Integration testing of aspect-oriented programs: a characterization study to evaluate how to minimize the number of stubs

Reginaldo Ré[1], Paulo Cesar Masiero[2]

[1] Universidade Tecnológica Federal do Paraná – Campus Campo Mourão
Caixa Postal 271 – 87301-005, Campo Mourão, PR, Brazil
reginaldo@utfpr.edu.br

[2]Depto de Ciências de Computação e Estatística – ICMC/USP - São Carlos
Caixa Postal 668 – 13560-970, São Carlos, SP, Brazil
masiero@icmc.usp.br

Resumo. *Um dos problemas encontrados no teste de programas orientados a objetos é a ordem em que classes são integradas e testadas. Esse problema também pode ser observado em programas orientados a aspectos. A estratégia incremental, que sugere que classes sejam testadas primeiramente e, então, integradas aos aspectos, é frequentemente sugerida como a estratégia mais adequada para integrar classes e aspectos. Este trabalho apresenta um estudo sobre ordenação de classes e aspectos em programas orientados a aspectos para miniminar o número de módulos pseudo-controlados implementados durante a integração. Um estudo de caracterização em que um sistema de telecomunicação é integrado utilizando quatro ordens diferentes é apresentado. As estratégias de ordenação analisadas foram a combinada, que é uma extensão da estratégia OO otimizada; a incremental+, que é a ordem incremental melhorada; a reversa, que é a ordem inversa da combinada; e a randômica, que foi definida aleatoriamente.*

Abstract. *A problem related to the integration test of object-oriented programs is the order that classes are integrated and tested. This problem also appears in aspect-oriented programs. The incremental integration strategy, which suggests that classes are tested first and then integrated to the aspects, is often proposed as the more adequate strategy to integrate classes and aspects. This work presents a study about ordering classes and aspects in aspect-oriented programming to minimize the number of stubs in integration testing. A characterization study in which a Telecom system is integrated in four different orders is reported. The ordering strategies analyzed were the combined, which is an extension of an optmized OO strategy; incremental+, which is the incremental order improved; reverse, which is the combined reverse order; and, random, which was aleatorily defined.*

1. Introduction

Aspect-Oriented programming (AOP) aims at facilitating software development through the separation of concerns related to functional and non-functional software characteristics [Kiczales et al. 1997]. Despite its numerous advantages this approach also presents several new challenges for software development [Alexander and Bieman 2004; Krinke and Störzer 2003]. Several proposals have been made to support the development of aspect-oriented software and solve some of these new challenges [Aldawud and Bader 2003; Baniassad and Clarke 2004]. In the context of software integration testing strategies, several authors proposed the strategy of incremental test of aspect-oriented programs [Ceccato et al. 2005; Zhou et al. 2004]. In general terms, the incremental approach suggests that the base code (or component) should be developed, unit-tested, integrated and tested; then the aspects are developed and integrated to the base code. Supporters of the incremental approach justify it by the fact that AOP is an extension of object-oriented programming (OOP) and an aspect usually only adds behavior (or functions) to the base code, which is not conscious of this.

This problem is similar to the integration testing of object-oriented programs: the order that classes are tested and integrated with other classes [Tai and Daniels 1999]. The order in which classes are developed, the number of stubs that need to be implemented, and the order and easiness errors are found are all influenced by the order of integration. When one class requires another to be available before it can be executed there is a *dependency* relationship [Abdurazik and Offutt 2006]. This problem arises when the system under test is composed by classes in a dependency cycle. The effect of breaking a dependency cycle is that a stub must be created for the class that is the sink of the edge removed. When there is no cycle, this problem can be easily solved by a reverse topological ordering of classes based on the dependency relationship. Several proposals have been made to order implementation and test of classes aiming at minimizing the number of stubs and therefore minimize the development effort [Briand et al. 2003; Kung et al. 1995; Le Traon et al. 2000; Tai and Daniels 1999].

This paper presents a characterization study conducted to analyze the results of four integration orders of a extended Telecom system implemented using AOP: combined [Ré et al. 2007; Ré and Masiero 2005], which is an extension of an optimized OO strategy; incremental+, which is the incremental order improved; reverse, which is the combined reverse order; and, random, which was aleatorily defined. The first two use an algorithm proposed by Briand et al. [2003]. In Section 2 we discuss related work and motivation for our work. In Section 3 we briefly review the ordering algorithm proposed by Ré et al. [2007] and present an aspect-oriented program that was used in the study. In Section 4 we describe the study and discuss the results for the four orders considered. Concluding remarks are presented in Section 5.

2. Related Work and Motivation

Following well known strategies from object-oriented software development testing, test of aspect-oriented programs comprises several general steps: first it must consider a class, its methods and crosscutting aspects (intra-class/aspect), then test several classes and aspects combined (inter-class/aspect). Zhao [2003] proposed a structural testing technique to test aspect-oriented software that considers clusters of classes and aspects. However, Zhao´s paper focuses more on the data structure and algorithm to do the test than in an integration strategy.

An incremental testing approach is proposed by Ceccato et al. [2005] in which all base classes are initially tested without considering aspects, then gradually the aspects are added and tested with the test cases designed to test the base classes and finally test cases to evaluate the behavior of base classes in the presence of aspects (integration test) are added. They comment that a problem of this approach is the need of frequently creating stubs to simulate the behavior of aspects from which the base classes are dependent. This happens for example with data persistence. This very same problem has been noted by Soares et al. [2002], who used the incremental approach to develop and test an application in the area of health care and implemented data persistence and distribution using aspects. However, they did not detail this subject in their paper: they did not mention the use of stubs for integration testing neither proposed a specific order for the integration of classes and the integration of aspects.

Also, many researchers have found crosscutting concerns that are dependent of other crosscutting concerns, which leads to implementations with one aspect depending on another aspect. The AspectJ Team [2002] shows an example of a Telecom system to control long distance calls that uses an aspect to measure call durations. This aspect, by its turn, is also crosscut by a logging aspect. This is also an example of aspect that implements a functional requirement. Use of aspects to implement (variable) requirements in software product lines and to implement business rules have also been studied and reported by Loughran and Rashid [2004] and Cibrán et al. [2003].

All these considerations and the works reviewed here motivated us to look into this problem of what is a good strategy to do integration testing of classes and aspects in the context of a software development process. In particular, we are interested in the order of integration of classes and aspects aiming at minimizing the need to create stubs in the line of the works of Briand et al. [2003] and Le Traon et al. [2000]. From the analysis of the literature we found that aspects can form dependency cycles among themselves and in some cases they may form cycles with classes thus leading to the need of creating stubs during integration testing [Kienzle et al. 2003].

We did not find articles in the literature that discuss integration testing strategies to minimize the number of stubs for AO programs as well as studies regarding the influence of the implementation of functional concerns as aspects and of aspects that are partially known to the base code they advise, such as persistence. In this paper we report a characterization study to practically compare several integration strategies that builds on our previous work [Ré et al. 2007; Ré and Masiero 2005] proposing the use of the algorithm of Briand et al. [2003] to calculate the order of integration testing of classes and aspects [Ré et al. 2007; Ré and Masiero 2005].

3. Strategies for Ordering Classes and Aspects

The work of Kung et al. [1995] was the first to propose a solution to the problem of dependency cycles and to apply it to integration testing of OO programs. They proposed a diagram called ORD (Object Relationship Diagram - a digraph) to represent the dependency relationships of inheritance ("I"), aggregation ("Ag"), and association ("As"). The ordering algorithm applied to an ORD is based on two main concepts: *cluster*, which is the set with the maximum number of vertices mutually reachable in the ORD, and *break of cycles*, which is the temporary removing of an edge with the objective of making the digraph acyclic. They proposed that only the edges representing associations must be removed, because inheritance and aggregation present data coupling, control coupling, and code dependency. When an edge is removed, a stub is created to simulate the behavior of a class participating in the relationship that generated the dependency. The dependency of code and coupling of data and control result in more complex stubs. The class ordering for test is given after a reversal topological ordering of the acyclic graph.

The work of Tai and Daniels [1999] improved on the work of Kung et al. [1995]. They calculated level numbers for each class and proposed a criterion for removing edges based on the calculus of weight for a vertex, which is the multiplication of the number of edges reaching the source vertex by the number of edges leaving the sink vertex. Le Traon et al. [2000] proposed another type of graphical representation for class dependency and used recursively the algorithm of Tarjan to identify strongly connected components (SCC)[Tarjan 1972].

Briand et al. [2003] presented a proposal that uses ideas from the algorithms of Le Traon et al. [2000] and of Tai and Daniels [1999]: the algorithm of Tarjan is used recursively to identify SCCs and weights are associated to edges that represent dependency of the type that should be removed to break the dependency cycle. This strategy is deterministic and minimizes the number of specific stubs, differently from Le Traon et al. [2000], and the result is not sub-optimal as the result of Tai and Daniels [1999]. A stub of a class `A` is specific when it is implemented to test the integration with only one other class `B`. It is realistic when it can be used to test not only `B` but any other class that depends on `A`.

Ré et al. [2007] proposed an adaptation to the ORD to represent dependencies created by an aspect-oriented program so that the algorithm of Briand et al. [2003] can be used to compute the integration testing order of classes and aspects and the stubs needed. This algorithm is briefly described later in this section.

Figure 1 shows part of a class and aspect diagram of a simple Telecom application proposed by The AspectJ Team [2002] as an example of AspectJ programming and extended by the authors to include data persistency and changed to make the aspect `Timing` to depend on the aspect `Billing`. It is used here to show how to create the extended AO-adapted ORD and as the basis for the study reported in Section 4. The example implements the concerns of timing, billing and call transfers as aspects. This system helps customers to accept, join, and end local and long distance calls. The base classes simulate the customers, calls and connections. Some classes of the system are:

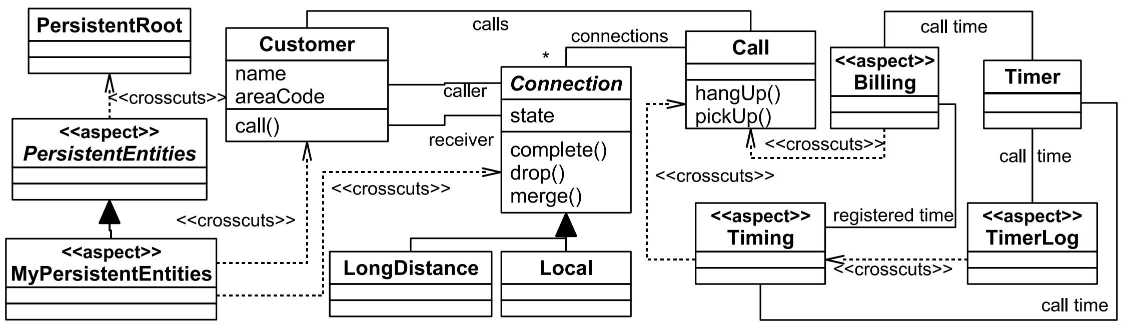

Figura 1. Telecom System.

- `Customer`, which handles the customer´s data such as name, phone number, area code, and password to have access to services offered by the operator on the Web.

- `Connection`, an abstract class extended by the `Local` and `LongDistance` classes, which handle these types of calls.
- `Call` and `Timer`, whose names are self-explanatory.

Some of the system's aspects are:

- `Timing`, which implements the timing concerns and measures the duration of each call connection by initializing and stopping a timer associated to each connection.
- `Billing`, which implements the billing concern establishing a customer to pay each connection and the correct amount, according to the call type: local or long distance.
- `TimerLog`, which implements a log file and displays on the screen the initial and final times of each timer.

A crosscutting persistence framework has been added to this system to persist the data related to each connection [Camargo and Masiero 2005]. Figure 1 shows only part of the framework: the class `PersistentRoot` and the aspects `PersistentEntities` and `MyPersistentEntities`. The abstract aspect `PersistentEntities` contains methods typically used for data persistence, such as store and load data, which are introduced by the class `PersistentRoot`. The aspect `MyPersistentEntities` makes concrete the aspect `PersistentEntities` and is responsible for indicating which classes are persistent and also to make possible to insert new methods related to persistence operations in persistent classes without changing the framework´s structure.

Aspects always depend only on the join points defined by the pointcuts and objects of context used by advices, except the join points with a conditional clause. These, besides the dependency defined by the join point, have also dependency on the instances of the classes that appear in the expression for the conditional clause. Pointcuts can be defined by a combination of several syntactic elements contained in their declaration. Thus, to define the dependency relationship it is necessary to analyze all the elements in the declaration of a pointcut. Depending on how early planning of integration test is done, the pointcuts could not be already coded. Thus, the information contained in the design artifacts, which is sometimes called a crosscutting interface [Krechetov et al. 2006], could be used. There are several proposes in the literature for notation that extend class diagrams to represent aspects. However, in this case some fine grained details of the model may be lost. For example, we can have an aspect representing persistence when in fact an actual implementation of persistence may contain a greater number of aspects.

The regular vertex in the ORD definition is extended to represent both classes and aspects and new relationships among the two types of vertices representing classes and aspects that contain advices and intertype declarations. An aspect can associate one or more advices to one or more pointcuts. The association between a pointcut and a base code, be it a class method or other advice, is a crosscutting association represented by a dependency relationship noted by a "C" (from Crosscutting). This example is depicted in Figure 2: the aspect `Billing` crosscuts the class `Call`. In the same figure, the aspects `Billing` and `Timing` are examples of relationships that occur only between aspects. The type of dependency generated by this relashionship is a use dependency, denoted by a "U". The use dependency can also happen between aspects composed just by pointcuts, when a pointcut is defined using another pointcut. Advices defined in pointcuts sometimes need context information from the pointcuts. These information are captured by the pointcut using the clauses `this`, `target`, and `args`, or by the use of `thisJoinPoint`. In that case, there is an association dependency ("As") with objects involved in pointcut definition. This is shown in Figure 2, by the relationship between `Timing` and `Customer`.

It is possible in AspectJ to use intertype declarations to change the class structure adding methods and attributes and modifying inheritance relationships. Attributes introduced in the class can be used regularly by any other method of the class if it is visible. Methods introduced can also use private attributes of the base class. Therefore, there is a strong relationship between the base classes and the aspects containing intertype relationships. These cannot be tested independently from the base code. This dependency type is labeled with "It". This type of dependency is also characterized when an aspect changes an inheritance hierarchy using a `declare parents` clause. In Figure 2, we can see these types of dependencies in two clusters of classes and aspects: between `Billing` and `Local`; and among `MyPersistentEntities`, `PersistentRoot` and `Connection`. The use of this type of construction has a strong impact on the base classes and on testing since a change on the system structure can be significant. It is also used in regression testing to identify which classes are affected when classes belonging to the system are changed [Kung et al. 1995; Milanova et al. 2002].

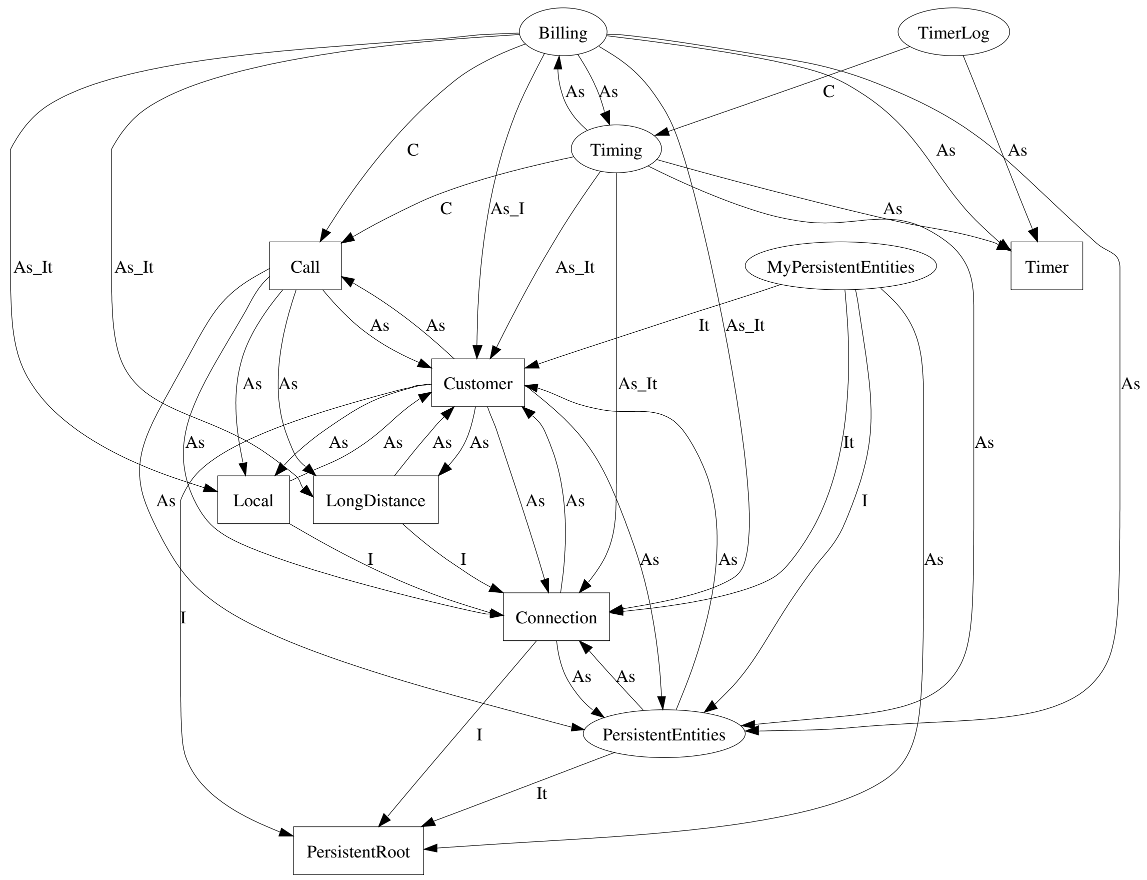

Figura 2. ORD of the Telecom system

Finally, there can be inheritance relationships among aspects and among classes and aspects. Aspects can be specialized if there is a concrete aspect, the last in the inheritance hierarchy, and an aspect can be a specialization of a class. These relationships generate an inheritance dependency in the extended ORD, labeled "I", which is similar to the inheritance among classes already defined in the ORD, as is illustrated by the aspects `PersistentEntities` and `MyPersistentEntities`, in Figure 2. The order proposed by the algorithm to this ORD (and the stubs required) showed in Table 1 is: `PersistentRoot`, `Connection (Customer,PersistentEntities)`, `PersistentEntities(Customer)`, `Customer (Call, Local,LongDistance)`, `LongDistance`, `Local`, `Call`, `MyPersistentEntities`, `Timer`, `Billing (Timing)`, `Timing` and `TimerLog`.

4. Description of the characterization study

4.1. Study definition and planning

The objective of the characterization study was to analyze the effort to create stubs during integration testing of a system developed with AspectJ using different strategies of integration from the point of view of software developers and testers to verify the best strategy to use. The number of stubs, the number of its methods, attributes, pointcut descriptors (PCDs), and intertype declarations are used to calculate the cost. The study was divided into two parts, according with the integration strategy used to test a system. First, the ORD for the Telecom system was built and the order of classes and aspects was computed by the algorithm of Briand et al. [2003] adapted to POA by Ré et al. [2007]. In the second part of the study, classes and aspects were integrated in four different orders: **combined**, **incremental+**, **reverse** and **random**.

The combined order is the one proposed by the algorithm of Briand et al. [2003] applied to the ORD combining aspects and classes as presented in Section 3. The incremental+ approach consists of testing classes first using the order proposed by Briand et al. [2003] algorithm then integrating aspects in the order proposed by the optimization algorithm applied only to the aspects. Reverse is the reverse combined order. The random order was defined by drawing of lots.

We expected that integration in the reverse order and in the random order would have an implementation cost greater than the other two strategies. We nevertheless decided to include these order in the study to analyze the stubs that would be created and to have an idea of what would be the maximum cost of integration, that is, to have them functioning as controls to the result of the other two orders.

The four integration testings were done in twelve steps – one for each class or aspect of the Telecom system. Each step comprised six tasks:

1. classes and aspects were analyzed to determine which of them would be the module under test;
2. classes and aspects implemented and integrated in previous steps were selected;
3. stubs were implemented;
4. the module under test was integrated to the set of already integrated modules (step 2) and tested with the stubs;
5. the results were analyzed;
6. data regarding the stubs were collected;

For each integration testing we collected the actual number of stubs produced and also the number of attributes, methods, pointcut descriptors and intertype declarations. The latter was separated into declarations to introduce elements (attributes and methods) in classes and declarations to change the program structure (change of hierachy and aspect precedence) as they have different costs when created in a stub and to test.

The following modules and artifacts have been used:

- the module under test in each step (a class or an aspect);
- the modules already integrated and tested in previous steps;
- a test suite, represented by a JUnit class [JUnit 2002];
- a set of stubs of classes and aspects.

All four integration testings have been conducted by one of the co-authors of this paper. A test suite was created using the functional testing criteria equivalence partitioning [Myers 1979]. The same testing suite was used in all four integrations with few minor changes in the JUnit code. A change was made in the Telecom system to facilitate the study: the aspect `Billing`, responsible to calculate the price of calls was changed to indicate when a call was completed so that an aspect responsible for timing could correctly register the duration of the calls.

4.2. Combined strategy

The plan for the integration testing using the combined strategy is shown in Table 1. The data collected from the actual integration is shown in Table 2. The final number of stubs created was 7, as planned and according to the algorithm result. However, we noticed that to test abstract aspects they must be concretized before integration, which has a cost. This happened to the `MyPersistentEntities` aspect. Additionally, we noticed that in some cases a stub created in a previous step had to be reused. This is for example the case of class `LongDistance` that reused the stub of `Call` through `Customer`. This reuse has a very low cost: just copying the stub. We decided to register in Table 2 these two costs in the sequence 7/1/7. It is also interesting to notice that for the class `Customer` two specific stubs were created. Thus, they were computed as two different stubs.

4.3. Incremental+ strategy

The plan for the integration testing for the incremental+ strategy is shown in Table 3 without the fourth column shown in Table 1. The data collected from the actual integration is shown in Table 4. Figure 3 shows how we split the ORD in two ORDs and applied the algorithm of Briand et al. [2003] to obtain the order shown in Table 3. Notice that classes `Timer` and `PersistentRoot` were considered together with the aspects because they are only related to aspects. If they were considered with the classes they coud be integrated in any order.

A stub was needed to test `PersistentEntities` to simulate the behavior of a concrete specialization, the aspect `MyPersistentEntities`. This stub has a footnote in Table 4 to indicate that it is a concrete aspect. It is important to notice that creation of the stub `MyPersistentEntities` was not expected initially, but the test can only be executed with a concrete aspect. This increased the actual number of stubs from 5 to 8. This problem is similar to the test of abstract classes. Thus, we considered the number of stubs of concrete aspects in our calculation, but kept it separated.

Tabela 1. Integration testing plan for the combined strategy.

Order	Module under test	Stub	Tested modules needed
1	PersistentRoot	–	
2	Connection	Customer and PersistentEntities	PersistentRoot
3	PersistentEntities	Customer	Connection and PersistentRoot
4	Customer	Call, Local and LongDistance	PersistentEntities, Connection and PersistentRoot
5	LongDistance		Customer and Connection
6	Local		Customer and Connection
7	Call		Customer, LongDistance, Local, Connection and PersistentEntities
8	MyPersistentEntities		Customer, Connection, PersistentEntities and PersistentRoot
9	Timer		
10	Billing	Timing	PersistentEntities, Timer, Connection, Customer, Call, Local and LongDistance
11	Timing		Billing, PersistentEntities, Timer, Connection, Customer and Call
12	TimerLog		Timing and Timer
	5 stubs of classes		
	2 stubs of aspects		

Tabela 2. Cost of stubs for the combined strategy.

Order	Module		Stub		#Attr.	#Met.	#PCDs	#Members Introduced	#Other Declarations
1	PersistentRoot	C	–	–	–	–	–	–	–
2	Connection	C	Customer[a]	C	1	3	–	–	–
			PersistentEntities	A	0	1	0	1	1
3	PersistentEntities	A	Customer[a]	C	3	8	–	–	–
			MyPersistentEntities[b]	A	0	0	0	0	3
4	Customer	C	Call	C	2	7	–	–	–
			Local	C	0	0	–	–	–
			LongDistance	C	0	0	–	–	–
			MyPersistentEntities[bc]	A	0	0	0	0	3
5	LongDistance	C	Call[c]	C	2	7	–	–	–
			Local[c]	C	0	0	–	–	–
			MyPersistentEntities[bc]	A	0	0	0	0	3
6	Local	C	Call[c]	C	2	7	–	–	–
			MyPersistentEntities[bc]	A	0	0	0	0	3
7	Call	C	MyPersistentEntities[bc]	A	0	0	0	0	3
8	MyPersistentEntities	A	–	–	–	–	–	–	–
9	Timer	C	–	–	–	–	–	–	–
10	Billing	A	Timing	A	0	1	0	3	0
11	Timing	A	–	–	–	–	–	–	–
12	TimerLog	–	–	–	–	–	–	–	–
Total stubs/concretization/reused			7/1/7		6	20	0	4	4/1

[a]Specific stub.
[b]Concrete aspect.
[c]Indirectly reused.

To test Connection, which is an abstract class, we made an initial change making it concrete. This could be done because, differently from the abstract aspects , it did not

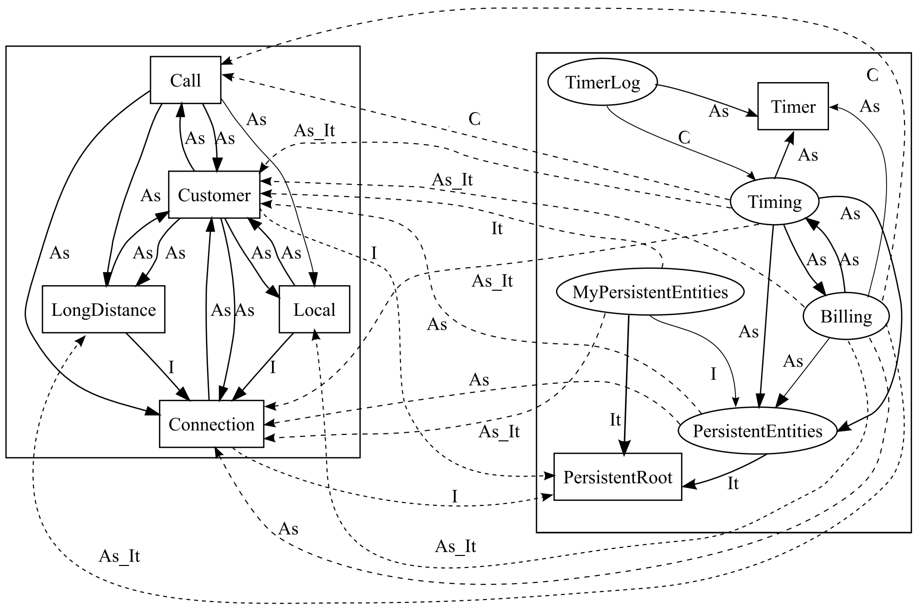

Figura 3. ORD Reorganization to show classes and aspects separated

Tabela 3. Integration testing plan for the incremental+ strategy

Order	Module under test	Stub
1	`Connection`	`Customer`
2	`Customer`	`Call`, `Local` and `LongDistance`
3	`LongDistance`	
4	`Local`	
5	`Call`	
6	`Timer`	
7	`PersistentRoot`	
8	`PersistentEntities`	
9	`MyPersistentEntities`	
10	`Billing`	`Timing`
11	`Timing`	
12	`TimerLog`	
	4 stubs of classes	
	1 stub of aspect	

contain abstract methods and therefore can be instantiated. Therefore, it was not necessary to create concrete specializations of classes to be instantiated.

4.4. Reverse strategy

The actual result of this integration testing is presented in Table 6. It can be noticed confronting Table 5 and Table 6 that the actual number of stubs used was smaller than the expected number. We discovered in this study that the join points selected by pointcuts of type `call` are not used if context information is used, as for example the clauses `this` and `thisJoinPoint`. This was the cause in this study for the actual number of stubs be smaller than the expected number. An example is the aspect `TimerLog` that needs the class `Timer` as well as the aspect `Timing`. Notice also that the dependency of type “C” between `TimerLog` and `Timing` was generated by a join point of type `call`. During the testings this dependency became between `TimerLog` and the class that represents the test cases and not anymore between `TimerLog` and `Timing`.

Tabela 4. Cost of stubs for the incremental+ strategy

Order	Module		Stub		#Attr.	#Met.	#PCDs	#Members Introduced	#Other Declarations
1	`Connection`	C	`Customer`	C	1	4	–	–	–
			`PersistentEntities`[a]	A	0	0	0	2	0
2	`Customer`	C	`Local`	C	0	1	–	–	–
			`LongDistance`	C	0	1	–	–	–
			`Call`	C	2	7	–	–	–
			`PersistentEntities`[a]	A	0	0	0	3	2
3	`LongDistance`	C	`Call`[c]	C	2	7	–	–	–
			`Local`[c]	C	0	1	–	–	–
			`PersistentEntities`[c]	A	0	0	0	3	2
4	`Local`	C	`Call`[c]	C	2	7	–	–	–
			`PersistentEntities`[c]	A	0	0	0	3	2
5	`Call`	C	`PersistentEntities`[a]	A	0	0	0	4	2
6	`Timer`	C	–	–	–	–	–	–	–
7	`PersistentRoot`	C	–	–	–	–	–	–	–
8	`PersistentEntities`	A	`MyPersistentEntities`[b]	A	0	0	0	0	3
9	`MyPersistentEntities`	A	–	–	–	–	–	–	–
10	`Billing`	A	`Timing`	A	0	1	0	3	0
11	`Timing`	A	–	–	–	–	–	–	–
12	`TimerLog`	A	–	–	–	–	–	–	–
Total of stub/concretization/reused			8/1/5		3	14	0	12	7

[a]Specific stub.
[b]Concrete aspect.
[c]Indirectly reused.

Tabela 5. Integration testing plan for the reverse strategy

Order	Module under test	Stub
1	`TimerLog`	`Timer` and `Timing`
2	`Timing`	`Connection`, `Customer`, `Timer`, `Billing` and `PersistentEntities`
3	`Billing`	`Connection`, `Customer`, `Call`, `Local`, `LongDistance`, `Timer` and `PersistentEntities`
4	`Timer`	
5	`MyPersistentEntities`	`Connection`, `Customer`, `PersistentRoot` and `PersistentEntities`
6	`Call`	`Connection`, `Customer`, `Local`, `LongDistance` and `PersistentEntities`
7	`Local`	`Connection` and `Customer`
8	`LongDistance`	`Connection` and `Customer`
9	`Customer`	`PersistentRoot`, `Connection` and `PersistentEntities`
10	`PersistentEntities`	`Connection` and `PersistentRoot`
11	`Connection`	`PersistentRoot`
12	`PersistentRoot`	
	26 stubs of classes	
	7 stubs of aspects	

4.5. Random strategy

Table 7 presents the planning for the integration testing in random order. The result of the integration testing is shown in Table 8. It can be seen that the actual number of stubs needed was smaller that the expected number. There are two reasons for this: dependencies of type `call` also occurred, as in the reverse strategy, and intertype declarations that change the specialization hierarchy may not need stubs.

The second case is illustrated by the aspect `MyPersistentEntities` which changes the inheritance hierarchy of `Customer` and turns it in a specialization of `PersistentEntities`. Therefore, all methods and attributes are inherited by

Tabela 6. Cost of stub for the reverse combined strategy.

Order	Module		Stub		#Attr.	#Met.	#PCDs	#Members Introduced	#Other Declarations
1	`TimerLog`	A	`Timer`	C	2	3	–	–	–
2	`Timing`	A	`Connection`	C	2	4	–	–	–
			`Customer`	C	0	1	–	–	–
			`Timer`	C	2	3	–	–	–
			`Billing`	A	0	0	2	0	1
3	`Billing`	A	`Connection`	C	2	6	–	–	–
			`Customer`	C	2	6	–	–	–
			`Local`	C	0	0	–	–	–
			`LongDistance`	C	0	0	–	–	–
			`Timer`	C	2	3	–	–	–
4	`Timer`	C	–	–	–	–	–	–	–
5	`MyPersistentEntities`	A	`Connection`	C	0	0	–	–	–
			`Customer`	C	0	0	–	–	–
			`PersistentRoot`	C	1	2	–	–	–
			`PersistentEntities`	A	0	0	0	0	0
6	`Call`	C	`Connection`	C	3	6	–	–	–
			`Customer`	C	2	4	–	–	–
			`Local`	C	0	3	–	–	–
			`LongDistance`	C	0	3	–	–	–
7	`Local`	C	`Connection`	C	2	4	–	–	–
			`Customer`	C	0	0	–	–	–
8	`LongDistance`	C	`Connection`	C	2	4	–	–	–
			`Customer`	C	-	-	–	–	–
9	`Customer`	C	`PersistentRoot`	C	0	0	–	–	–
			`Connection`	C	3	8	–	–	–
			`PersistentEntities`	A	0	0	0	0	0
10	`PersistentEntities`	A	`Connection`	C	4	7	–	–	–
			`PersistentRoot`	C	0	0	–	–	–
11	`Connection`	C	`PersistentRoot`	C	0	0	–	–	–
12	`PersistentRoot`	C	–	–	–	–	–	–	–
Total of stub/concretization/reused			28/0/0		29	60	2	0	1

Except for the single stub `Billing` in step 2, all other stubs are specific.

`Customer` and during integration of class `Connection` it is not necessary to implement a module to simulate neither `PersistentRoot` nor `MyPersistentEntities`, because their methods and attributes were simulated directly by the stub of `PersistentEntities`. Although the number of modules became smaller, the number of methods and attributes is not changed. From this we can infer that there is a small reduction of effort because it is not necessary to implement the code of the stub's body.

This can be generalized as follows. Let M be the set of classes and aspects specified by an ORD and let $m_1, m_2, m_3 \in M$ be tested in this order. Let m_1 be a module depending on m_2 and m_3. Let m_2 contain a set of attributes Att and/or a set of methods Met that are either introduced by m_3 or a change of the inheritance hierarchy introduced by any aspect that makes m_3 a generalization of m_2. Since s_2 e s_3 are stubs that simulate m_2 e m_3, respectively, then s_2 can simulate directly the attributes and methods belonging to Att and Met, therefore making unnecessary to implement s_3.

4.6. Analysis of the results obtained

A summary of the results of this study is shown in Table 9. Notice that we considered the sum of specific and concrete aspects as the total of stubs. Considering the numbers of

Tabela 7. Integration testing plan for the random strategy

Order	Module	Stub
1	Timer	
2	Connection	Customer, PersistentRoot and PersistentEntities
3	Local	Customer
4	Billing	LongDistance, Call, Customer, Timing and PersistentEntities
5	Customer	Call, PersistentRoot, LongDistance, PersistentEntities
6	TimerLog	Timing
7	PersistentEntities	PersistentRoot
8	Timing	Call
9	MyPersistentEntities	PersistentRoot
10	PersistentRoot	
11	Call	LongDistance
12	LongDistance	
	13 stubs of classes	
	5 stubs of aspects	

Tabela 8. Cost of stubs for the random strategy

Order	Module under test		Stub		#Attr.	#Met.	#PCDs	#Members Introduced	#Other Declarations
1	Timer	C	–	–	–	–	–	–	–
2	Connection	C	Customer[a]	C	1	2	–	–	–
			PersistentEntities[a]	A	0	0	0	2	0
3	Local	C	Customer[a]	C	1	2	–	–	–
			PersistentEntities[b]	A	0	0	0	2	0
4	Billing	A	Customer[a]	C	1	2	–	–	–
			LongDistance[a]	C	0	0	–	–	–
			Timing	A	0	1	0	1	0
			PersistentEntities[a]	A	0	0	0	3	0
5	Customer	C	Call	C	2	7	–	–	–
			LongDistance[a]	C	0	1	–	–	–
			PersistentEntities[a]	A	0	0	0	4	0
6	TimerLog	C	–	–	–	–	–	–	–
7	PersistentEntities	A	MyPersistentEntities[c]	A	0	0	0	0	3
			PersistentRoot	C	0	0	–	–	–
			Call[b]	C	2	7	–	–	–
8	Timing	A	MyPersistentEntities[bc]	A	0	0	0	0	3
			Call[b]	C	2	7	–	–	–
			PersistentRoot[b]	C	0	0	–	–	–
			LongDistance[b]	C	0	1	–	–	–
9	MyPersistentEntities	A	Call[b]	C	2	7	–	–	–
			LongDistance[b]	C	0	1	–	–	–
			PersistentRoot[b]	C	0	0	–	–	–
10	PersistentRoot	C	–	–	–	–	–	–	–
11	Call	C	LongDistance[b]	C	1	0	–	–	–
12	LongDistance	C	–	–	–	–	–	–	–
Total of stub/concretization/reused			11/1/10		5	15	0	10	3

[a]Specific stub.
[b]Concrete aspect.
[c]Indirectly reused.

Table 9, it can be observed that in the combined strategy there is a tendency of producing a greater number of stubs of classes than stubs of aspects thus leading to a greater number of methods and attributes simulated. On the other hand, there is a tendency of producing a greater number of stubs of aspects in the incremental+ strategy than stubs of classes thus leading to a tendency of a greater number of both types of intertype declarations.

In the reverse order we notice that the number of stubs was much greater than the number of aspects, specially the number of stubs of classes. This is due to the early integration of aspects and to the greater number of associations among aspects and other modules. This shows that it is not a good idea starting integration from the aspects, confiming intuition. The random strategy had a worse cost than the incremental+ and combined strategies and better than the reverse strategy considering only the number of stubs.

Tabela 9. Effort to create stubs in absolute numbers

Strategy	#Attr.	#Met.	#PCDs	#Intr.	#Decl.	**Cost of members**	#Stub of class	#Stub of aspect	**#Total of stubs**
Combined	6	20	0	4	4	**34**	5	3	**8**
Incremental+	3	14	0	12	7	**36**	4	5	**9**
Reverse	29	60	2	0	1	**92**	25	3	**28**
Random	5	15	0	10	3	**33**	7	4	**12**

We also calculated the cost that each member adds to the stub. In Table 10 we show the average number of stub's members considering that all have the same weight. This was done simply diving the total number of stubs by the total number of each member. Using this metric, the random order improves its performance as it produced a number of stubs slightly better than the combined order. This topic is further discussed in the conclusions, but we can see that if we consider the number and cost of members, other sequences than the one analyzed may be better.

Tabela 10. Average cost of each stub

Strategy	Attr. per stub	Met. per stub	PCDs per stub	Intr. per stub	Decl. per stub	Avg. cost of each stub
Combined	0,75	2,50	0,00	0,50	0,50	**4,25**
Incremental+	0,33	1,56	0,00	1,33	0,78	**4,00**
Reverse	1,04	2,14	0,07	0,00	0,04	**3,29**
Random	0,42	1,25	0,00	0,83	0,25	**2,75**

Concluding, the combined strategy performed slightly better in this study than the incremental+ strategy considering only stubs and sligthly worse considering also the members created in the stubs.

5. Concluding Remarks

The first conclusion of our study is that we can adapt the rules to construct the ORD for AOP programs to have a result closer to the actual number of stubs found in the experiments. This could be done leting out of the ORD the dependencies generated by pointcuts of type `call` without context information. This is possible to do but depends on the artifacts avaliable to create the ORD: design documentation only may not contain the information about pointcuts. The same can be done with intertype declarations that change the hierarchy of specializations, as occured with `MyPersistentEntinties`.

The second conclusion is that we need to investigate further the role of members in a stub. The work of Milanova et al. [2002] and Labiche et al. [2000] address this problem and tries to minimize it for object oriented programs. Milanova et al. [2002] present a proposal based on Java bytecode, that is different from our approach of deriving the ORD from design documentations. The proposal of Labiche et al. [2000] consists in: postpone the test of abstract classes up to the test of one of its concrete specializations; change the dependencies from an abstract class to one of its concrete specializations. Although the solution of Labiche et al. [2000] do not use ORDs, it seems to be easy to adapt it and use it in the construction of ORDs. Generaly, let M be the set of classes and aspects present in an ORD and m_1, $m_2 \in M$, such that m_1 depends on m_2. Let MHC be the set of all concrete specializations of m_2. The dependency between m_1 and m_2 disappears and dependency among m_1 and all members of MHC are created.

We also noticed that many stubs of aspects are very close to the actual implementation. Thus, the effort to implement them later in the integration phase is small. Also, stubs of aspects that implement concerns that cannot be completely uncounsious from the base code, as persistence, can be reused in other applications, paying back the effort to develop them.

Referências

Abdurazik, A. and Offutt, J. (2006). Coupling-based class integration and test order. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 50–56, New York, NY, USA. ACM Press.

Aldawud, O; Elrad, T. and Bader, A. (2003). A UML profile for aspect-oriented software development. In *3rd Int. Workshop on Aspect-Oriented Modeling*. Aldawud, O.; Kandé, M.; Booch, G.; Harrison, B. and Stein, D.

Alexander, R. T. and Bieman, J. M. (2004). Towards the systematic testing of aspect-oriented programs. Technical Report CS-4-105, Colorado State University, Fort Collins, Colorado.

Baniassad, E. L. A. and Clarke, S. (2004). Theme: An approach for aspect-oriented analysis and design. In *26th International Conference on Software Engineering (ICSE 2004), 23–28 May 2004, Edinburgh, United Kingdom*, pages 158–167. IEEE Computer Society.

Briand, L. C., Labiche, Y., and Wang, Y. (2003). An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607.

Camargo, V. V. and Masiero, P. C. (2005). Object-oriented frameworks. In *XIX SBES - Brazilian Symposium on Software Engineering*, pages 200–216, Uberlândia – Brazil. in Portuguese.

Ceccato, M., Tonella, P., and Ricca, F. (2005). Is AOP code easier or harder to test than OOP code? In *First Workshop on Testing Aspect-Oriented Programs (WTAOP 2005)*, Chicago, Illinois – USA.

Cibrán, M., D'Hondt, M., and Jonckers, V. (2003). Aspect-oriented programming for connecting business rules. In *6th International Conference on Business Information Systems*.

JUnit (2002). Junit. JUnit.org.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Aksit, M. and Matsuoka, S., editors, *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer.

Kienzle, J., Yu, Y., and Xiong, J. (2003). On composition and reuse of aspects. In Leavens, G. T. and Clifton, C., editors, *FOAL: Foundations of Aspect-Oriented Languages at the AOSD 2003*.

Krechetov, I., Tekinerdogan, B., Garcia, A., Chavez, C., and Kulesza, U. (2006). Towards an integrated aspect-oriented modeling approach for software architecture design. In *8th Workshop on Aspect-oriented Modeling (AOM'06) at the Fifth International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany.

Krinke, J. and Störzer, M. (2003). Interference analysis for AspectJ. In *Workshop on Foundations of Aspect-Oriented Languages at the Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD '2003)*, Boston – USA.

Kung, D. C., Gao, J., Hsia, P., Lin, J., and Toyoshima, Y. (1995). Class firewal, test order and regression testing of object-oriented programs. *Journal of Object-Oriented Program*, 8(2):51–65.

Labiche, Y., Thévenod-Fosse, P., Waeselynck, H., and Durand, M.-H. (2000). Testing levels for object-oriented software. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 136–145, New York, NY, USA. ACM Press.

Le Traon, Y., Jéron, T., Jézéquel, J., and Morel, P. (2000). Efficient OO integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12–25.

Loughran, N. and Rashid, A. (2004). Framed aspects: Supporting variability and configurability for aop. In *Software Reuse: Methods, Techniques and Tools: 8th International Conference on Software Reuse, ICSR 2004, Madrid, Spain, July 5-9.*, volume 3107 of *Lecture Notes in Computer Science*, pages 127–140. Springer.

Milanova, A., Rountev, A., and Ryder, B. (2002). Constructing precise object relation diagrams. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 586, Washington, DC, USA. IEEE Computer Society.

Myers, G. J. (1979). *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.

Ré, R., Lemos, O. A. L., and Masiero, P. C. (2007). Minimizing stub creation during integration test of aspect-oriented programs. In *WTAOP '07: Proceedings of the 3rd workshop on Testing aspect-oriented programs*, pages 1–6, New York, NY, USA. ACM Press.

Ré, R. and Masiero, P. C. (2005). Avaliação da abordagem incremental no teste de integração de programas orientados aspectos. In *XIX SBES - Brazilian Symposium on Software Engineering*, volume 19, pages 168–183, Uberlândia – Brazil. in Portuguese.

Soares, S., Laureano, E., and Borba, P. (2002). Implementing distribution and persistence aspects with AspectJ. In *Proc. of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 174–90. ACM Press.

Tai, K.-C. and Daniels, F. J. (1999). Interclass test order for object-oriented software. *Journal of Object-Oriented Programming*, 12(4):18–25,35.

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160.

The AspectJ Team (2002). The AspectJ programming guide. Xerox Croporation.

Zhao, J. (2003). Data-flow-based unit testing of aspect-oriented programs. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, page 188, Washington, DC, USA. IEEE Computer Society.

Zhou, Y., Richardson, D., and Ziv, H. (2004). Towards a practical approach to test aspect-oriented software. In Beydeda, S., Gruhn, V., Mayer, J., Reussner, R., and Schweiggert, F., editors, *Testing of Component-Based Systems and Software Quality, TECOS 2004 (Workshop Testing Component-Based Systems)*, pages 1–16, Erfurt – Germany.