Measuring Understandability of Aspect-Oriented Code

Mathupayas Thongmak[1] and Pornsiri Muenchaisri[2]

[1] Department of Management Information Systems, Thammasat Business School, Thammasat University, Thailand
Mathupayas@gmail.com
[2] Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Thailand
Pornsiri.Mu@chula.ac.th

Abstract. Software maintainability is one of important factors that developers should concern because two-thirds of a software system's lifetime-cost involve maintenance. Understandability is one of sub-characteristics that can describe software maintainability. Aspect-oriented programming (AOP) is an alternative software development paradigm that aims to increase understandability, adaptability, and reusability. It focuses on crosscutting concerns by introducing a modular unit, called "aspect". Based on the definition of understandability that "the related attributes of software components that users have to put their effort on to recognizing the logical concept of the components", this paper proposes seven metrics for evaluating understandability of aspect-oriented code using different levels of dependence graphs. The metrics are applied to two versions of aspect-oriented programs to give an illustration.

Keywords: Software Metrics, Aspect-Oriented, Understandability.

1 Introduction

Software quality has become essential to good software development. Quality factors consist of efficiency, reliability, reusability, maintainability, etc. Software maintainability is an important factor that developers should concern because two-thirds of a software system's lifetime cost involves maintenance [1]. Maintainability characteristic composes of three sub-characteristics: Testability, Understandability, and Modifiability [2]. For understandability sub-characteristic, to comprehend software, the factors that influences on the comprehensibility are an internal process of humans and an internal software quality itself [3, 4]. Aspect-oriented programming (AOP) is an alternative software development paradigm that aims to increase comprehensibility, adaptability and reusability [5]. It focuses on crosscutting concerns by introducing a modular unit, called "aspect".

There are many research works studied on aspect-oriented software measurements and understandability assessments. Zhao proposes dependence graphs for aspect-oriented programs and presents cohesion metrics and structural metrics based on the graphs [6, 7, 8]. Shima et al. introduce an approach to experimental evaluation of software understandability from the internal process of humans [4]. Jindasawat et al. investigate correlations between object-oriented design metrics and two

H. Cherifi, J.M. Zain, and E. El-Qawasmeh (Eds.): DICTAP 2011, Part II, CCIS 167, pp. 43–54, 2011.

sub-characteristics of maintainability: understandability and modifiability [9]. Sheldon et al. propose metrics for maintainability of class inheritance hierarchies [10]. The metrics are evaluated from understandability and modifiability sub-characteristics. Genero et al. study relationships between size and structural metrics of class diagrams and maintainability time [11].

This paper aims to propose metrics for understandability of AspectJ code structure. We define understandability as "The related attributes of software components that users have to put their effort on to recognizing the logical concept of the components". Some related works are discussed in section 2. Section 3 presents the notion of Aspect-Oriented Programming and AspectJ. Section 4 shows dependency graphs adapted from [6] as a representation for aspect-oriented program structure. Metrics for understandability are introduced in section 5. The paper ends with some conclusion and future work.

2 Related Work

This section summarizes the related work. Some researches are mentioned to the understandability measurements, and some works are about aspect-oriented software metrics.

The dependence graphs of our work are adapted from [6]. Zhao introduces dependency graphs for aspect-oriented program and defines some metrics based on his graphs. In [8], Zhao proposes cohesion metrics to evaluate how tightly the attributes and modules of aspects cohere. And in [7], he presents complexity metrics to measure the complexity of an aspect-oriented program from different levels of dependence graphs.

Both Jindasawat et al. and Genero et. al use controlled experiments to investigate correlations between object-oriented design metrics and two sub-characteristics of maintainability: understandability and modifiability [9, 11]. Jindasawat et. al find the relationships between metrics from class and sequence diagrams, and understandability exam scores and modifiability exam scores. Genero et. al study the correlation between metrics from class diagrams, and understandability times and modifiability times.

Sheldon et al. propose objective metrics for measuring maintainability of class inheritance hierarchies [10]. The metrics are composed of understandability and modifiability metrics. Shima et al. introduce an approach to experimental evaluation of software understandability [4]. They propose software overhaul as an approach to externalize the process of understanding software systems and the probabilistic models to evaluate understandability.

3 Aspect-Oriented Programming with AspectJ

Aspect-oriented programming (AOP) paradigm, also called aspect-oriented software development (AOSD), attempts to solve code tangling and scattering problems by modularization and encapsulation of crosscutting concerns [12]. To encapsulate various types of crosscutting concerns, AOP introduces a new construct called an *aspect*. An *aspect* is a part of a program that crosscuts its core concerns, the *base code* (the non-aspect part of a program), by applying *advice* over a number of *join point*, called a *pointcut*. *Join point*s are well-defined point in the flow of a program where the

aspect can apply along the program execution. Join points include method execution, the instantiation of an object, and the throwing of an exception. A *pointcut* is a set of join points. Whenever the program execution reaches one of the join points described in the pointcut, a piece of code associated with the pointcut (called advice) is executed. This allows a programmer to describe where and when additional code should be executed in addition to an already defined behavior. *Advices* are method-like constructs that provide a way to express crosscutting action at the join points that are captured by a pointcut [13]. There are three kinds of advice: *before*, *after*, and *around*. A *before advice* is executed prior to the execution of join point. An *after advice* is executed following the execution of join point. An *around advice* surrounds the join point's execution. It has the ability to bypass execution, continue the original execution, or cause execution with an altered context.

AspectJ is the first and the most popular aspect-oriented programming language [12]. Two ways that AspectJ interact with the base program are pointcut and advice, and inter-type declarations. Pointcut and advice are described above. The join points in AspectJ are *method call*, *method* or *constructor execution*, *field read* or *write access*, *exception handler execution, class* or *object initialization*, *object pre-initialization*, and *advice execution* [13]. An *inter-type declaration*, also called *introduction*, is a mechanism that allows the developer to crosscut concerns in a static way [14]. Six types of possible changes through Inter-type declaration are adding members (methods, constructors, fields) to types (including other aspects), adding concrete implementation to interfaces, declaring that types extend new types or implement new interfaces, declaring aspect precedence, declaring custom compilation errors or warnings, and converting checked exceptions to unchecked.

```
ce0  public abstract class Account {
ac1    private float _balance;
ac2    private int _accountNumber;
mc3    public Account(int accountNumber) {
s4       _accountNumber = accountNumber;
       }
mc5    public void debit(float amount) {
s6       setBalance(getBalance() + amount);
       }
mc7    public void credit(float amount) throws
       InsufficientBalanceException {
s8       float balance = getBalance();
s9       if (balance < amount) {
s10        throw new InsufficientBalanceException
           ("Total balance not sufficient");
s11      } else {
s12        setBalance(balance - amount);
         }
       }
mc13   public float getBalance() {
s14      return _balance;
       }
mc15   public void setBalance(float balance) {
s16      _balance = balance;
       }
   }
ce17 public class SavingsAccount extends Account {
mc18   public SavingsAccount(int accountNumber) {
s19      super(accountNumber);
       }
mc20   public void debit(float amount) {
s21      setBalance(getBalance() + amount + 1);
       }
mc22   public static void main(String[] args) throws
       InsufficientBalanceException {
s23      SavingsAccount account = new SavingsAccount(12456);
s24      account.debit(100);
s25      account.credit(50);
       }
   }
ce26 class InsufficientBalanceException extends Exception {
mc27   public InsufficientBalanceException(String message) {
s28      super(message);
       }
   }
ase29 public aspect MinimumBalanceRuleAspect {
ai30   private float Account._minimumBalance;
aa31   private static int throwInsufficientCount = 0;
ma32   public static void getCount() {
s33      System.out.println(throwInsufficientCount);
       }
mi34   public float Account.getAvailableBalance() {
s35      return getBalance() - _minimumBalance;
       }
pe36   pointcut newSavingsAccount(Account account):
       execution(SavingsAccount.new(..)) ||
       execution(*Account.debit(*))) && this(account);
pe37   pointcut throwInsufficientBalance(Account account, float
       amount): execution(* Account.credit(*)) && this(account)
       && args(amount);
ae38   after(Account account) : newSavingsAccount(account) {
s39      account._minimumBalance = 25;
       }
ae40   before(Account account, float amount) throws
       InsufficientBalanceException :
       throwInsufficientBalance(account, amount) {
s41      if (account.getAvailableBalance() < amount) {
s42        throwInsufficientCount++;
s43        getCount();
s44        throw new InsufficientBalanceException
           ("Insufficient available balance");
         }
       }
   }
```

Fig. 1. An example of AspectJ program

Figure 1 shows an AspectJ program from [13]. The program contains one aspect *MinimumBalanceRuleAspect* and three classes *Account*, *SavingsAccount*, and *InsufficientBalanceException*. The aspect owns one method *getCount()* and one attribute *throwInsufficientCount*. It adds one method *getAvailableBalance()* and one attribute *_minimumBalance* to the class *Account* and also introduces two advices related to poincuts *newSavingAccount* and *throwInsufficientBalance* respectively. We apply this example to show the idea of understandability effort measurement in the next section.

4 Dependence Graphs for Aspect-Oriented Software

We measure the effort required for understanding aspect-oriented software based on *aspect-oriented software dependence graph* (ASDG) adapted from [6, 7, 15]. There are three levels of dependency graphs representing an aspect-oriented system, i.e., module-level, class/aspect-level, and system-level. To produce the ASDG of aspect-oriented program, we construct the *software dependence graph* (SDG) for non-aspect code from the *class dependence graphs* (CDGs) containing *method dependence graph*s (MDGs) first, then construct the *aspect interprocedural dependence graphs* (AIDGs) to represent aspects from *advice dependence graphs* (ADGs), *introduction dependence graphs* (IDGs), *pointcut dependence graphs* (PDGs) and *method dependence graphs*. Finally, we determine weaving points between SDG and AIDGs to form the ASDG. The rest of this section explains these dependency graphs from [6, 15] and describes the graph modification points to be more suitable for understandability effort measurements.

4.1 Module-Level Dependence Graphs

In aspect-oriented systems, an aspect contains several types of module, i.e., *advice*, *intertype declaration*, *pointcut*, and *method*, and a class contains only one type of module called *method* [16]. In this paper, we apply three types of module-level dependence graphs: *method dependence graph* (MDG), *advice dependence graph* (ADG), and *introduction dependence graph* (IDG) to represent method, advice, and method introduction respectively, and introduce *pointcut dependence graph* (PDG) to show a pointcut.

The MDG is an arc-classified digraph whose vertices represent statement or predicate expressions in the method. An MDG also includes formal parameter vertices and actual parameter vertices to model parameter passing between models. A *formal-in vertex* and a *formal-out vertex* are used to show a formal parameter of the method and a formal parameter that may be modified by the method. An *actual-in vertex* and an *actual-out vertex* are used to present an actual parameter at call site and an actual parameter that may be modified by the called method. There are two types of arcs representing dependence relationships in the graph, i.e., *control dependence*, *data dependence*, and *call dependence*. Control dependence represents control conditions

on which the execution of a statement or expression depends in the method. It is used to link between method and statement, statement and statement, method and formal parameter, and method and actual parameter. Data dependence represents the data flows between statements in the method. It is used to connect statement and formal parameter and to join statement and actual parameter. Call dependence represents call relationships between statements of a call method and the called method. The examples of MDG are shown as parts of CDGs in Figure 2. The ADG and IDG are constructed with the similar notations with the MDG. ADGs and IDGs are displayed as parts of Figure 4.

For the alteration of module-level dependence graphs, we add *parameter in signature vertices* to the MDG and IDG to represent parameter-in and parameter-out defined in the signature of a method and introduce *parameters in signature of method dependence* arc to link the method or the method introduction with parameters in their signature. We exclude local variables in order to decrease the graphs' complexity. We also omit control dependence arcs between methods and formal parameters/actual parameters in the situation that data dependence is linked between method's statement and formal/actual parameters to avoid redundant metric's count. We show these control dependence arcs only in case there is no explicit relationship between statements of a method and its parameters.

Moreover, we add PDGs to represent poincuts in an aspect-oriented program. In PDGs, the *pointcut vertices* are added to model the poincuts in the program. Each pointcut has its own *parameter in signature vertices.* We also link pointcut vertices and *blank vertices* using *crosscutting dependence arcs* to show possible joinpoints that the poincuts will crosscut.

4.2 Class/Aspect-Level Dependence Graph

The *class dependence graph* (CDG) and the *aspect interprocedural dependence graph* (AIDG) are used to depict a single Java class and a single aspect in the program respectively. Figure 2 shows CDGs of class Account, class SavingsAccount, and class InsufficientBalanceException. The CDG is a digraph consisting of a collection of MDGs which each represents a single method in the class. In this level, *parameter dependence arcs* are added to connect actual-in and formal-in, and formal-out and actual-out vertices to model parameter passing between the methods in the class. The *class membership dependence arcs* are added to show that each method is a member of the class. The methods or attributes of superclasses are also inherited to the subclass.

For the CDG, we add *attribute of class vertices* to model member variables of a class. The vertices are linked to the *class vertex* by class membership dependence arc. We also add call dependence arcs to *blank vertices* to represent call relationships between vertices inside the class and vertices outside the class and add parameter dependence arcs to blank vertices to represent parameter passing between internal class's methods and external class's methods. We did not draw classes outside the scope of source code such as API classes.

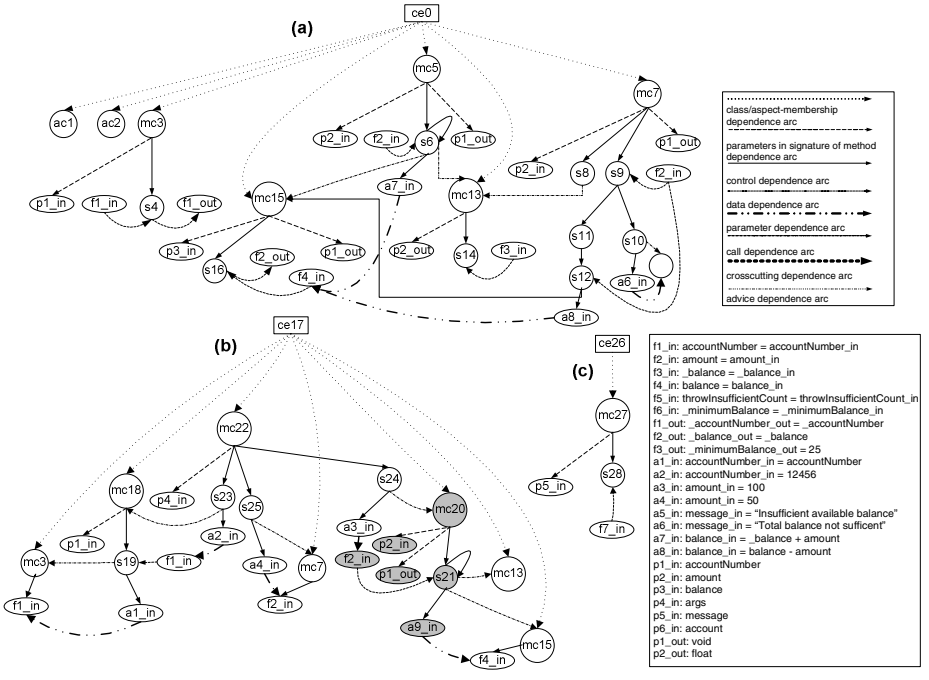

Fig. 2. (a) A CDG for class Account, (b) A CDG for class SavingAccount, and (c) A CDG for class InsufficientBalanceException

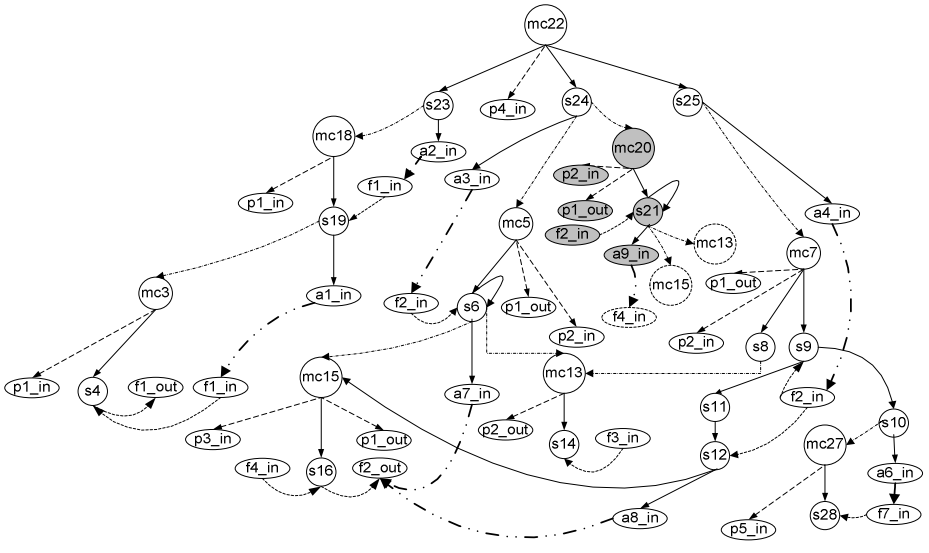

Fig. 3. An SDG for non-aspect code

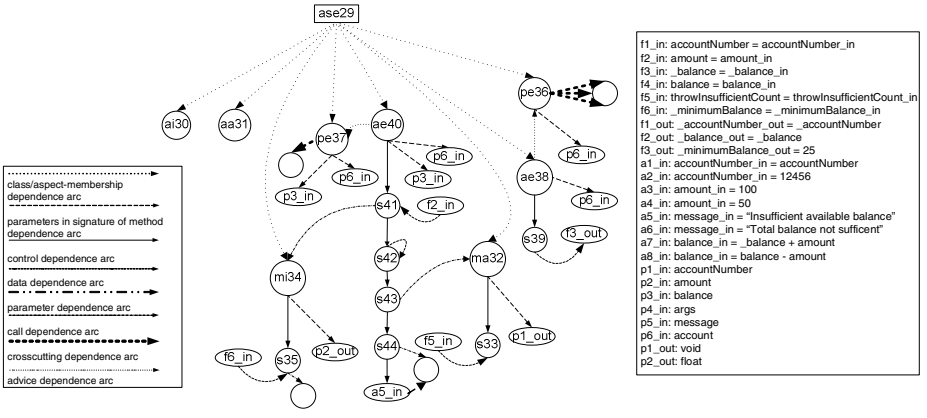

Fig. 4. An AIDG for aspect MinimumBalanceRule

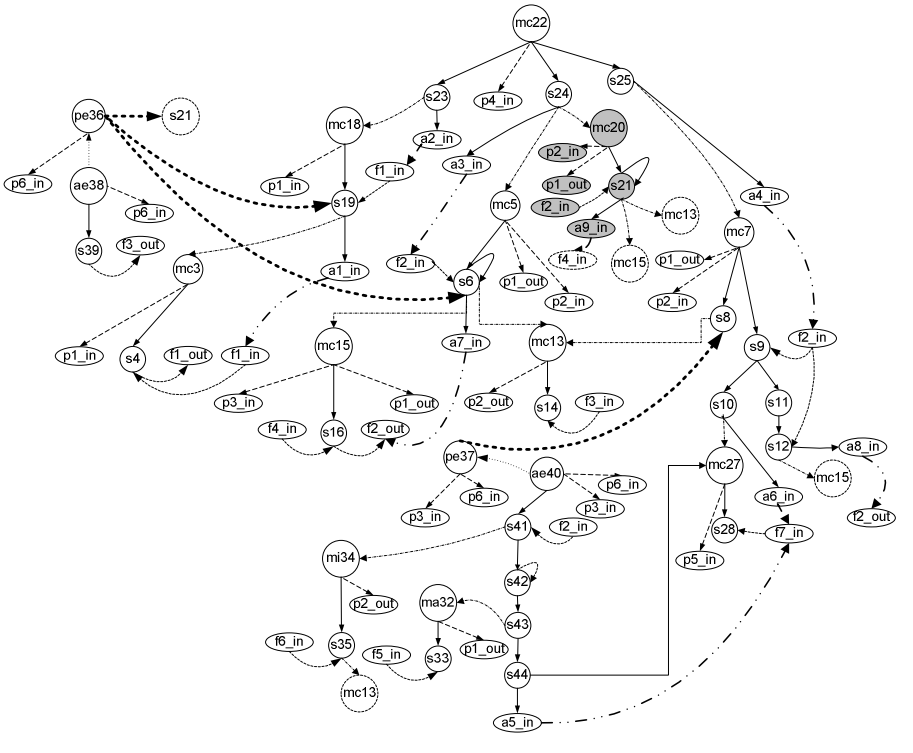

Fig. 5. An ASDG for aspect-oriented program in Figure 1

The AIDG is a digraph that consists of a number of ADGs, IDGs, PDGs, and MDGs. The structure of AIDG is similar to CDG. Figure 4 depicts an AIDG for MinimumBalanceRuleAspect of program in Figure 1. The *aspect membership dependence arcs* are used to show memberships in the aspect. For the modifications of AIDG, we add *attribute of aspect vertices* and *attribute introduction vertices* to represent member variables or attribute introductions of an aspect. Like CDG, we put in call dependence arcs to *blank vertices* to represent call relationships between inside class vertices and outside class vertices and add parameter dependence arcs to blank vertices to represent parameter passing between internal class's methods and external class's methods. We also add *advice dependence arcs* to show the relationship between ADGs and their PDG.

4.3 System-Level Dependence Graphs

Graphs in this level consist of the *software dependence graphs* (SDG) for a Java program before weaving and the *aspect-oriented system dependence graph* (ASDG) for complete aspect-oriented program.

An SDG combines a collection of MDGs from each class in the non-aspect code. The graph starts from the main() method, then the main method calls other methods. In this level, a SDG explicitly shows the relationships between methods in different classes, but cut off the class vertices and class membership dependence arcs to clearly show flows of calling between methods. An example of SDG is displayed in Figure 3. Figure 5 presents an ASDG for complete aspect-oriented program in Figure 1. An ASDG are assembled from an SDG for non-aspect code and AIDGs. In the ASDG, crosscutting relationships between the joinpoints in non-aspect code and the pointcuts are explicitly shown.

5 Metrics for Understandability

In this section, on the assumption that the software understanding effort can be an indicator of understandability measurement, we define some metrics for understandability based on each level of dependency graphs, i.e., module-level, class/aspect-level, and system-level. First of all, we will mention about a guideline for applying our metrics. The proposed metrics are composed from various types of dependency arcs. Each type of arcs has its own level of comprehension. Then, each metric is computed as follows:

$$U = w_1 * NOAs_1 + w_2 * NOAs_2 + \ldots + w_n * NOAs_n$$

where U is the understandability metric value, w_i is a weight value for each type of dependencies, and NOA_n is number of appearances of arcs in the same type.

We are not going to discuss about the weight value here, just simply suppose that the weight values are determined by the experts. In each metric described afterwards, we assume that all weights are equals to 1 so we except to show weighted variables in the equations.

5.1 Module-Level Metrics

Metrics in this level are defined based on the MDGs for methods, ADGs for advices, IDGs for method introductions, and PDGs for pointcuts. Attached to each definitions, we illustrate our measurement examples based on program and graphs (not included the grey highlights) in Figure 1-5.

To understand each *method* or *method introduction*, we have to understand its parameters, its statements in the method/method introduction, the called methods, and the parameters of each statement (both formal and actual parameters). So we define understandability efforts for MDG and IDG as:

$U_{MDG/IDG}$* = the number of *control dependence arcs* + the number of *data dependence arcs* + the number of *parameter in signature of method dependence arcs* + the number of *call dependence arcs

For each *advice*, we have to comprehend its parameters, its statements in the method/method introduction, the called methods, the parameters-in/out of each statement, and the pointcuts that the advice depends on. The effort for understanding ADG is calculated as:

U_{ADG}* = the number of *control dependence arcs* + the number of *data dependence arcs* + the number of *parameter in signature of method dependence arcs* + the number of *call dependence arcs* + the number of *advice dependence arcs

To comprehend each pointcut, we have to know its parameters and its selected join points. Hence, we have the following metric:

U_{PDG}* = the number of *parameter in signature of method dependence arcs* + the number of *crosscutting dependence arcs

For example in Figure 2, $U_{MDG:mc3} = 4$, $U_{MDG:mc5} = 8$, $U_{MDG:mc7} = 14$, $U_{MDG:mc13} = 3$, $U_{MDG:mc15} = 5$, $U_{MDG:mc18} = 5$, $U_{MDG:mc22} = 10$, and $U_{MDG:mc27} = 3$. In Figure 4, $U_{MDG:ma33} = 3$, $U_{IDG:mi34} = 4$, $U_{PDG:pe36} = .2$, $U_{PDG:pe37} = 3$, $U_{ADG:ae38} = 4$, and $U_{ADG:ae40} = 12$.

5.2 Class/Aspect-Level Metrics

We define some class/aspect level metrics grounded on CDGs for classes and AIDGs for aspects. In this level, the relationships between modules are explicitly shown, so we add counts of class membership dependencies and parameter dependencies to the measurements. Under the assumption that we use effort only once to understand repeated called methods, the redundant number of calls to the same method are omitted from the equations. Then, the metric for CDG understandability are defined as:

***U_{CDG}* = summation of *U_{MDGs}* for all inherited methods and methods in the class + the number of *class membership dependence arcs* + the number of *parameter dependence arcs* – the number of redundant *call dependence arcs* count** or

***U_{CDG}* = the number of *all dependence arcs* in CDG – the number of redundant *call dependence arcs* count in CDG**

The efforts for understanding ADG are calculated as:

***U_{AIDG}* = summation of *U_{MDGs}* for all inherited methods and methods in the aspect + summation of *U_{ADGs}* for all advices in the aspect + summation of *U_{PDGs}* for all poincuts in the aspect + the number of *aspect membership dependence arcs* + the number of *parameter dependence arcs* – the number of redundant *call dependence arcs* count** or

***U_{AIDG}* = the number of *all dependence arcs* – the number of redundant *call dependence arcs* count**

For instance in Figure 2, $U_{CDG:ce0} = 42$, $U_{CDG:ce17} = 60$, and $U_{CDG:ce26} = 4$. In Figure 4, $U_{AIDG:ase29} = 38$.

5.3 System-Level Metrics

Lastly, we propose metrics at the system level base upon the SDG for non-aspect code and ASDG for the whole aspect-oriented system. The metric for Java based code are proposed as follows:

***U_{SDG}* = summation of *U_{CDGs}* for all classes in non-aspect code – summation of redundant *U_{MDGs}* count in all CDGs - the number of redundant *call dependence arcs* count not shown in CDGs -** or

***U_{SDG}* = the number of *all dependence arcs* + the number of *class membership dependence arcs* in all CDGs – the number of redundant *call dependence arcs* count**

The last metric is used to measure total understandability effort of an aspect-oriented program. The metric are measured as:

***U_{ASDG}* = *U_{SDG}* for non-aspect code + summation of *U_{AIDGs}* for all aspects - the number of redundant *call dependence arcs* count not shown in SDG and AIDGs** or

***U_{ASDG}* = the number of *all dependence arcs* + the number of *class membership dependence arcs* in all CDGs + the number of *aspect membership dependence arcs* in all AIDGs – the number of redundant *call dependence arcs* count**

Then the metric in Figure 3, $U_{SDG} = 72$. In Figure 5, $U_{ASDG} = 109$.

5.4 An Example

The measurement examples above are based on the program in Figure 1 that is not included the grey parts. After alteration program by adding code in the grey highlights (adding one method debit() to class *SavingAccount* and modifying the set of join points in pointcut *newSavingAccount*), the measurement values of all metrics are as following. The values of metrics that have been changed are represented in bold face.

For module-level, the metrics in Figure 2: $U_{MDG:mc3} = 4$, $U_{MDG:mc5} = 8$, $U_{MDG:mc7} = 14$, $U_{MDG:mc13} = 3$, $U_{MDG:mc15} = 5$, $U_{MDG:mc18} = 5$, $\mathbf{U_{MDG:mc20} = 8}$, $U_{MDG:mc22} = 10$, and

$U_{MDG:mc27}$ = 3. In Figure 4: $U_{MDG:ma33}$ = 3, $U_{IDG:mi34}$ = 4, $\boldsymbol{U_{PDG:pe36}}$ **=.4**, $U_{PDG:pe37}$ = 3, $U_{ADG:ae38}$ = 4, and $U_{ADG:ae40}$ = 12. For class/aspect-level, the metrics in Figure 2: $U_{CDG:ce0}$ = 42, $\boldsymbol{U_{CDG:ce17}}$ **= 60**, and $U_{CDG:ce26}$ = 4. In Figure 4: $\boldsymbol{U_{AIDG:ase29}}$ **= 40**. Finally, the metric in system-level in Figure 3: $\boldsymbol{U_{SDG}}$ **= 78**. In Figure 5: $\boldsymbol{U_{ASDG}}$ **= 117**.

6 Conclusion and Future Work

This paper proposes seven objective metrics for evaluating understandability of aspect-oriented software from the understanding effort used. These metrics are based on three levels dependency graphs mapping from aspect-oriented program code, i.e., MDGs, ADGs, IDGs, and PDGs in module-level, CDGs and AIDGs in class/aspect-level, and SDG and ASDG in system-level. They are composed from summarizing number of each type of dependency arcs in the graphs. The measures are applied to two versions of aspect-oriented programs to give an illustrative example. For further research, we plan to find more objective metrics that are related to the understandability or other maintainability sub-characteristics. In addition, the proposed metrics should be also explored their thresholds to be used as a guideline for the result assessment.

References

1. Page-Jones, M.: The Practical Guide to Structured System Design. Yourdon Press, New York (1980)
2. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. International Thomson Computer Press (1996)
3. Informatics Institute, `http://www.ii.metu.edu.tr/~ion502/demo/ch1.html`
4. Shima, K., Takemura, Y., Matsumoto, K.: An Approach to Experimental Evaluation of Software Understandability. In: International Symposium on Empirical Software Engineering (ISESE 2002), pp. 48–55 (2002)
5. Stein, D., Hanenberg, S., Unland, R.: A UML-based Aspect-Oriented Design Notation. In: 1st International Conference on Aspect-Oriented Software Development, pp. 106–112 (2002)
6. Zhao, J.: Dependence Analysis of Aspect-Oriented Software and Its Applications to Slicing, Testing, and Debugging. Technical-Report SE-2001-134-17, Information Processing Society of Japan, IPSJ (2001)
7. Zhao, J.: Towards a Metrics Suite for Aspect-Oriented Software. Technical-Report SE-136-25, Information Processing Society of Japan, IPSJ (2002)
8. Zhao, J., Xu, B.: Measuring aspect cohesion. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 54–68. Springer, Heidelberg (2004)
9. Jindasawat, N., Kiewkanya, M., Muenchaisri, P.: Investigating Correlation between the Object-Oriented Design Maintainability and Two Sub-Characteristics. In: 13th International Conference on Intelligent & Adaptive Systems, and Software Engineering (IASSE 2004), pp. 151–156 (2004)
10. Sheldon, F.S., Jerath, K., Chung, H.: Metrics for maintainability of class inheritance hierarchies. Journal of Software Maintenance and Evolution: Research and Practice 14, 147–160 (2002)

11. Genero, M., Piatini, M., Manso, E.: Finding "Early" Indicators of UML Class Diagrams Understandability and Modifiability. In: International Symposium on Empirical Software Engineering (ISESE 2004), pp. 207–216 (2004)
12. Gregor, K., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
13. Ramnivas, L.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications (2003)
14. Guyomarc'h, J.Y., Guéhéneuc, Y.G.: On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics. In: 9th ECOOP workshop on Quantitative Approaches in Object-Oriented Software Engineering, pp. 42–47 (2005)
15. Zhao, J.: Applying Program Dependence Analysis to Java Software. In: Workshop on Software Engineering and Database Systems, International Computer Symposium, pp. 162–169 (1998)
16. Zhao, J.: Measuring Coupling in Aspect-Oriented Systems. In: 10th International Software Metrics Symposium (2004)