Special Session on System Quality and Maintainability-SQM2007, 2007

[18] I. Heitlager, T. Kuipers, and J. Visser, A practical model for measuring maintainability, In Quality of Information and Communications Technology, 2007 (QUATIC 2007), 6th International Conference on the, pp. 30-39, 2007.

[19] F. B. Abreau, M. Goulao and R. Esteves, Towards the design quality evaluation of object-oriented software systems, In Proceedings of the 5th International Conference on Software Quality, 1995.

[20] N. Fenton, Software measurement: A necessary scientific basis, IEEE Transactions on Software Engineering, Vol. 20, No. 3, March 1994.

[21] L. Briand, S. Morasca, and V. Basili, Property-based software engineering measurement, IEEE Transaction on Software Engineering, 22(1), 68-86, 1999.

[22] G. Poels and G. Dedene, Distance-based software measurement: Necessary and sufficient properties for software measures,Information and Software Technology, 42(1), 35-46.

[23] H.D. Rombach, Design measurement: Some lessons learned, IEEE Software, vol. 7, No. 2, pp 17-25, 1990.

[24] Borland Together Tool, 2008.

[25] R. V. Binder, Design for testability in Object-Oriented Systems, Communication of ACM, vol. 37, no. 9, 1994.

[26] C. Mike, Succeeding with Agile: Software Development using Scrum. Pearson Education India, 2010.

[27] J. Blankenship, M. Bussa and S. Millett, The art of agile development, in Pro Agile. NET Development with ScrumAnonymousSpringer, 2011, pp. 1-11.

[28] C. Larman, Agile and Iterative Development: A Manager's Guide,Addison-Wesley Professional, 2004.

[29] J. Shore, The Art of Agile Development. O'Reilly, 2008.

[30] T. Dyba and T. Dingsoyr, "What do we know about agile software development?",IEEE Software, vol. 26, pp. 6-9, 2009.