CHAPTER III

THESIS OVERVIEW AND A COLLECTION OF DESIGN PRINCIPLES

Chapter III illustrates an overview of the thesis. It describes all collaborating components and links those components into the big picture. This chapter also presents the aspect-oriented guidelines which are extracted and are defined to fulfill the aspect-oriented software maintainability measurement. They support both *high level design* (e.g. logical functions) and *detailed design* (e.g. pseudocode). Fifteen guidelines are classified into 4 categories: *the design guidelines for pointcuts and advices*, *the design guidelines for aspects*, *the design guidelines for the relationships between aspects and other components*, and *the design guidelines for inheritance relationships*. In addition, this chapter describes a set of collected and filtered design principles.

3.1 Thesis Overview

Figure 3.1 shows an overview of constructing aspect-oriented software maintainability metrics. Firstly, aspect-oriented design guidelines are presented and a collection of design heuristics and bad smells are selected and are sorted out in Chapter III. Secondly, the design guidelines along with the rest design heuristics and bad smells are extracted their violation check definitions and are validated the definitions using a set of example systems developed from the principle sources in Chapter IV. Chapter V shows applications and some concerned points of applying design principle violation check definitions. Thirdly, the design principles are explored their relationships to the aspect-oriented software maintainability to select the related design principles, are weighted their influences on maintainability, and are combined to construct the aspect-oriented maintainability metrics in Chapter VI. Finally, Chapter VII shows the metric utilization. These metrics are used to evaluate fifty aspect-oriented systems and to compare maintainability of two systems written in AspectJ and in Java. These applications show that the maintainability metrics can evaluate and compare both aspect-oriented software and object-oriented software.

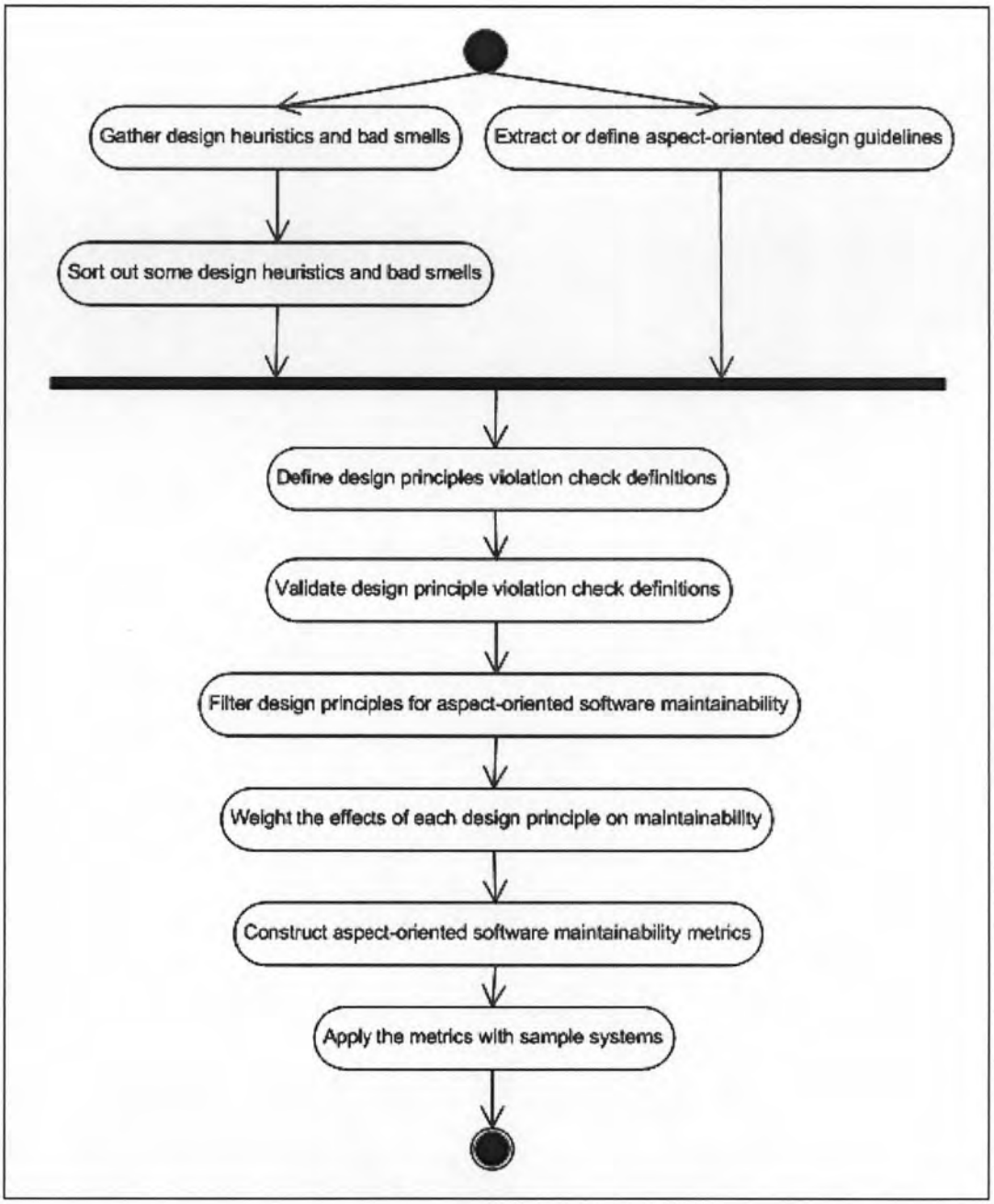

Figure 3.1: Activity diagram for aspect-oriented software maintainability metrics construction.

3.2 Aspect-Oriented Design Guidelines

This section extracts or defines aspect-oriented design guidelines in terms of descriptions, effects of violating them, and illustrations of breaking and following them to support evaluating aspect-oriented software. Guideline 5, Guideline 7, Guideline 8, Guideline 10, and Guideline 14 are extracted from some research studies. The rest

guidelines are newly defined from considering aspect-oriented programming concepts, aspect-oriented software characteristics, and some object-oriented design principles. Fifteen guidelines are categorized into 4 groups: *the design guidelines for pointcuts and advices*, *the design guidelines for aspects*, *the design guidelines for the relationships between aspects and other components*, and *the design guidelines for inheritance relationships*.

3.2.1 Design Guidelines for Pointcuts and Advices

Guideline 1: Don't make the core concern unused by replacing the entire of its context with the crosscutting concern.

Description: *Around advice* can modify the execution of the core concern that is at the join point, it can replace it, or it can even bypass it. However, refusing the core concern behavior by replacing the entire core concern context with the crosscutting concern will make the core concern unused. Thus, the crosscutting concern should continue the original execution of the core concern instead of changing the overall core concern context.

Violation Effects: A violation of the guideline will make the core concern to be useless. Furthermore, completely altering the core concern's behavior will automatically block the execution of lower precedence aspects which crosscut the same core concern.

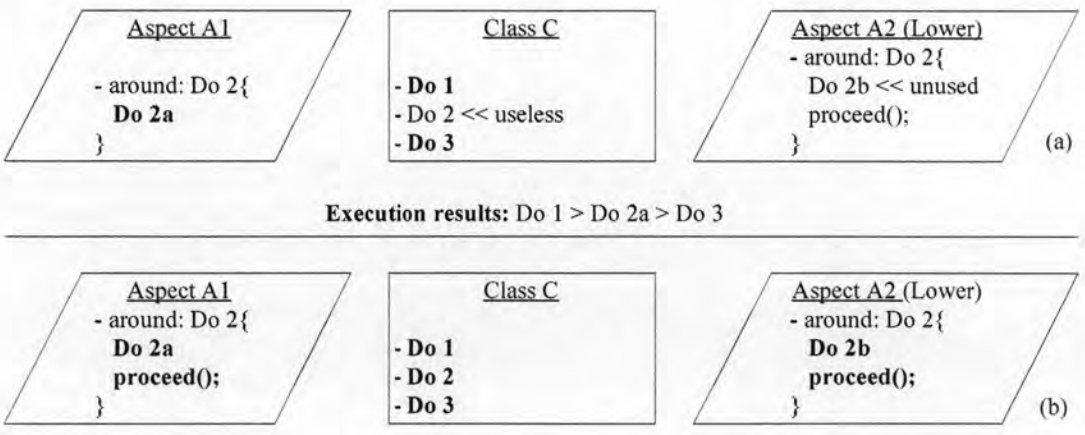

Figure 3.2: (a) A system violating the Guideline 1 and (b) a system following the Guideline 1

Illustrations: Figure 3.2 shows a system consisting of class *C*, aspect *A1*, and aspect *A2*. Aspect *A1* and aspect *A2* surround the same join point of class *C* that is behavior *Do2*. Nevertheless, in Figure 3.2(a), aspect *A1* which has higher precedence than aspect *A2* did not call *proceed()*. This violation prevents aspect *A2* from execution and causes behavior *Do2* useless. Figure 3.2(b) corrects the violation by calling *proceed()* in aspect *A1*. This correction leads the system to perform the expected sequence of behaviors.

Guideline 2: Do suitably use wildcard operators for each captured join point to increase the pointcut readability and to decrease the unintended join point capture.

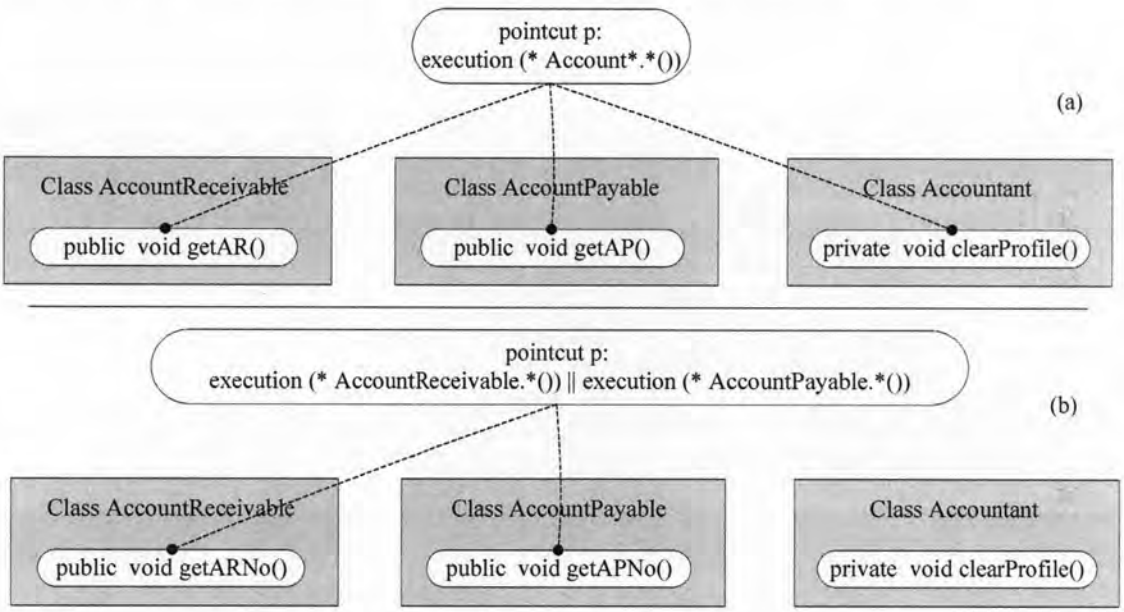

Figure 3.3: (a) A system violating the Guideline 2 and (b) a system following the Guideline 2

Description: A *captured joint point* is a simple join point which can be combined with other join points and binary operators (||, &&) to form a pointcut. Each captured join point should contain at most two operators (asterisks plus dotted lines). A*sterisk* (*) and *dotted line* (..) wildcard operators are used to denote characters. Although, using the wildcard operators can decrease the pointcut size, the more wildcard operators used can increase the more possibility of unexpected join point captures. Each captured join point should contain at most two asterisk and dotted line operators. The number two is

taken from the average number of the wildcard operators used in each capture join point of fifty sample systems in Section 7.1.

Violation Effects: A violation of the guideline increases the difficulty of specifying a pointcut scope. In addition, using too many wildcard operators may cause an unintended join point capture after the base program was changed. The *unintended join point capture* occurs when a pointcut accidentally catches a join point which should not be captured.

Illustrations: Figure 3.3 shows a system consisting of class *AccountReceivable*, class *AccountPayable*, and class *Accountant* which is newly added after changing requirements. At first, pointcut *p* captures the execution of *AccountReceivable's public method* and *AccountPayable's public method*. After adding class *Accountant*, pointcut *p* accidentally captures the execution of *Accountant's private methods* as shown in Figure 3.3(a). Figure 3.3(b) presents pointcut *p* which eliminates some wildcard operators. Specifying the clear class names instead can help the right join points to be captured by the pointcut.

Guideline 3: Do define a concise pointcut for capturing sibling join points to decrease the accidental join point miss when a new class is added to the hierarchy.

Description: *Sibling join points* mean join points which catch similar things in different sibling classes. Defining a concise pointcut using the *plus* (+) wildcard operator helps developers to easily capture sibling join points. Thus, a pointcut should be defined using the plus operator, common join points, and a superclass name or an interface name instead of the combination of a join point of each sibling class.

Violation Effects: A violation of the guideline increases the probability of accidental join point miss. The *accidental join point miss* happens when a pointcut may no longer capture a join point that should be caught due to an alteration of the base program.

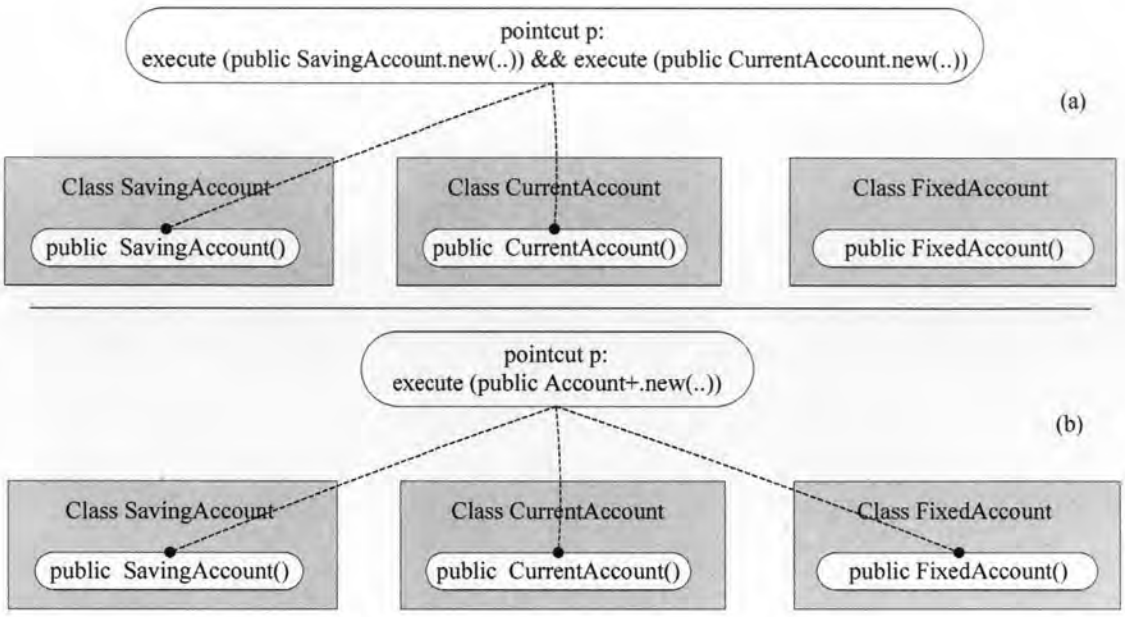

Figure 3.4: (a) A system violating the Guideline 3 and (b) a system following the Guideline 3

Illustrations: Figure 3.4 shows a system consisting of class *SavingAccount*, class *CurrentAccount*, which are subclasses of abstract class *Account*, and class *FixedAccount* which is newly added to be another subclass of abstract class *Account* after changing requirements. At first, pointcut *p* captures the execution of two constructors of *Account's subclasses*. After adding class *FixedAccount*, pointcut *p* fails to capture the execution of *FixedAccount's constructor* as shown in Figure 3.4(a). Figure 3.4(b) presents pointcut *p* which is concisely defined with a plus wildcard operator. Defining the higher level with the superclass name can help the pointcut to correctly capture join points when a new subclass is added.

3.2.2 Design Guidelines for Aspects

Guideline 4: Don't use an aspect privilege to leak the secret of other types.

Description: *Aspect privilege* is an aspect which has the prerogative of accessing protected, package-private (default), or even private resources of others. However, an aspect privilege should be carefully used, especially in accessing the private context because reaching the private members of other types signalizes that the aspect is trying to steal the secret information and is trying to break the encapsulation concept of others.

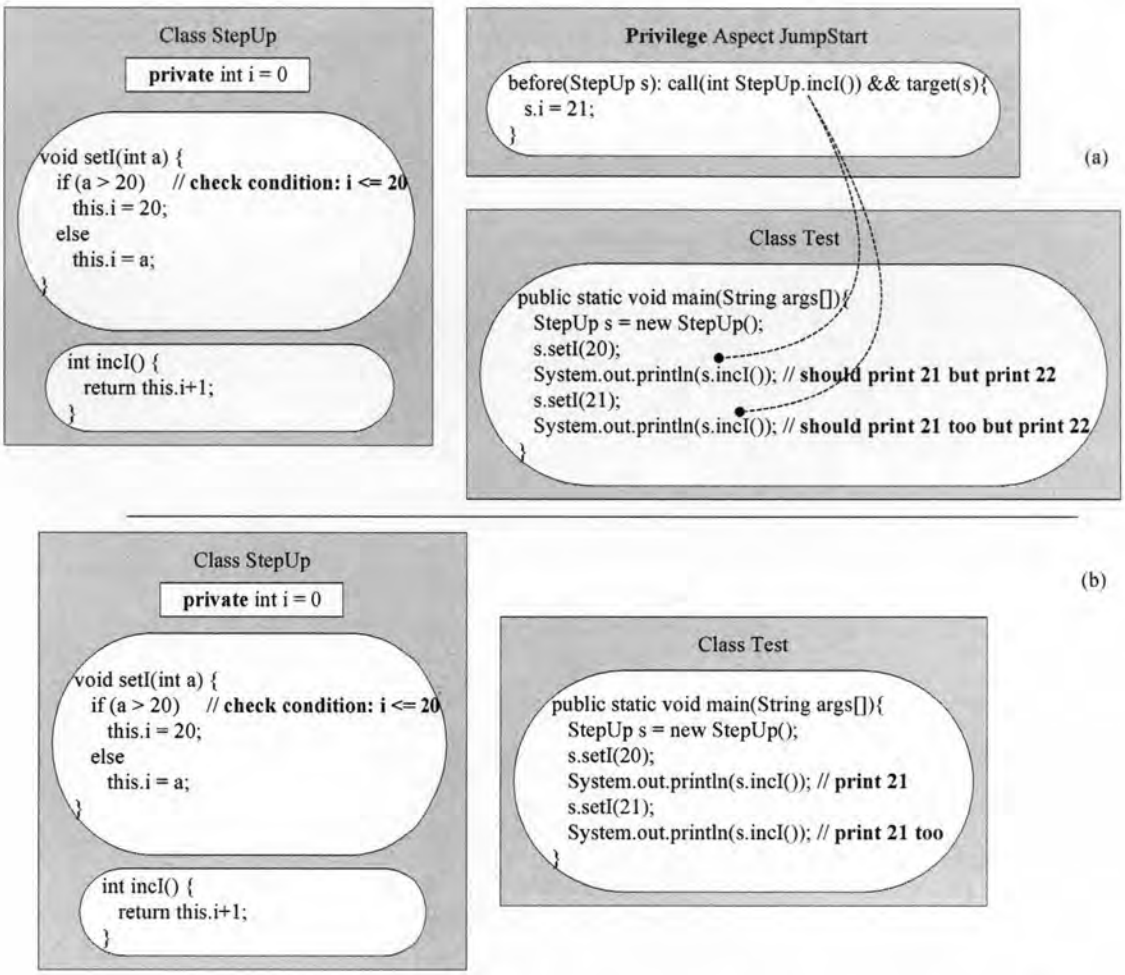

Figure 3.5: (a) A system violating the Guideline 4 and (b) a system following the Guideline 4

Violation Effects: Theoretically, an aspect should be oblivious by the core concern. So, any changes which are made to private parts of the core concern may not be recognized by their owner. Accessing to the private members without awareness of the core concern may affect the other aspects which crosscut to the same concern. A violation of the guideline also leads the base unit and all units which depend on those private parts to perform unexpected behaviors.

Illustrations: Figure 3.5 shows a system consisting of class *StepUp*, class *Test*, and aspect *JumpStart*. Aspect *JumpStart* with privilege can change the private *i* value of class *StepUp* without passing any conditional checks. This prerogative enables class *Test* printing number 22 twice instead of number 21 twice as shown in Figure 3.5(a). Figure 3.5b(b) presents the system which follows the objectives of class *Test* and class

StepUp. Class *Test* correctly prints the number 21 twice. Also, the conditional check of method *setI()*, class *StepUp*, are not broken.

Guideline 5: Do create an advantageous aspect which crosscuts at least two points in the core concern.

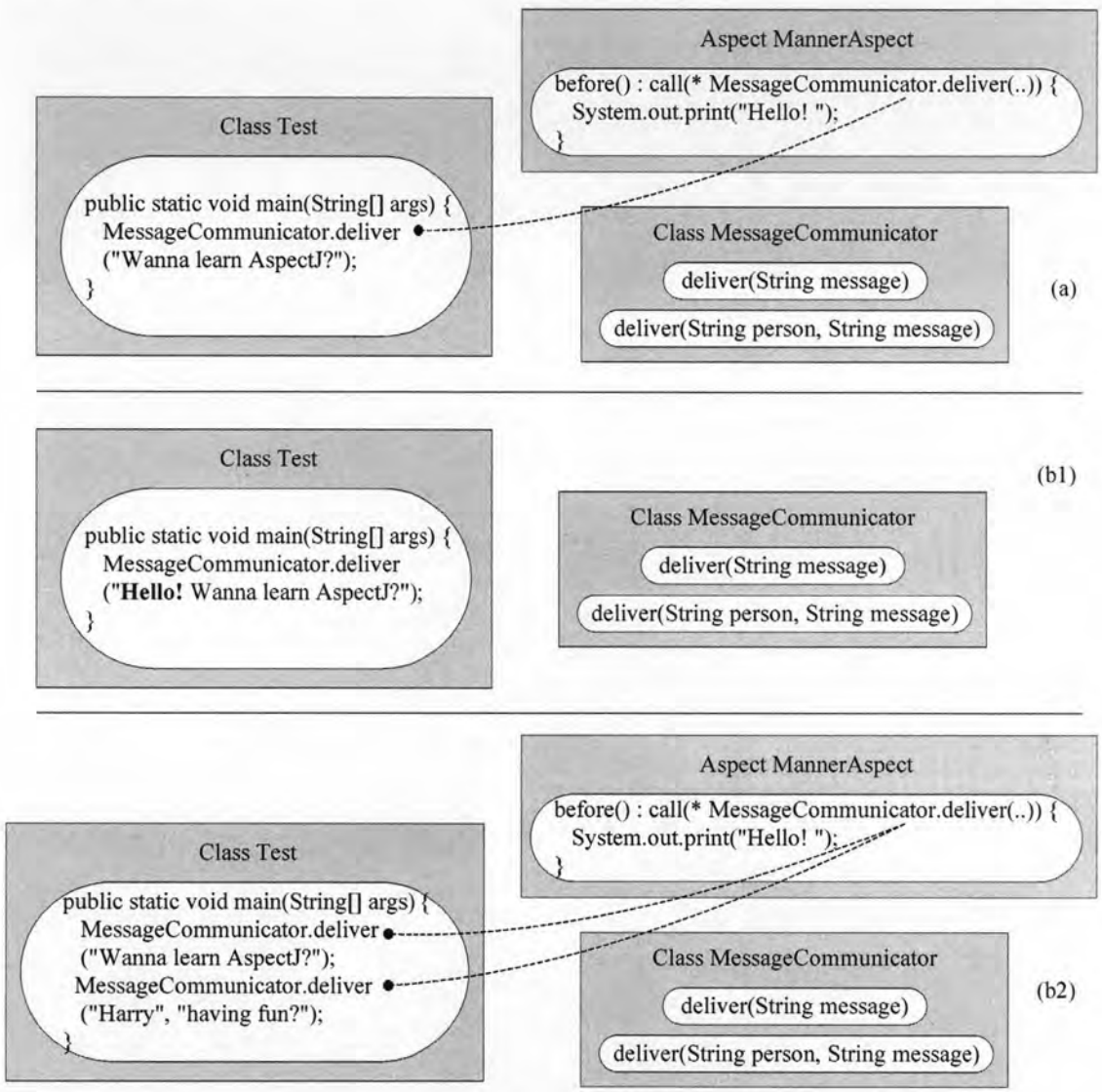

Figure 3.6: (a) A system violating the Guideline 5 and (b1, b2) a system following the Guideline 5

Description: An aspect is aimed to behave a crosscutting concern that tangles and scatters around the system. *Crosscutting contexts* include captured join points, advices, inter-type member declarations, parent declarations, precedence declarations, warning declarations, error declarations, and soft declarations. Anyway, a useful aspect should

contain the crosscutting contexts which crosscut totally more than one place in the core concern.

Violation Effects: This design guideline is given support by the remark: *AOP is not useful for singletons*, which means if you've gotten an orthogonal concern, and you're sure that the orthogonal concern will not propagate to other loci as the system evolves, it is probably a bad idea to use AOP for that concern [29]. A violation of the guideline increases an unnecessary component (aspect), also unnecessary couplings between the aspect and its base class. The aspect which crosscuts less than two places in the core concern should be demoted.

Illustrations: Figure 3.6 shows systems consisting of class *Test*, class *MessageCommunicator*, and aspect *MannerAspect*. Aspect *MannerAspect* crosscuts only one place in class *Test* to add string "Hello!" as shown in Figure 3.6(a). This waste aspect declaration should be fixed by directly adding the string "Hello!" to the base code as displayed in Figure 3.6(b1). Proper aspect utilization for supporting at least two scattering points of concern is shown in Figure 3.6(b2).

3.2.3 Design Guidelines for the Relationships Between Aspects and Other Components

Guideline 6: Do localize all crosscutting contexts within a set of corresponding aspects.

Description: Crosscutting contexts may be scattered or tangled around the system. However, once a crosscutting concern is introduced, the other units should be entirely free of these contexts. A crosscutting concern is modeled as a set of corresponding aspects, so all classes and aspects besides the corresponding aspects should be without the context of the crosscutting concern.

Violation Effects: Placing the scattering behavior only in a crosscutting concern confines the potential changes to that behavior in one area. A violation of the guideline by dispersing a crosscutting concern over the system causes context couplings between the corresponding aspects and other units. Furthermore, this violation requires more modification effort from developers when the scattering behavior changes.

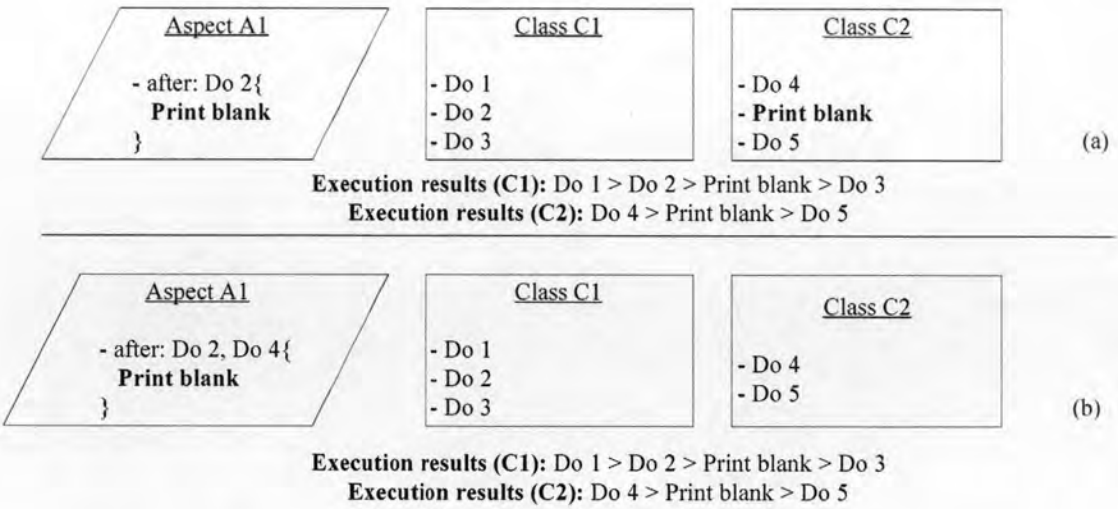

Figure 3.7: (a) A system violating the Guideline 6 and (b) a system following the Guideline 6

Illustrations: Figure 3.7b shows a system consisting of class *C1*, class *C2*, and aspect *A1*. Aspect *A1* crosscuts behavior *Do2* to perform behavior *Print blank* after behavior *Do2*. Class *C2* performs the same behavior *Print blank* after behavior *Do 4*, but it defines its own *Print blank* method. Consequently, this violation forces the developer having to correct two points, as shown in Figure 3.7(a), if behavior *Print blank* is required to be changed to *Print stars*. Confining behavior *Print blank* in one area by replacing *C2*'s *Print blank* with an advice can help the developer to change the system easier as presented in Figure 3.7(b).

Guideline 7: Do try to separate one set of aspects to be independent of another.

Description: *A set of aspects* means an abstract aspect and corresponding aspects that are used to model a crosscutting concern. Separating two sets of aspects absolutely supports developing multi-dimensional separation of concerns and reusing those sets of aspects [30].

Violation Effects: Calling to another aspect in the different hierarchy indicates the dependency between two aspects. A violation of the guideline makes designing or altering one set of depending aspects cannot be done separately from another set of dependent aspects.

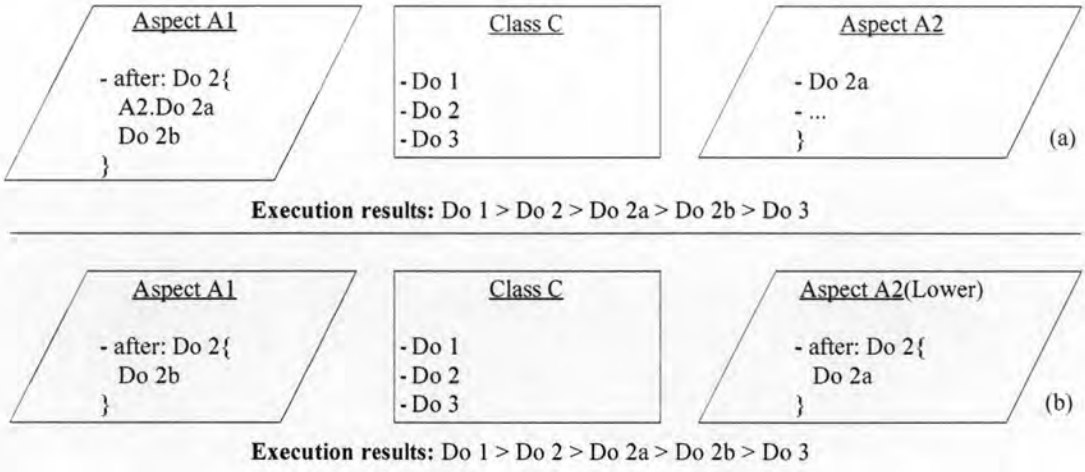

Figure 3.8: (a) A system violating the Guideline 7 and (b) a system following the Guideline 7

Illustrations: Figure 3.8 shows a system consisting of class *C*, aspect *A1*, and aspect *A2*. Figure 3.8(a) shows aspect *A1* which crosscuts class *C*, calls to behavior *Do2a* of aspect *A2*, and then performs behavior *Do2b*. Aspect *A1* and aspect *A2* dependency causes behavior *Do2a* cannot be changed or be deleted without notification of aspect *A1*. Figure 3.8(b) separates two dependent aspects. This separation promotes reusability of aspect *A1*, also supports an alteration of aspect *A2*.

Guideline 8: Do minimize a number of aspects which crosscut to the same concern.

Description: A concern is *tangled* if both it and at least one of other concerns are related to the same target element [31]. Tangling of two concerns may lead the second-order crosscutting concern to perform the unexpected behavior. So, the number of aspects which crosscut the same concern should be kept to a minimum. None of concern intersections are the most proper.

Violation Effects: A violation of the guideline means an aspect is tangled another aspect, thus given a chance to produce the ripple effects to each other.

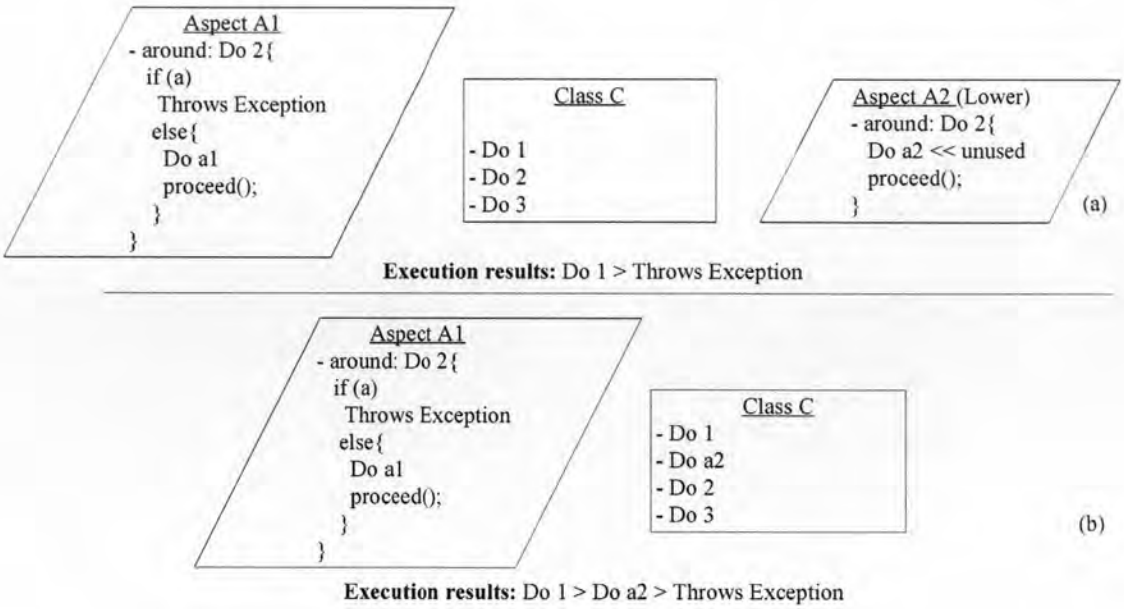

Figure 3.9: (a) A system violating the Guideline 8 and (b) a system following the Guideline 8

Illustrations: Figure 3.9 shows systems consisting of class *C*, aspect *A1*, and aspect *A2*. Figure 3.9(a) shows aspect *A1* which crosscuts behavior *Do2*, checks condition *a*, and throws exception or perform behavior *Doa1* and behavior *Do2*. Nevertheless, throwing exception of aspect *A1* may lead the *around advice* of aspect *A2* which catches the same join point to be accidentally stopped its execution. Figure 3.9(b) eliminates crosscutting to the same join point. This adjustment results that behavior *Doa2* behavior is executed finally.

Guideline 9: Do keep scattering behaviors that represent the same concern in one set of corresponding aspects.

Description: A concern is *scattered* if it relates to multiple target elements. Scattering behaviors can be both duplicate sequence of behaviors and behaviors which are intended to support the crosscutting concern but are not located in the corresponding aspects.

Violation Effects: A violation of the guideline forces developers taking more time to correct changes. Scattering behaviors, which are called by a set of aspects only, but are located in other classes or other aspects, should be moved to those corresponding aspects.

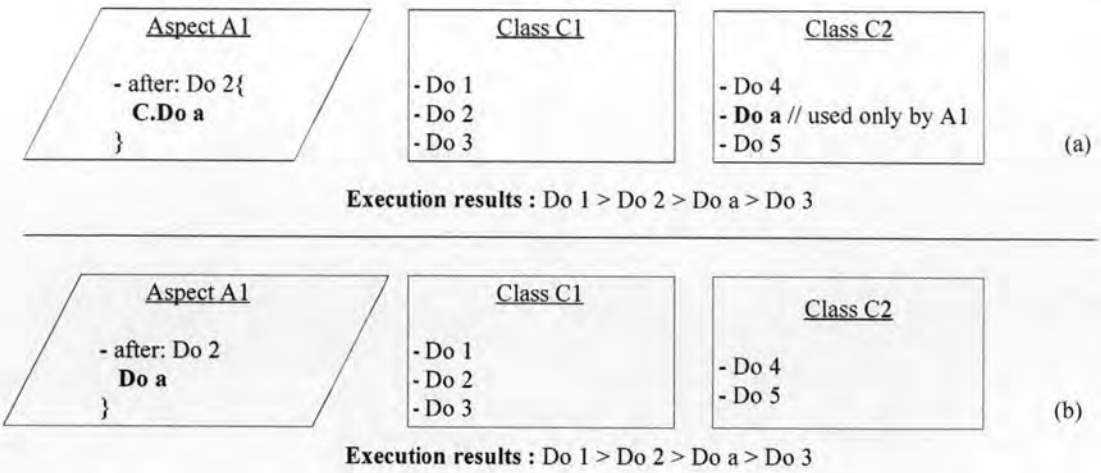

Figure 3.10: (a) A system violating the Guideline 9 and (b) a system following the Guideline 9

Illustrations: Figure 3.10 shows systems consisting of class *C1*, class *C2*, and aspect *A1*. Aspect *A1* crosscuts behavior *Do2* and asks class *C2* to perform behavior *Do a*. Behavior *Do a* of class *C2* is used by aspect *A1* only. Consequently, this scattering forces the developer having to correct two locations when the behavior of the crosscutting concern changes as shown in Figure 3.10(a). Moving behavior *Do a* to aspect *A1* as presented in Figure 3.10(b) help developers to easier change the crosscutting behavior.

Guideline 10: Don't let a base class be dependence on its crosscutting aspects.

Description: Due to the reusability purpose, a base class should not depend on its crosscutting aspects. Also, frequent changes of the crosscutting aspects which are generally unstable will affect the base class depending on them. In addition, this guideline is supported by the notion of *locality* which states that all dependencies between patterns (aspects) and participants (base classes) should be localized in the pattern code [32, 33]. Because of the above reasons, a class may be referred to by one or more aspects, but not vice versa.

Violation Effects: A violation of the guideline decreases the base class reusability. Reusing the base class is required to take the dependent aspects along with the base class.

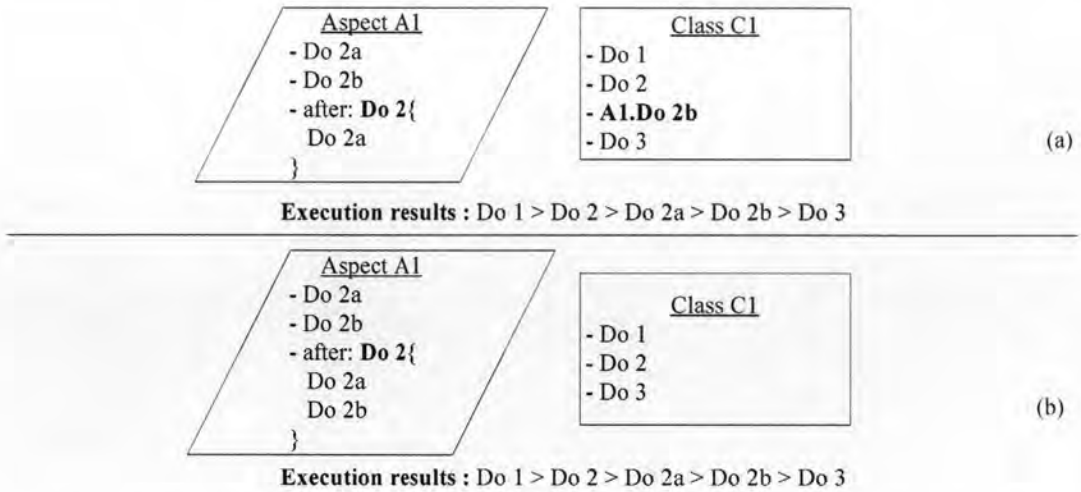

Figure 3.11: (a) A system violating the Guideline 10 and (b) a system following the Guideline 10

Illustrations: Figure 3.11 shows systems consisting of class *C1* and aspect *A1*. In Figure 3.11(a), aspect *A1* crosscuts behavior *Do2*. Class *C1* calls to behavior *Do2a* of aspect *A1*. This dependency causes class *C1* cannot be reused without an attachment of aspect *A1*. Figure 3.11(b) scopes the dependency between aspect *A1* and class *C1* within aspect *A1*. This adjustment helps class *C1* can be used anywhere without aspect *A1*.

3.2.4 Design Guidelines for Inheritance Relationships

Guideline 11: Don't use an inheritance relationship to model a crosscutting relationship because an aspect is not a special type of the class that it crosscuts.

Description: Aspect-oriented software shares the same structure as the object-oriented software. An inheritance relationship is the object-oriented structure which can be applied to the aspect-oriented software as well. Some programming language such as *AspectJ* allows an aspect to inherit a class. Nevertheless, an inheritance relationship is suitable for representing *a-kind-of* relationship. So, an aspect should not extend a class just because the inheritance helps it to access the class members easier.

Violation Effects: Because the aspect is not a special type of the class, a violation of the guideline confuses the system users about the real behavior of the aspect. Also, it increases inappropriate dependency between the aspect and the class.

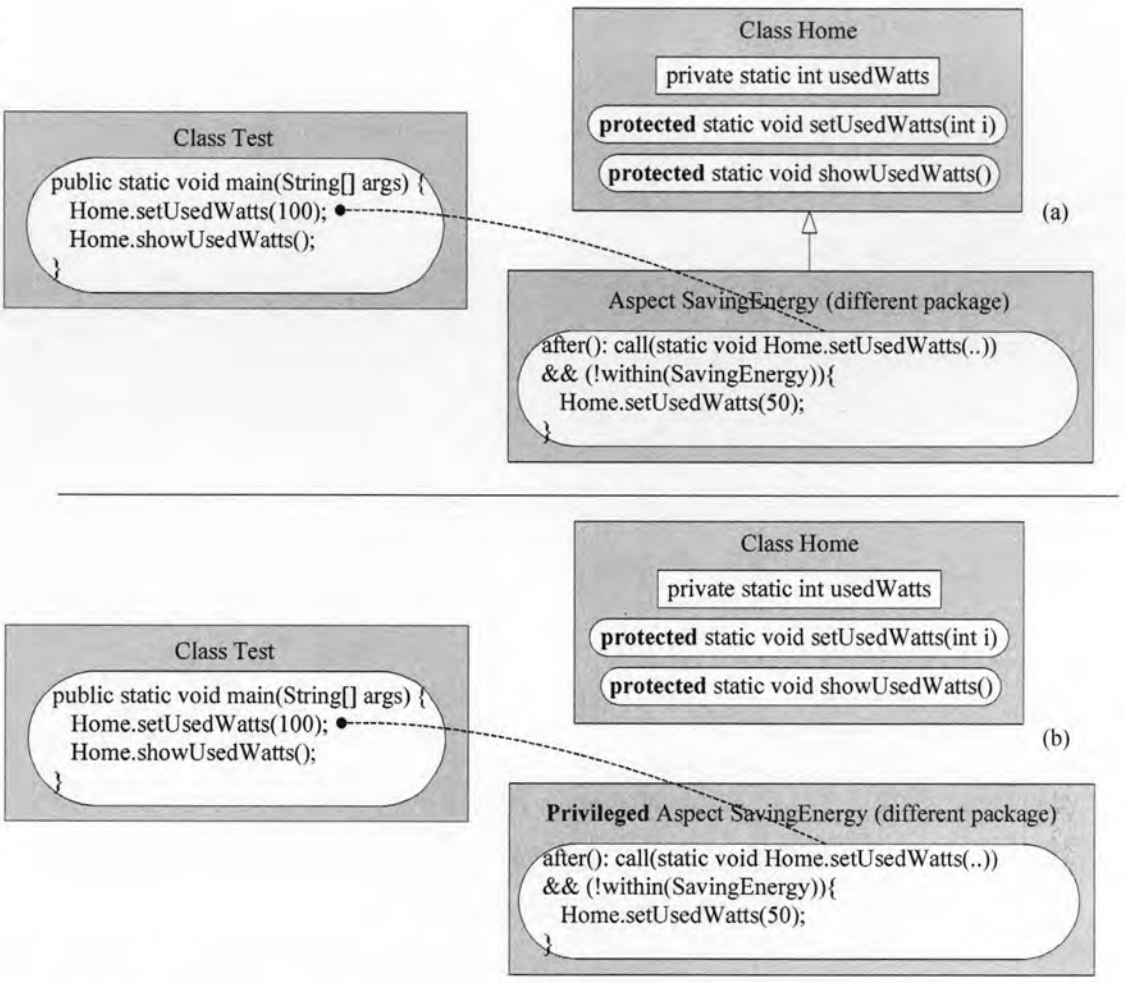

Figure 3.12: (a) A system violating the Guideline 11 and (b) a system following the Guideline 11

Illustrations: Figure 3.12 shows systems consisting of class *Test*, class *Home*, and aspect *SavingEnergy* which is stored in the different package. In Figure 3.12(a), aspect *SavingEnergy* crosscuts class *Test*, extends class *Home*, and calls to the protected method *setUsedWatts* to change the value of field *usedWatts*. However, aspect *SavingEnergy* is not a kind of class *Home*. Thus, Figure 3.12(b) gives the privilege to aspect *SavingEnergy*, but the aspect is controlled to call to the protected method of class *Home* only.

Guideline 12: Don't introduce methods to all subclasses to override a concrete method of their abstract superclass.

Description: A method in the aspect-oriented software can override the inherited method as same as a method in the object-oriented software. However, introducing

inter-type methods to all subclasses to override the concrete method of the abstract superclass will cause that concrete method unused.

Violation Effects: A violation of the guideline causes a concrete method of the abstract superclass unused. Moreover, it indicates that some part of the system is wasted.

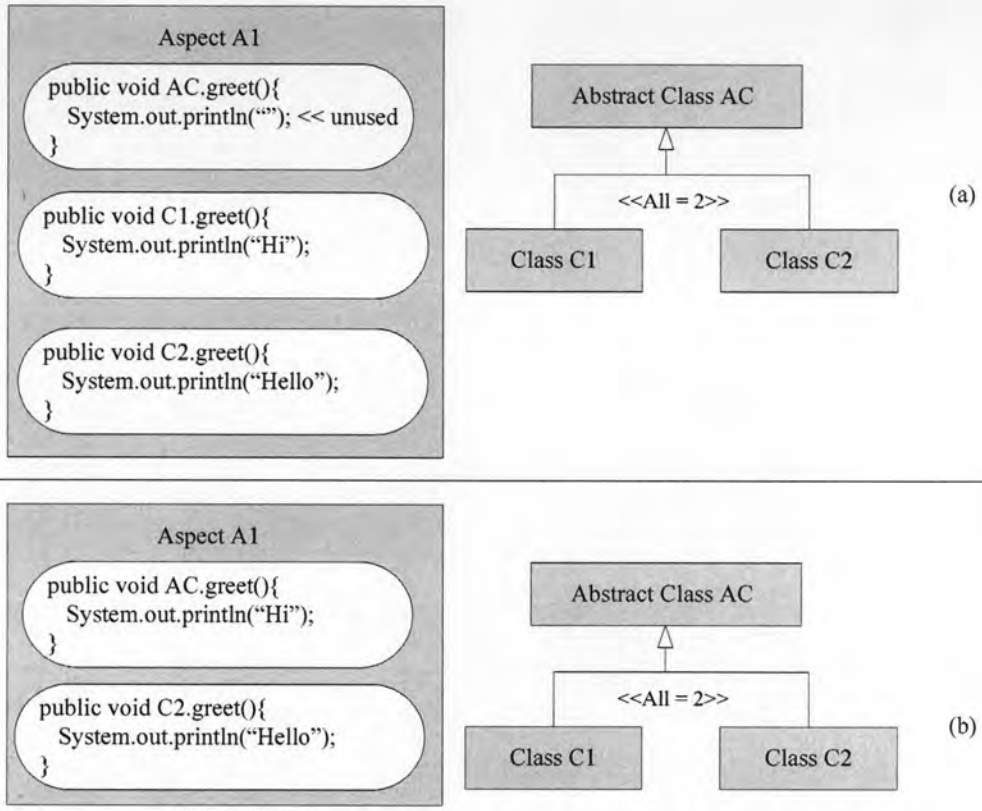

Figure 3.13: (a) A system violating the Guideline 12 and (b) a system following the Guideline 12

Illustrations: Figure 3.13 shows systems consisting of abstract class *AC*, subclass *C1*, subclass *C2*, and aspect *A1*. Figure 3.13(a) shows aspect *A1* which introduces the concrete method *greet()* to abstract class *AC*. Aspect *A1* also introduces the concrete method *greet()* to both subclasses of abstract class *AC* which leads method *greet()* of class *AC* unused. Figure 3.13(b) utilizes the concrete method *greet()* by letting class *C1* to take the greeting behavior from its parent instead.

Guideline 13: Don't define methods of all subaspects to override a concrete method of their abstract aspect.

Description: For the reason conforming to the *Guideline 12*, all subaspects should not define methods to completely override a concrete method of their abstract aspect.

Violation Effects: A violation of the guideline causes a concrete method of the abstract aspect useless. Besides, it points that some part of the system is over-defined.

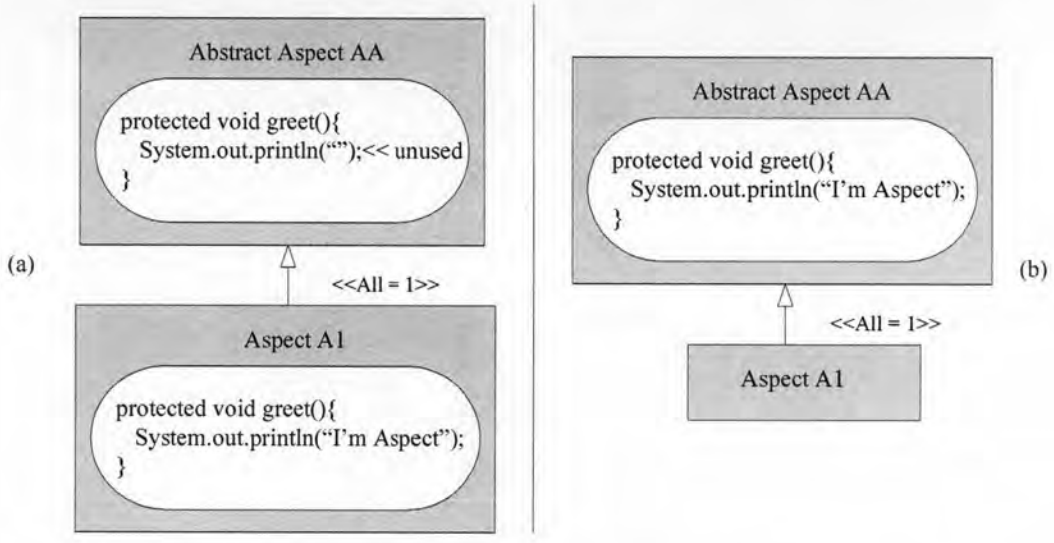

Figure 3.14: (a) A system violating the Guideline 13 and (b) a system following the Guideline 13

Illustrations: Figure 3.14 shows systems consisting of abstract aspect *AA* and subaspect *A1*. Figure 3.14(a) shows aspect *A1*, the only one subaspect of abstract aspect *AA*, which defines the concrete method *greet()* to override the concrete method *greet()* of abstract class *AC*. This definition leads method *greet()* of abstract aspect *AA* useless. Figure 3.14(b) operates the concrete method *greet()* of the abstract aspect.

Guideline 14: Do extract an abstract aspect if the same crosscutting concern has two or more variations.

Description: Generally, an abstract aspect should be promoted for the purpose of reusability [32, 34, 35]. However, modeling an abstract aspect with one subaspect to the concern that will not propagate as the system evolves reveals the over-design system. Thus, an abstract aspect should be extracted when it has two or more variations.

Violation Effects: A violation of the guideline raises the unnecessary coupling between an abstract aspect and its subaspects. Also, it increases the system size by the abstract aspect component.

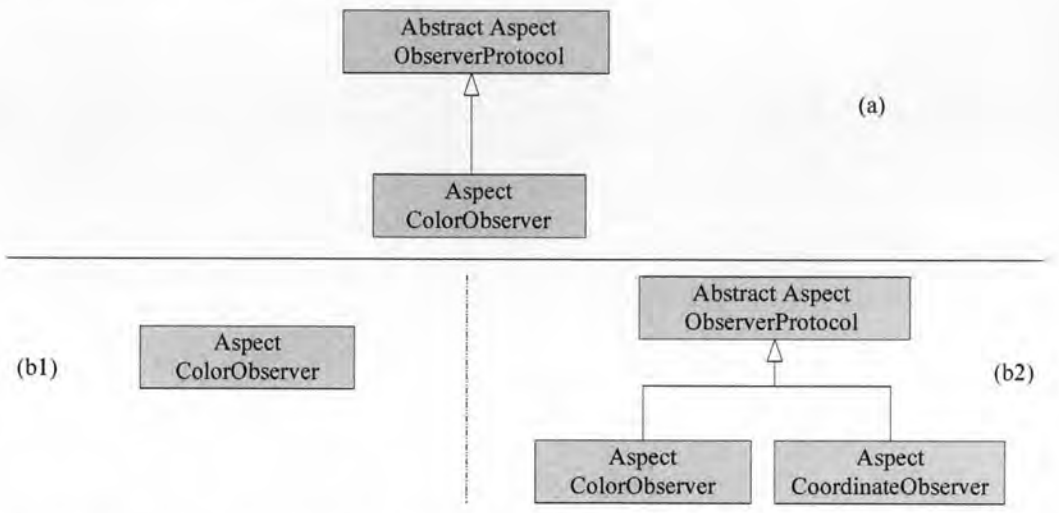

Figure 3.15: (a) A system violating the Guideline 14 and (b) a system following the Guideline 14

Illustrations: Figure 3.15 shows systems consisting of abstract aspect *ObserverProtocol*, subaspect *ColorObserver*, and subaspect *CoordinateObserver*. Figure 3.15(a) shows abstract aspect *ObserverProtocol* with one subaspect only that is aspect *ColorObserver*. So, this abstract aspect should be demoted as shown in Figure 3.15b(b1). Figure 3.15(b2) shows a set of corresponding aspects which abstract aspect *ObserverProtocol* is fully utilized by its subaspects, thus it is suitable for extracting.

Guideline 15: Don't introduce a method which has the same signature as an inherited method to the realized interface.

Description: Aspect-oriented programming allows implicit multiple inheritances. So, introducing a concrete method to an interface should be done carefully, beware of introducing a method which has the same signature as an inherited method to the base class's interface.

Violation Effects: A violation of the guideline confuses the class's users about its real behavior (the inherited behavior from a superclass or the behavior of an inter-type method introduced to an interface). Furthermore, it causes a compilation error in an implementation phase.

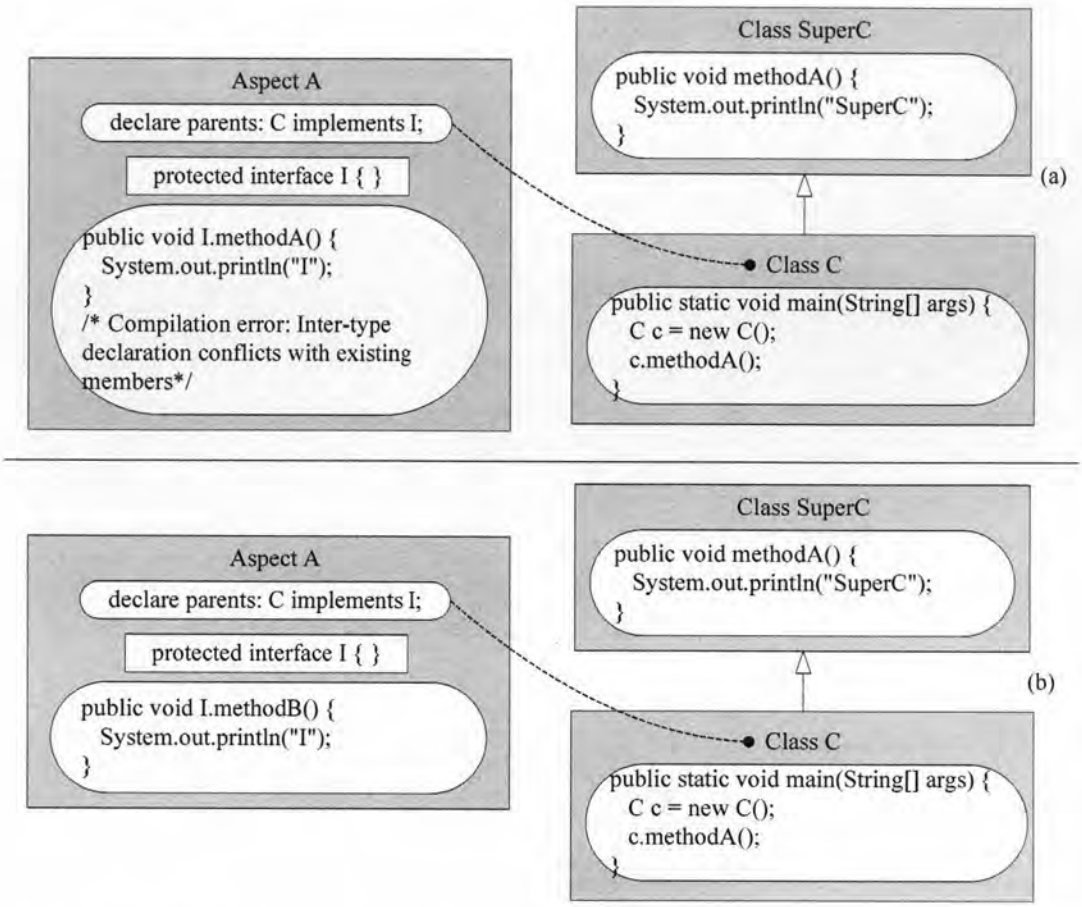

Figure 3.16: (a) A system violating the Guideline 15 and (b) a system following the Guideline 15

Illustrations: Figure 3.16 shows systems consisting of superclass *SuperC*, class *C*, and aspect *A*. Figure 3.16(a) shows aspect *A* which crosscuts the structure of class *C* by declaring class *C* to realize from interface *I*. Accidentally, it also introduces method *methodA()*, which has the same signature as the inherited method *methodA()*, to interface *I*. This introduction produces a compilation error, also confusion of behavior *methodA* behavior. Figure 3.16(b) manages these problems by renaming method *I.methodA()*.

3.3 Design Heuristics and Bad Smells

Because the aspect-oriented system structure contains both the aspect-oriented part and the object-oriented part, design principles are collected from 4 sources to complete the aspect-oriented software measurement: 61 object-oriented design heuristics from [36], 22 bad smells from object-oriented system refactorings [15], 5 bad

smells from aspect-oriented system refactorings [16, 17], and 15 aspect-oriented design guidelines in Section 3.2. All design principles are listed in Appendix A.

Table 3.1: Sorted out design heuristics and bad smells

Sorted out Reasons	Sorted out Design Heuristics and Bad Smells
Principles which overlap others	Heuristic 3.3, 3.4, 4.6, 5.3, Large Class, Divergent Change, Switch Statements, Middle Man, Inappropriate Intimacy, Data Class, Refused Bequest
Principles which are skipped by AspectJ/ Java	Heuristic 5.12, 6.1 – 6.3
Principles which require additional information from humans	Heuristic 2.2, 2.4, 2.8, 2.11, 3.1, 3.5, 3.6, 3.9, 4.1 – 4.4, 4.8 – 4.12, 5.1, 5.4, 5.7, 5.8, 5.11, 5.14, 5.16, 5.18, 5.19, 7.1, 8.1, 9.1, Shotgun Surgery, Data Clumps, Primitive Obsession, Parallel Inheritance Hierarchies, Temporary Field, Alternative Classes with Different Interfaces, Incomplete Library Class, Comments

However, in the process of defining design principle violation check definitions, some design heuristics and bad smells are sorted out for 3 main reasons described in Table 3.1. Firstly, some design principles present the similar goals on the contrary views such as *Heuristic 3.10: Agent classes* of the object-oriented design heuristics and *Middle Man* of bad smells for object-oriented refactorings overlap each other. Secondly, design principles such as design heuristics which relate to explicit extension from two or more classes (*multiple inheritances*) are skipped by AspectJ/ Java specifications. Lastly, concrete violation check definitions for some design principles are not able to specify because they require some semantic judging from measurers. For example, the *Heuristic 4.8: distribute system intelligence vertically down narrow and deep containment hierarchies* are not able to specify its concrete violation check definition, because an evaluation of the system intelligence distribution requires the semantic comprehension of designers. The remaining design heuristics, bad smells, and aspect-oriented design principles are used to define design principle violation check definitions in the next chapter.