JPI UML Software Modeling

Aspect-Oriented Modeling for Modular Software

Cristian Vidal Silva
Escuela de Ingeniería Informática, Facultad de Ingeniería y Administración, Universidad Bernardo O'Higgins
Santiago, Chile

Rodolfo Villarroel
Escuela de Ingeniería Informática, Facultad de Ingeniería
Pontifica Universidad Católica de Valparaíso
Valparaíso, Chile

Leopoldo López
Instituto de Investigación y Desarrollo Educacional, IIDE
Universidad de Talca
Talca, Chile

Miguel Bustamante
Escuela de Ingeniería Comercial
Facultad de Economía y Negocios, Universidad de Talca,
Talca, Chile

Rodolfo Schmal
Escuela de Ingeniería Informática Empresarial
Facultad de Economía y Negocias
Universidad de Talca
Talca, Chile

Víctor Rea Sanchez
Facultad de Ciencias de la Ingeniería
Universidad Estatal de Milagro
Milagro, Ecuador

***Abstract*—Aspect-Oriented Programming AOP extends object-oriented programming OOP with aspects to modularize crosscutting behavior on classes by means of aspects to advise base code in the occurrence of join points according to pointcut rules definition. However, join points introduce dependencies between aspects and base code, a great issue to achieve an effective independent development of software modules. Join Point Interfaces JPI represent join points using interfaces between classes and aspect, thus these modules do not depend of each other. Nevertheless, since like AOP, JPI is a programming methodology; thus, for a complete aspect-oriented software development process, it is necessary to define JPI requirements and JPI modeling phases.**

Towards previous goal, this article proposes JPI UML class and sequence diagrams for modeling JPI software solutions. A purpose of these diagrams is to facilitate understanding the structure and behavior of JPI programs. As an application example, this article applies the JPI UML diagrams proposal on a case study and analyzes the associated JPI code to prove their hegemony.

Keywords—JPI; UML; AOP; JPI UML Class Diagram; JPI UML Sequence Diagram

I. INTRODUCTION

Aspect-Oriented Programming, AOP [4] [5] [6] [8] is an extension of Object-Oriented Programming OOP that introduces aspects, i.e., modules that advise classes' behavior or add structural members to base classes. Aspects are intended to isolate and modularize crosscutting concerns in classes and methods of software components.

Even though AOP isolates crosscutting concerns, it also introduces implicit dependencies between advised classes and aspects. First, aspects define pointcut PC rules, which alter advised classes' behavior; base classes are completely oblivious about changes to their behavior and structure during program execution. Second, changes in the signature of advised methods of target classes can produce ineffective or spurious aspects, i.e., occurrences of *the fragile pointcut problem* [1] [3]. Furthermore, [2] [3] [9] observe that dependencies between classes and aspects compromise independent development of base modules and aspects code. In classic AOP, developers of both, base code and aspects, need some knowledge about of all software modules, i.e., base classes and aspects that might advise them, rules and associated advice code.

For isolating crosscutting concerns and getting modular AOP programs without the mentioned implicit dependencies, [1] proposed the concept of Join Point Interface JPI as new AOP programming methodology. Like classic AOP [4] [5] [6], aspects in JPI isolate crosscutting functionalities; but, unlike classic AOP, JPI aspects do not provide PC rules. Instead, aspects in JPI implement defined join point interfaces. In addition, in JPI, advised classes define like PC rules for the join point interfaces exhibition.

Looking for a complete JPI software development process, this article proposes deploying two types of UML diagrams: class diagrams and sequence diagrams to model JPI programs, and presents a running example of a JPI program. Thus, the main goal of this article is to present diagrams to understand the structure and behavior of JPI programs and apply them to a case study to analyze their hegemony with the associated JPI code for a complete JPI software development process. Clearly, this is basic for the goal of reaching a model-driven JPI development methodology in the future.

This paper is organized as follows: Section II describes traditional UML class diagrams along with proposed extension to support JPI, JPI UML class diagrams. Section II also describes the running JPI program example, and applies JPI UML class diagrams on it; Section III presents traditional UML sequence diagrams and their extension JPI UML sequence diagrams. Like Section II, Section III applies JPI sequence diagrams on the running example; Section IV presents, for the running example, a consistency analysis of JPI code and JPI UML diagrams; Section V describes related work; and, Section VI presents the conclusions and future work.

II. UML CLASS DIAGRAMS

A. Classic UML Class Diaggrams

For object-oriented software modeling, UML class diagrams model the resources used to build and operate the system. Class diagrams model each resource in terms of its structure and relationships to other resources [7].

As an example, taking in account a *Shopping Session System SSS* that preserves a record of costumers, items in the stock, and transactions. The *SSS* also maintains information about each shopping session that a costumer initiates; a shopping session may include any number of transactions. Figure 1 shows an UML class diagram for the described structure of *SSS* in which classes include described attributes and methods.

In general, new requirements for *SSS* will demand changes in the entire system. For example, let us consider the following new system requirements:

1) *Frequent customer should receive a discount,*
2) *To log all transactions.*

For these requirements, a classic solution consists of adding new attributes and methods to either *ShoppingSession* or *Transaction* class. Hence, either the *buying*(..) method of class *ShoppingSession* or the constructor method of class *Transaction* would invoke new required methods; mentioned methods would include non-natural attributes and behavior no needed for their core purpose. Thus, these extensions represent clear examples of crosscutting concerns.

B. JPI UML Class Diagrams

This article follows ideas of [12] to propose and apply on the *ShoppingSession* system JPI class-based diagram to model *JPI* systems. The stereotype <<jpi>> labels join point interfaces which may not contain attributes or methods. In addition, a class linked to a JPI *exhibits* that join point interface and possibly defines a *pointcut* PC rule for that exhibition, i.e., a rule that defines a design policy through aspects and thus precludes any design violation at the join point events. In our proposal, aspects are represented as normal classes that define attributes and methods, and stereotyped by <<aspect>>. Since aspects implement join point interfaces, they directly link to a join point interface class and define a kind of join point (before, around, and after) for the join point interface implementation.

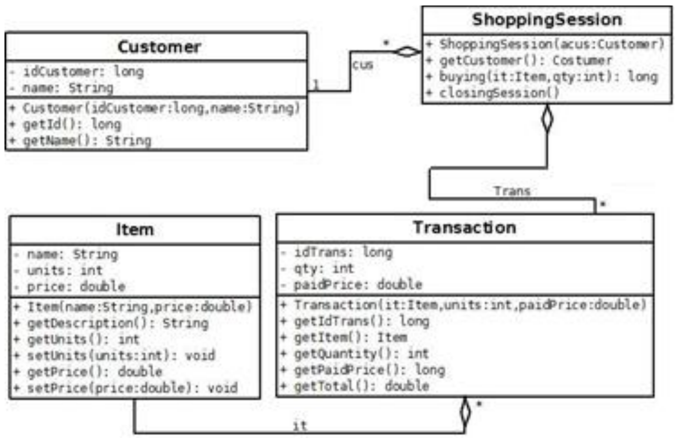
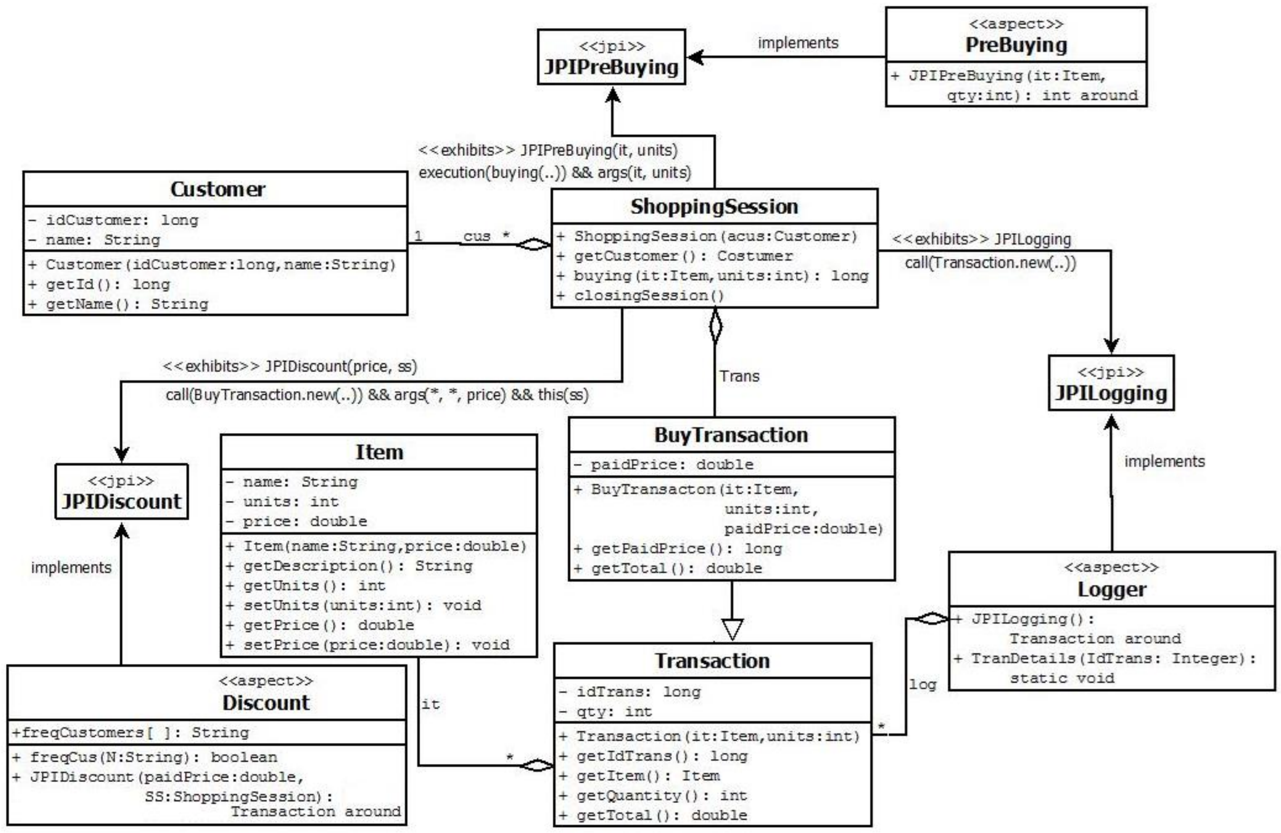


Fig. 1. UML class diagram of the system Shopping Session

Figure 2 shows a JPI UML class diagram for the *JPI SSS* version. Note that there is a join point interface *JPIPreBuying* to link the *ShoppingSession* class and *PreBuying* aspect. In these associations, *JPIPreBuying* defines a method exhibited by class *ShoppingSession* and implemented by aspect *PreBuying*; class *ShoppingSession* defines a PC rule for the *buying(..)* method execution. Furthermore, Figure 2 presents a join point interface *JPIDiscount* to link the *ShoppingSessio*n class and *Discount* aspect, as well as, a join point interface *JPILogging* to link the *ShoppingSession* class and *Logger* aspect. For the first mentioned association, *ShoppingSession* exhibits the method *JPIDiscount(price, ss)*, a method defined by *JPIDiscount* and implemented by the *Discount* aspect, and defines a PC rule for the *BuyTransaction* class invocation. In this case, *price* is an argument of the constructor whereas *ss* corresponds to the *ShoppingSession* instance that invokes for the execution of *BuyTransaction* class constructor. It is necessary to remark, each link from a class to a join point interface is stereotyped by the name <<exhibits>> to indicate the associated join point interface method and its arguments along with a PC rule to define the join points occurrence. Similarly, the *implements* signature labels links from aspects to join point interfaces. Thus, since JPI UML class diagrams only applies stereotypes for associations and JPI elements; therefore, usual UML tools seems able for JPI UML class modeling.

Next section presents details about a proposal for the behavior modeling of a JPI system by JPI sequence diagrams, and presents example models for scenarios of the *ShoppingSession* system, as well.



<<jpi>> JPIPreBuying
implements
<<aspect>> PreBuying
+ JPIPreBuying(it:Item, qty:int): int around
<<exhibits>> JPIPreBuying(it, units)
execution(buying(..)) && args(it, units)
Customer
- idCustomer: long
- name: String
+ Customer(idCustomer:long,name:String)
+ getId(): long
+ getName(): String
1
cus *
ShoppingSession
+ ShoppingSession(acus:Customer)
+ getCustomer(): Costumer
+ buying(it:Item,units:int): long
+ closingSession()
<<exhibits>> JPILogging
call(Transaction.new(..))
<<jpi>> JPILogging
<<exhibits>> JPIDiscount(price, ss)
call(BuyTransaction.new(..)) && args(*, *, price) && this(ss)
Trans
<<jpi>> JPIDiscount
Item
- name: String
- units: int
- price: double
+ Item(name:String,price:double)
+ getDescription(): String
+ getUnits(): int
+ setUnits(units:int): void
+ getPrice(): double
+ setPrice(price:double): void
BuyTransaction
- paidPrice: double
+ BuyTransacton(it:Item, units:int, paidPrice:double)
+ getPaidPrice(): long
+ getTotal(): double
implements
implements
<<aspect>> Logger
+ JPILogging(): Transaction around
+ TranDetails(IdTrans: Integer): static void
log
<<aspect>> Discount
+freqCustomers[ ]: String
+ freqCus(N:String): boolean
+ JPIDiscount(paidPrice:double, SS:ShoppingSession): Transaction around
it
*
Transaction
- idTrans: long
- qty: int
+ Transaction(it:Item,units:int)
+ getIdTrans(): long
+ getItem(): Item
+ getQuantity(): int
+ getTotal(): double
*

Fig. 2. JPI UML class diagram of updated version of system ShoppingSession

III. JPI UML SEQUENCE DIAGRAMS

UML sequence diagrams model execution scenarios of object-oriented programs [7]. Hence, to model the interaction between participants in the execution of *JPI* systems, this article proposes the so-called JPI sequence diagram which considers aspects as a sort of participants stereotyped by <<aspect>> in diagrams, and interactions between participant objects and aspects at join points are denoted by *advices*. Our modeling hypothesis is that by means of JPI sequence diagrams, the associated behavior of JPI programs for model-scenarios is deductible.

According to JPI notation, *join point interfaces* act as a bridge to let in the UML class instance catches up the result of the defined aspect's method [1] [2] [3]. Communication between aspects and classes instances is synchronous. Thus, when an instance of an aspect advises an object, i.e., it implements a *join point interface* for that class instance, the advised object, in order to continue its actions, waits for a message from the aspect to proceed.

AOP languages like AspectJ as well as JPI only permit *around advices* to explicitly proceed. Therefore, in AspectJ and JPI, *before* and *after advices* implicitly proceed associated to the advised classes' methods execution, i.e., *before* or after advices must execute and then the advised classes can continue their execution. Like for the *around advices* behavior, this proposal considers messages for an explicit activation from aspects to class instances to proceed. Given these ideas, rules to model JPI program-behavior execution scenarios by means of JPI sequence diagrams correctly are:

I. Object and aspects in any execution scenario are participants.
II. As usual, objects participants communicate by synchronous and asynchronous messages represented by ⟶ and →, respectively. A participant that sends a synchronous message waits for a return message, represented by - - >, from the target object to continue its actions.
III. A participant can create or delete an existent object. For objects creation, a box that represents an object participant is linked to the creation-sent message. An arrow like a return message represents a creation-sent message. Destruction messages, synchronous or asynchronous, imply that the affected object will definitely end its activities and a cross at the bottom of its lifeline after its destruction represents this situation.
IV. When a participant receives a message, an activation gray line is created until it finishes its associated actions and returns.
V. When a class B exhibits a join point interface with a *pointcut* PC, and a participant *a'* of class *A* sends a message *M* to a participant *b'* of class *B* asking for a method involved in the PC rule, there will be a synchronization point in *b'* lifeline, if PC rule holds. A JPI message denotes the JPI method name and the values of its arguments. These values are usually

conformant with the advised method signature, i.e., matching the number and types of parameters. For example, message 3.0 of Figure 3 shows a JPI message *<<around>> JPIPreBuying(it, qty = units)*, i.e., *it* and *qty* are arguments of the *JPIPreBuying* method call, and *it* takes the value of *it* from the source participant, in this case from *sp1*, and *qty* takes the value of *units* from that source as well. In addition, advice <<around>> stereotypes the JPI message.

To preserve the UML sequence diagrams semantic, three important rules are proposed and applied here:

- 1st, a JPI message from a participant object *O* to a participant aspect *A* always appears after the invocation message of the advised method.
- 2nd, for any synchronized message *M* from a participant *X1* to a participant *X2*, then *M* requires a return message from *X2* to *X1*.
- 3rd, for *around* advices, *proceed* calls are nested non-return calls.

In addition, for a JPI message from an object to an aspect, the message signature must be conformant to the JPI interface, which includes details for the advice execution by the aspect, i.e., kind of advice, parameters of the method in the JPI interface along with their values. Before performing any action, advised object waits for a *proceed message* from the aspect.

*Procee*d messages associated to *before* and *after* advices are like return messages in OOP-languages, whereas *around* advices cause that *proceed* messages behave like nested calls in an imperative language.

In general, *proceed* messages are more like the "*pony express*" in the Old West that delivers an important information (e.g. "*paidPrice with discount*" in the 1st *proceed* message of Figure 4) to the encamped (waiting) cavalry commander (the method) just before conducting the "correct" attack (method execution) to the enemy.

Following preceding mentioned rules, since a JPI message is a synchronized message, for the previous described sequence of Figure 3, the first aspect-participant sends *proceed* message 3.1 to the participant object *sp1*, which then can perform its actions. A *proceed* message indicates the preserved and updated values of arguments important for the advised method to execute. For example, message 3.1 of Figure 3 shows a proceed message, *proceed(it = new Item("null product", 0, 0), units = 0)*, for the participant *sp1*, i.e., a new item instantiates the argument *it* of advised method whereas *units* has the value of 0.

With these rules, it is possible to model behavior of JPI programs for particular scenarios. Since UML sequence diagrams allow modeling global scenarios and algorithmic behavior by means of combined fragments, thus this modeling proposal for JPI programs behavior would permit to understand JPI programs participants and their interactions for the reviewed scenarios, i.e., what a JPI program does, to obtain a semantics understanding about model JPI programs.

Figure 3 shows a JPI UML sequence diagram for the scenario in which a frequent customer wants to buy a product not sold by the ShoppingSession system: action 1 shows a *TestDriver* object that obtains *sp1*, an instance of *ShoppingSession*, for a frequent Costumer *c1 = {2, 'Cristian'}* who wants to buy 15 units of the item *b1*, a not in stock item, action 2.0 represented by the message *buying(it = b1, units = 15)* from *TestDriver* to *sp1*. Next, action 3.0 represents the <<around>> advice *JPIPreBuying(it, qty = units)* activation for the *PreBuying* aspect, meanwhile the action 3.1 represents a *proceed message* from *PreBuying aspect* that changes values of arguments *it* and *units* of the *sp1's* advised method, i.e., *it = new item("null product", 0, 0), units =0*. Action 4.0 takes into account the <<around>> advice in message *JPIDiscount(paidPrice=it.getPrice(), ss=this)* from *sp1* to an instance of aspect *Discount*. By action 4.1, aspect *Discount* sends a message to one of its methods *freqCustomer(ss.getCustomer())*; and action 4.2 represents a *proceed* message from *Discount* aspect to *sp1*. The result of this *proceed* message is to update *paidPrice* to *0.9*paidPrice* whereas the price of stocked item *ss* remains invariant. Action 5.0 follows an <<around>> advice without arguments from *sp1* to the aspect *Logger*. Action 5.1 is a *proceed* message from the aspect *Logger* to *sp1*. Action 6.0 is a message that creates a new *Transaction* instance *t1* with arguments *it*, *qty*, and *paidPrice*; and action 6.1, *return t1*, is a message from the created *Transaction* object to *sp1* that returns itself. After these actions, since proceed messages return in a LIFO order like nested procedure calls, and previous <<around>> messages have not returned yet, action 5.2 returns *t1* from *sp1* to the *Logger* aspect, and action 5.3, after transaction *t1* gets in the log, returns *t1* from *Logger* aspect to *sp1*. Likewise, action 4.2 returns *t1* from *sp1* to aspect *Discount*, and action 4.3 returns *t1* from aspect *Discount* to *sp1*. Since the latter is a "*null item*", the stock of item *it* does not change (decremented by 0). Likewise, action 3.2 is a return message, in this case, the message returns *idTrans=t1.getIdTrans()*, from *sp1* to *PreBuying* aspect, and message 3.3 returns from *PreBuying* aspect to *sp1* again. Finally, message 2.1 returns *idTrans* from *sp1* to *TestDriver* instance which sends message 7.0 to method *TranDetails(idTrans)* of the *Logger a*spect. Message 7.1 gives back the execution control to instance of *TestDriver* and the execution scenario finishes.

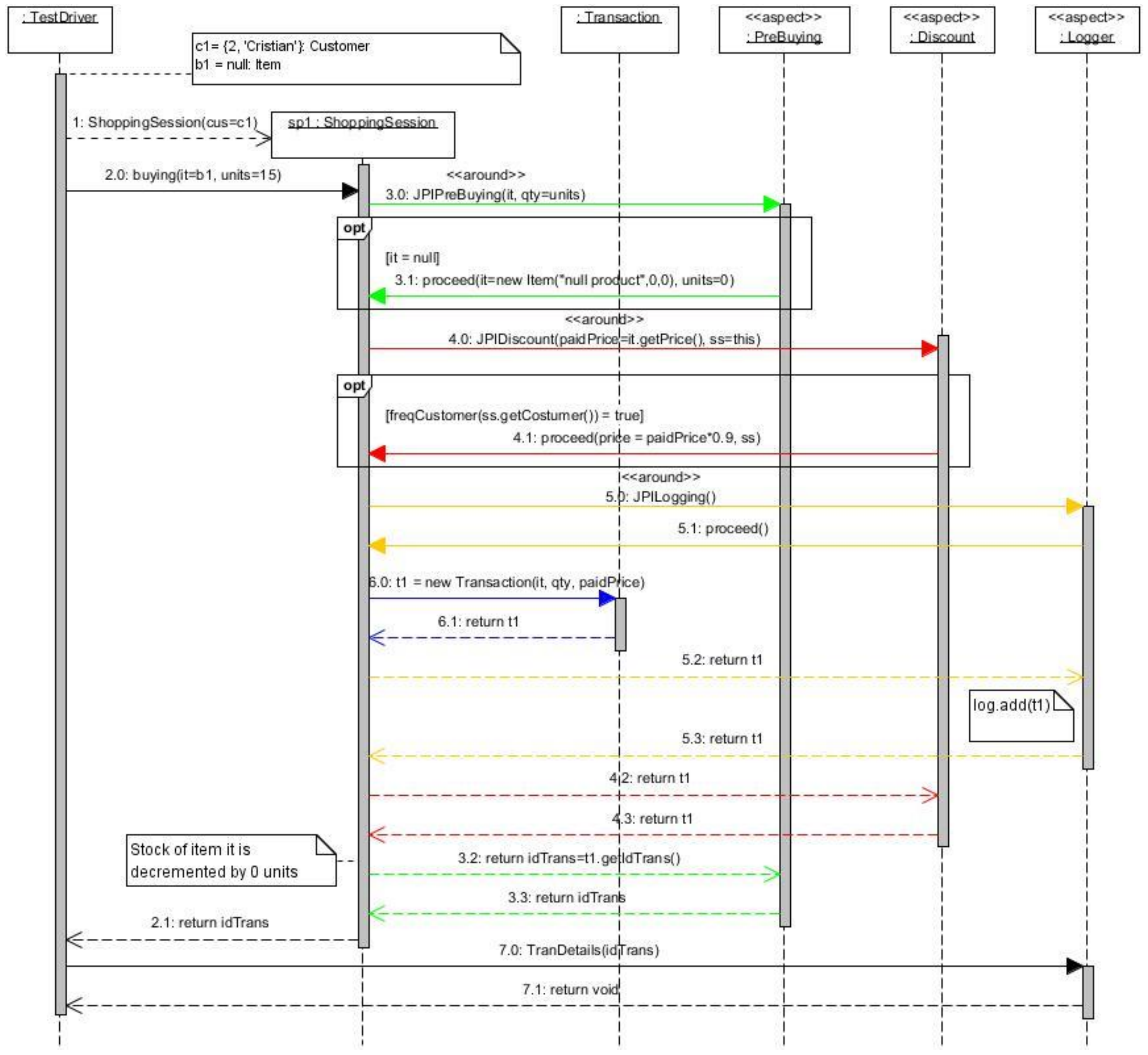

: TestDriver
: Transaction
<<aspect>> : PreBuying
<<aspect>> : Discount
<<aspect>> : Logger
c1= {2, 'Cristian'}: Customer
b1 = null: Item
1: ShoppingSession(cus=c1)
sp1 : ShoppingSession
2.0: buying(it=b1, units=15)
<<around>>
3.0: JPIPreBuying(it, qty=units)
opt
[it = null]
3.1: proceed(it=new Item("null product",0,0), units=0)
<<around>>
4.0: JPIDiscount(paidPrice=it.getPrice(), ss=this)
opt
[freqCustomer(ss.getCostumer()) = true]
4.1: proceed(price = paidPrice*0.9, ss)
<<around>>
5.0: JPILogging()
5.1: proceed()
6.0: t1 = new Transaction(it, qty, paidPrice)
6.1: return t1
5.2: return t1
log.add(t1)
5.3: return t1
4.2: return t1
4.3: return t1
Stock of item it is decremented by 0 units
3.2: return idTrans=t1.getIdTrans()
3.3: return idTrans
2.1: return idTrans
7.0: TranDetails(idTrans)
7.1: return void

Fig. 3. JPI UML sequence diagram for a frequent customer buying a product not sold by the ShoppingSession system

Figures 4, 5, and 6 describe other execution scenarios for the ShoppingSession system, which semantic is similar to the Figure 3 described semantic.

Next section presents part of the *ShoppingSession* system code, to review and verify its functioning according to the described modeled scenarios.

IV. ANAYZING JPI CODE

Figure 7 presents the code of class *ShoppingSession* that exhibits the join point interfaces *JPIPreBuying*, *JPIDiscount*, and *JPILogger*: execution of method *buying(..)* exhibits *JPIPreBuying* whereas the call of constructor of class *Transaction* exhibits *JPIDiscount* and *JPILogger*. Figures 8, 9, and 10 show the aspects *PreBuying*, *Discount*, and *Logger* to implement mentioned join point interfaces. In addition, Figure 11 presents the code for the join point interfaces definition. Clearly, this code solution structurally represents associations and components of JPI UML class diagram of Figure 2.

In the functioning logic analysis of the code class *ShoppingSession* and its exhibited aspects, i.e., aspect *PreBuying*, *Discount*, and *Logger*, a clear hegemony exists to the functioning logic of the sequence diagrams of Figures 3, 4, 5, and 6. When a frequent costumer buys units of an item *it*, Figure 3, 4, and 5 show the behavior of class *ShoppingSession* and its exhibited aspects when the item *it* represents a *null item*, item *it* is an item in stock, and item *it* is an item without enough units in stock, respectively. Figure 3 shows a sequence diagram that includes a UML 2.0 *opt* combined fragment, between class *ShoppingSession* and aspect *PreBuying*, with a constraint for item *it*, when *it* is a *null item*. For this scenario, aspect *PreBuying* instantiates item *it* to an item named "*null item*", and proceeds with the new value of *it* and *units = 0*, i.e., buying 0 units of item *it*. Figure 4 shows a sequence diagram that includes an *opt* combined fragment between class *ShoppingSession* and aspect *PreBuying*, with a constraint for item *it* that is fulfilled, i.e., item *it* !=null and it.getUnits() >= *qty*, *qty* represents the *units* argument in *PreBuying* aspect. For the latter scenario, aspect *PreBuying* proceeds with *it* and *qty* actual values without changes. Figure 5 shows a sequence diagram that includes an *opt* combined fragment, between class *ShoppingSession* and aspect *PreBuying*, with a constraint for item *it* that is not fulfilled in that scenario, though there are enough units in stock, i.e., item *it* is not a *null item*, but *it.getUnits() < qty*. For this scenario, Figure 5 shows that aspect PreBuying proceeds with item *it* and *units = 0*.

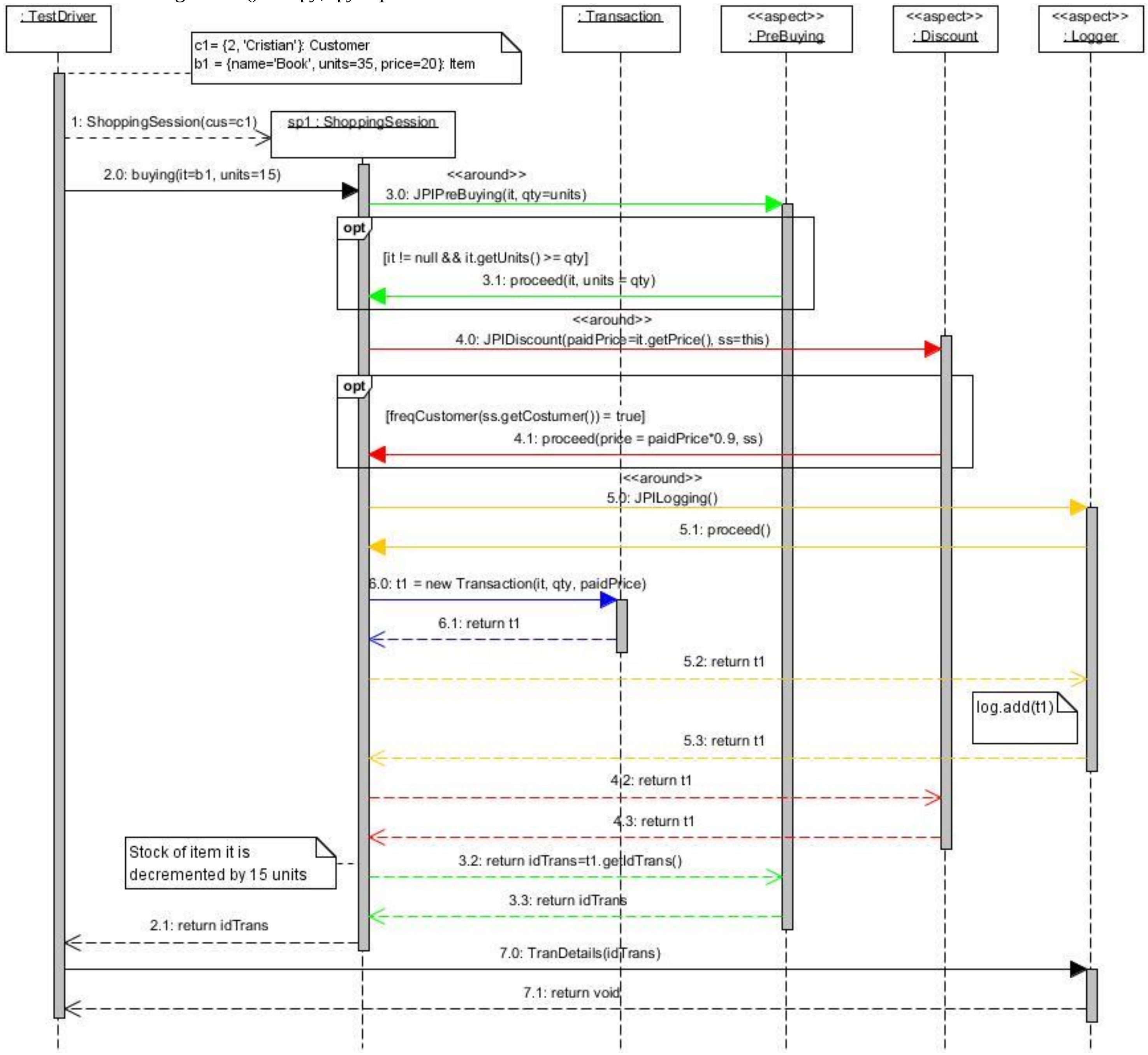

: TestDriver
: Transaction
<<aspect>> : PreBuying
<<aspect>> : Discount
<<aspect>> : Logger
c1= {2, 'Cristian'}: Customer
b1 = {name='Book', units=35, price=20}: Item
1: ShoppingSession(cus=c1)
sp1 : ShoppingSession
2.0: buying(it=b1, units=15)
<<around>>
3.0: JPIPreBuying(it, qty=units)
opt
[it != null && it.getUnits() >= qty]
3.1: proceed(it, units = qty)
<<around>>
4.0: JPIDiscount(paidPrice=it.getPrice(), ss=this)
opt
[freqCustomer(ss.getCostumer()) = true]
4.1: proceed(price = paidPrice*0.9, ss)
<<around>>
5.0: JPILogging()
5.1: proceed()
6.0: t1 = new Transaction(it, qty, paidPrice)
6.1: return t1
5.2: return t1
log.add(t1)
5.3: return t1
4.2: return t1
4.3: return t1
Stock of item it is decremented by 15 units
3.2: return idTrans=t1.getIdTrans()
3.3: return idTrans
2.1: return idTrans
7.0: TranDetails(idTrans)
7.1: return void

Fig. 4. JPI UML sequence diagram for a frequent customer buying a product in stock at the ShoppingSession system

When a non-frequent costumer buys *qty* units of an item *it* in stock, such as Figure 6 shows, aspect PreBuying proceeds without changing *it* and *qty* values.

Concerning aspect Discount that receives *paidPrice*, i.e., the price for the new item, and *ss*, the equivalent instance of ShoppingSession, such as Figures 3, 4, and 5 show, for a frequent costumer, the constraint of the second *opt* combined fragment of these figures is fulfilled, thus aspect Discount proceeds with *price* = *paidPrice**0.9, and preserves the *ss* value. Thus, there is always a discount for transactions performed by a frequent costumer. However, for a non-frequent costumer, Figure 6 shows that a discount does not apply on the paid price.

Regarding aspect *Logger*, for the mentioned scenarios, this aspect logs final values for each transaction, i.e., log of values of transactions after being updated by aspects *PreBuying* and *Discount*. Since each of these behaviors is modeled by JPI UML sequence diagrams, and they are consistent to the code of classes and aspects, the behavior of aspects *PreBuying*, *Discount*, and *Logger* is consistent with the functioning logic of *ShoppingSession* system execution scenarios expressed in the JPI UML sequence diagrams of Figures 3, 4, 5, and 6.

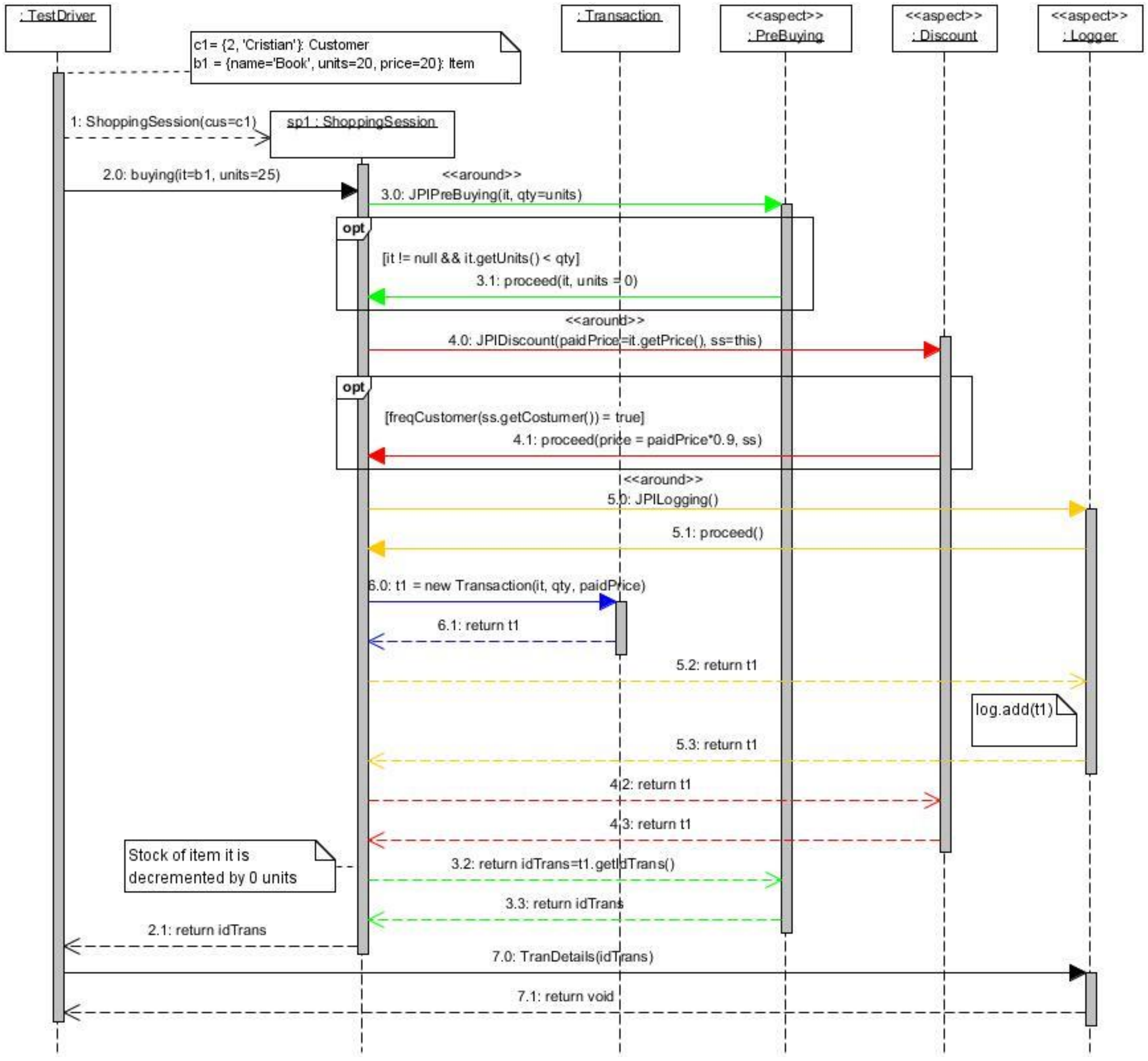

: TestDriver
: Transaction
<<aspect>> : PreBuying
<<aspect>> : Discount
<<aspect>> : Logger
c1= {2, 'Cristian'}: Customer
b1 = {name='Book', units=20, price=20}: Item
1: ShoppingSession(cus=c1)
sp1 : ShoppingSession
2.0: buying(it=b1, units=25)
<<around>>
3.0: JPIPreBuying(it, qty=units)
opt
[it != null && it.getUnits() < qty]
3.1: proceed(it, units = 0)
<<around>>
4.0: JPIDiscount(paidPrice=it.getPrice(), ss=this)
opt
[freqCustomer(ss.getCostumer()) = true]
4.1: proceed(price = paidPrice*0.9, ss)
<<around>>
5.0: JPILogging()
5.1: proceed()
6.0: t1 = new Transaction(it, qty, paidPrice)
6.1: return t1
5.2: return t1
log.add(t1)
5.3: return t1
4.2: return t1
4.3: return t1
Stock of item it is decremented by 0 units
3.2: return idTrans=t1.getIdTrans()
3.3: return idTrans
2.1: return idTrans
7.0: TranDetails(idTrans)
7.1: return void

Fig. 5. JPI UML sequence diagram for a frequent customer buying a product without enough stock at the ShoppingSession system

V. Related Work

As was mentioned, AOP represents a software development paradigm to modularize crosscut behavior [5]. For modeling traditional AOP solutions, [10] presents an UML use case application and extension of the formal language AspectZ for the aspect-oriented software requirements specification and analysis. Likewise, [11] describes an aspect-oriented UML class diagram-based and the OOAspectZ formal language for the software structure and requirements specification. In general, [14] surveys UML-based aspect-oriented design approach. Thus, mentioned research does not involve JPI ideas.

For JPI UML-based modeling, [13] presents an AspectZ extension, JPIAspectZ, for the formal modeling of JPI software requirements. Thus, given the JPI benefits for the modular software production, this research is of a high value looking for a complete JPI software development process.

VI. Conclusions

JPI is a novel aspect-oriented programming methodology that permits solving classical issues of traditional aspect-oriented programming, i.e., implicit dependencies among classes and aspects. Nevertheless, as traditional aspect-oriented programming, elements such as JPI UML or JPI formal languages like JPI AspectZ [10] [11] [13] do not exist to perform a complete software development process inspired by JPI methodological practices. To partially solve these issues, this article has proposed and applied as well, JPI UML diagrams, i.e., JPI UML class diagrams and JPI UML sequence diagrams, respectively, for modeling structure and behavior of software applications developed using JPI to ease aspect-based programming.

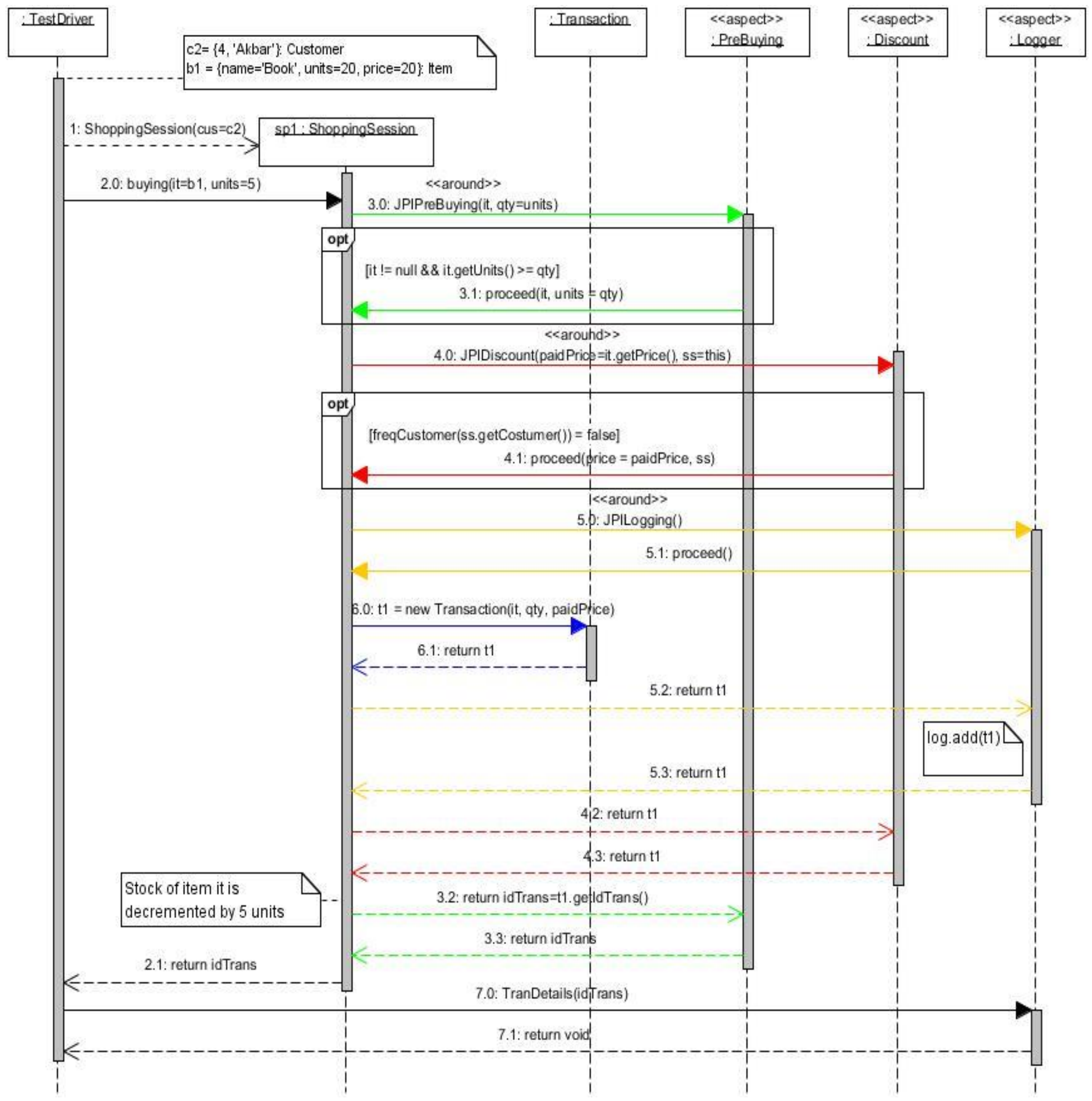

: TestDriver
c2= {4, 'Akbar'}: Customer
b1 = {name='Book', units=20, price=20}: Item
: Transaction
<<aspect>> : PreBuying
<<aspect>> : Discount
<<aspect>> : Logger
1: ShoppingSession(cus=c2)
sp1 : ShoppingSession
2.0: buying(it=b1, units=5)
<<around>>
3.0: JPIPreBuying(it, qty=units)
opt
[it != null && it.getUnits() >= qty]
3.1: proceed(it, units = qty)
<<around>>
4.0: JPIDiscount(paidPrice=it.getPrice(), ss=this)
opt
[freqCustomer(ss.getCostumer()) = false]
4.1: proceed(price = paidPrice, ss)
<<around>>
5.0: JPILogging()
5.1: proceed()
6.0: t1 = new Transaction(it, qty, paidPrice)
6.1: return t1
5.2: return t1
log.add(t1)
5.3: return t1
4.2: return t1
4.3: return t1
Stock of item it is decremented by 5 units
3.2: return idTrans=t1.getIdTrans()
3.3: return idTrans
2.1: return idTrans
7.0: TranDetails(idTrans)
7.1: return void

Fig. 6. JPI UML sequence diagram for a non-frequent customer buying a product in stock at the ShoppingSession system

JPI UML class diagrams allow capturing main modules of JPI programs, i.e., classes and join point interfaces, and associations between them. The presented JPI UML proposal clearly established associations among classes and join point interfaces, such as direction, stereotypes for different kind of advices, and *pointcu*t rules. By mean of JPI class diagrams, one can know and understand existing relationships among classes and join point interfaces.

JPI UML sequence diagrams capture the functioning logic of modeled execution scenarios of a JPI program, and by means of these diagrams, we hypothesize that the functioning of a program can be deduced. Our proposal used ***opt*** combined fragments to zoom conditions and behavior for the functioning logic of aspects. After applying JPI UML sequence diagrams and analyzing the code of the modeled program for the *ShoppingSession* example, this article has shown consistency between models and the program code derived by means of our methodological approach. Clearly, using JPI UML diagrams, there is a functioning logic hegemony between modeled execution scenarios and code of main class and the exhibited aspects. This issue permits continuing researching to look for a full JPI software development process.

References

[1] E. Bodden, "Closure Joinpoints: Block Joinpoints without Surprises," *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development, AOSD '11*. ACM, New York, NY, USA, pp. 117-128, March 2011.

[2] E. Bodden, E. Tanter, M. Inostroza, "A Brief Tour of Join Point Interfaces," *Proceedings of the 12th Annual International Conference Companion on Aspect-Oriented Software Development, AOSD '13 Companion*. ACM, New York, NY, USA, pp. 19–22, March 2013.

[3] E. Bodden, E. Tanter, M. Inostroza, "Join point interfaces for safe and flexible decoupling of aspects," ACM Transactions on Software Engineering and Methodology, ACM, New York, NY, USA, pp. 1–41, February 2014

[4] B. Griswold, E. Hilsdale, J. Hugunin, W. Isberg, G. Kiczales, M. Kersten, "Aspect-Oriented Programming with AspectJ™," *AspectJ.org, Xerox PARC, 2001*. Tutorial slides online at http://www.cse.msu.edu/sens/Software/aspectj/aspectj1.0.4/doc/tutorial.pdf [Accessed: 25-Sep-2015]

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, J. Irwin, "Aspect oriented programming," Proceeding of the *European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag LNCS 124, Finland, June 1997.

[6] G. Kiczales, M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*. ACM, New York, NY, USA, pp. 49-58, May 2005.

[7] T. Pender, "UML Bible," *John Wiley & Sons, Inc.*, New York, NY, USA, 1 Edition, 2003.

[8] L. Ramnivas, "AspectJ in Action: Practical Aspect-Oriented Programming," *Manning Publications Co.* Greenwich, CT, USA, 2003.

[9] F. Steimann, "The Paradoxical Success of Aspect-oriented Programming," Proceedings of the *21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06*, Portland, Oregon, USA, October 2006 .

[10] C. Vidal, R. Saens, C. Del Rio, R. Villarroel, "Aspect-Oriented Modeling: Applying Aspect-Oriented UML Use Cases and Extending AspectZ," *Computing and Informatics Journal*, Bratislava, Slovak , pp. 573-593, 2013.

[11] C. Vidal, R. Saens, C. Del Rio, R. Villarroel, "OOAspectZ and Aspect-Oriented UML Class Diagrams," *Ingeniería e Investigación Journal*, Medellín, Colombia, pp. 66-71, 2013.

[12] C. Vidal, R. Villarroel, "JPI UML: JPI class and sequence diagrams for aspect-oriented JPI applications," Proceedings of XXXIII *International Conference of the Chilean Computer Society*, Talca, Chile, November 2014.

[13] C. Vidal, R. Villarroel, C. Pereira, "JPIAspectZ: A formal specification language for aspect-oriented JPI applications," Proceedings of XXXIII *International Conference of the Chilean Computer Society*, Talca, Chile, November 2014.

[14] M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, E. Kapsammer, "A survey ofn UML-based aspect-oriented design modeling," *Journal ACM Computing Surveys CSUR*, New York, NY, USA. vol.43, issue 4, pp. 1-28, 2011.

```
package classes;
import java.util.*; import joinpointinterfaces.*;

public class ShoppingSession {
  private HashMap<Integer, Transaction> ShoppingSessionTrans;
  private Customer cus;

  exhibits Integer JPIPreBuying(Item it, int qty): execution(Integer buying(..)) && args(it, qty);
  exhibits BuyTransaction JPIDiscount(double price, ShoppingSession ss): call(BuyTransaction.new(..)) &&
                                                                        args(*, *, price) && this(ss);
  exhibits BuyTransaction JPILoggingBuy(): call(BuyTransaction.new(..));

  public ShoppingSession(Customer acus){...}
  public Customer getCustomer(){ return cus;}
  public void closingSession(){ ...}

  public Integer buying(Item it, int units){
    BuyTransaction buyTrans; Integer key;
    buyTrans = new BuyTransaction(it, units, it.getPrice());
    key = buyTrans.getIdTrans();
    ShoppingSessionTrans.put(key, buyTrans);
    it.setUnits(it.getUnits()-units);

    return key;
   }
 ...
}
```

Fig. 7. Class ShoppingSession code of the ShoppingSession system

```
package aspects;
import classes.*; import joinpointinterfaces.*;
public aspect PreBuying{
  Integer around JPIPreBuying(Item it, int qty){
      if (it != null){
         if (qty <= it.getUnits())
              return proceed(it, qty);
         }
      else
         it = new Item("null product", 0, 0);
      return proceed(it, 0);
      }
}
```

Fig. 8. Aspect PreBuying of the ShoppingSession system

```
package aspects;
import classes.*; import joinpointinterfaces.*;
public aspect Discount {
  /*To estabish aspects precedence*/
  declare precedence: Discount, Logger;
  final String freqCustomers[] = {"Laurie", "Cristian"};
  boolean frequentCostumer(String N){
    for(int i=0;i<freqCustomers.length; i++){
        if (freqCustomers[i].equals(N))
           return true;
        }
        return false;
   }
  BuyTransaction around JPIDiscount(double paidPrice, ShoppingSession ss){
    double factor = 1;
    if (frequentCostumer(ss.getCustomer().getName()))
        factor = 0.9; return proceed(paidPrice*factor, ss);
  }
}
```

Fig. 9. Aspect Discount of the ShoppingSession system

```
package aspects;
import java.util.*; import classes.*; import joinpointinterfaces.*;

public aspect Logger {
  private static HashMap<Integer, Transaction> log = new HashMap<Integer, Transaction>();
  BuyTransaction around JPILoggingBuy(){
    BuyTransaction BT = proceed();
    if (BT.getQuantity()==0)
       //Non-Successful Logging
    else
       //Successful Logging

    log.put(BT.getIdTrans(), BT);

    return BT;
   }
  public void ListLogger(){ ... //List of Transactions}
  public static void TranDetails(Integer idTrans){ ... // Details of Transaction idTrans}
}
```

Fig. 10. Aspect Logger of the ShoppingSession system

```
package joinpointinterfaces;
import classes.*;

jpi BuyTransaction JPIDiscount(double paidPrice, ShoppingSession SS);
jpi BuyTransaction JPILoggingBuy();
jpi Integer JPIPreBuying(Item it, int qty);
```

Fig. 11. JPI instances of the ShoppingSession system