Events in Use Cases as a Basis for Identifying and Specifying Classes and Business Rules

published in TOOLS 29 (TOOLS Europe 99) Conference, 7-10 June 1999, Nancy, France, pp. 204-213.

Danny C.C. Poo (PhD)
Department of Information Systems
School of Computing
National University of Singapore
Lower Kent Ridge Road
Singapore 119260
e-mail : dpoo@comp.nus.edu.sg <Danny Poo>

Abstract

Business rules are closely associated with events. This paper describes how events in use cases can be the basis for identifying classes and business rules. A process known as Event Scripting is used to document event and from it objects and their relationships are identified. Business rules identified with the events are attached to objects as part of their definitions in class specifications. The Event Scripting process is described in detail in this paper.

1. Introduction

A commonly recommended technique for identifying objects in object-oriented modeling is to search any documentation or requirement specifications for nouns that represent potential objects. One obvious problem with such an approach is the difficulty involved with having to deal with a large number of nouns.

Use Case Modeling, a business modeling technique proposed by Jacobson [2], provides an alternative approach. Instead of looking for nouns, the approach first breaks down the entire scope of system functionality into a number of use cases. A use case is a sequence of behaviorally related events that flow through a system [2,3].

Use Cases have been used to describe business processes and identify objects and their relationships. Since use cases are smaller statements of system functionality, the process of identifying objects by underlining nouns is made simpler and more efficient.

Business policy, according to Palmer [8] "is a set of rules, regulations, invariant and definitions that govern the behavior of an enterprise". Business rules have received a lot of attention and their importance in business information system modeling, specification and representation has been highlighted in the literature (see [5, 6, 7, 8, 11, 12]).

According to Rosca et al [11], business rules are formulated by business people to have planned consequences that contribute to the success of the business. Two similar applications therefore may not behave in the same manner because of their differing set of business policies, even though they may be defined by the same classes of objects in the same domain.

Business rules are close to the heart of a business. Rosca et al further noted in [11] "Business rules are characterized by their strategic importance to the business and consequently deserve special consideration. They emphasize certain characteristics that imply how they should be dealt with."

In identifying objects from use cases, business policies or rules in a use case description should therefore be given due consideration. In this paper, we shall discuss an extension to Use Case Modeling that takes into consideration business rule statements in use cases and suggest how they should be treated in the specification of *domain classes*.

This paper is organized in the following manner:

- Section 2: discusses Event Scripting as a technique for analyzing events.
- Section 3: discusses an approach for identifying objects/classes from Event Scripts.
- Section 4: discusses class specifications.
- Section 5: discusses how relationships between classes are modeled.
- Section 6: concludes the paper.

2. Events and Use Cases

A Use Case consists of events [3] and events are connected with business rules (see [5]). An *event* is a set of activities that are performed either fully (when the pre-conditions are satisfied) or not at all (when one or more of the pre-conditions is/are not satisfied) [3]. An event is invoked by a stimulus which can be an actor or by a point in time reached in the system [3]. An event has an effect on the state of a system. The effect may be in the form of creating or deleting objects, or changing the state of existing objects [10].

2.1. Event Scripting

Analyzing use cases can be reduced to analyzing events. In this paper, a process known as Event Scripting is used to describe and analyze events. Objects and their relationships are identified in the process.

Event Scripting is an extension to Use Case Modeling. It adds value to the business modeling process, just like extensions as proposed by Stanford in [12]. Event Scripting serves three purposes: to provide an approach for modeling events, to identify objects, classes, attributes and operations, and to identify business rules associated with events.

An event script is produced for each event identified in a use case. We shall illustrate the concept of Event Scripting using an example.

A "Returning Items" Use Case from a Library Application Domain is given in Textbox 1. The event identified from the Use Case is "Return a Non-Reference Book/Bound-Periodical" since it causes a change in the state of the system – the item returned is now in the library and the member does not have the loan of the item anymore.

An event is treated as happening at an instantaneous point in time. It is the point in time when a member (the stimulus or source) returns an item (a non-reference book or a bound-periodical in this case) to the library. Based on this definition, if a member returns 3 books, then there would be three separate instances of the event, even though the events are similar.

As can be seen from the use case description, the returning of a library item is associated with some business policies. Generally, business policies associated with events can be categorized into Pre-Event Conditions and Post-Event Conditions, Post-Event

Triggers, and Derivative Policies. Martin calls these policies Integrity Rules, Behavior Rules and Derivation Rules respectively [5].

```
This Use Case is started when a library member enters the library. A
library member can be a student member or a staff member. He/she
goes to the counter with the items to return. An item can be a non-
reference book or a bound-periodical (current issue of unbound
periodicals are not borrowable). A library staff at the counter
receives from the member each of the item he/she intends to return.
An item return is considered valid if the member had earlier
borrowed the item. An item returned may be overdue and if it is, the
amount of fine due is computed and made known to the member. The
member pays the fine and receives a receipt in return. If the item
returned is reserved by another member, a notification letter to
inform the member of the availability of the item is produced. When
all the items had been returned, the Use Case is said to be
complete.
```

Textbox 1. Returning Items Use Case

Event Scripting provides a structured approach to understanding events and their associated policies. Event scripts are semi-formal description of events produced in the Event Scripting process. Event scripts are designed as a structured means for discussing about an event between analysts and users. They can be used to help analysts understand more about an event, identify objects, classes, attributes and operations, and document and analyze business rules.

2.2. Event Scripts

The “Return a Non-Reference Book/Bound-Periodical” event can be described in an event script as shown in Textbox 2. It has the following sections: Event Name, Brief Meaning, Source, Participant Sets, Pre-Event Conditions, Inputs, Changes, Post-Event Triggers, Derivative Policies.

Event Name: This section states the name of the event described in the script.

Brief Meaning: This section states the gist of the event in brief.

Source: The originator of the event is specified here. The source here is the Member and not the Counter Staff who is the one that actually receives the item for returning. The Counter Staff is an example of a facilitator of an event and should be distinguished from the actual source.

Participant Sets: Each set specifies similar object types that can participate in the event. An event is participated by one or more object types and similar objects may participate in the same type of events across multiple instances of the same event. For example, a Staff Member and a Non-Reference Book may participate in an event “Return a Non-Reference Book/Bound-Periodical” (i.e. a staff member returns a non-reference book at the counter); an Under-Graduate Student and a Non-Reference Book may similarly participate in a separate instance of this event, and so on. Instead of describing each event in a separate event script, leading to multiple but similar event descriptions, a Participant Sets section is created to include all object classes that can participate in such an event. This will greatly reduce the complexity of Event Scripting for similar events. Information in the Participant Sets section can be used to identify class hierarchy at a later stage. In the above example, Staff, Under-Graduate Student, or Post-Graduate Student members

participate with a Non-Reference Book or Bound-Periodical object in one or more such events; they are thus included in the Participant Sets section.

Textbox 2. An Event Script

```
Event:
Return a Non-Reference Book/Bound-Periodical
Brief Meaning:
A member returns a book/bound-periodical to the library after a loan period.
Source:
Member
Participant Sets:
Member {Staff, Post-Graduate Student, Under-Graduate Student}
Item {Non-Reference Book, Bound-Periodical}
Pre-Event Conditions:
Member has earlier borrowed the book/bound-periodical.
Inputs:
1.  membership number
2.  accession number
3.  due-date
4.  loan-count
5.  status
Changes:
1. Member's loan-count is decremented by 1.
2. Item's status is changed to "available".
Post-Event Triggers:
1.  If item's loan is overdue then compute fine amount.
2.  If item is reserved then print a notification letter.
Derivative Policies:
1. Item's loan is overdue IF number-of-days-overdue>0.
2. number-of-days-overdue = today - due-date.
3. Fine amount = FineUpTo10Days IF number-of-days-overdue <= 10.
4. Fine amount = FineUpTo10Days + FineMoreThan10days IF number-of-days-overdue > 10.
5. FineUpTo10Days = staff-rate-10days-Book * number-of-days-overdue IF (member is a
   Staff) AND (item is a Non-Reference Book).
6. FineUpTo10Days = staff-rate-10days-Periodical * number-of-days-overdue IF (member
   is a Staff) AND (item is a Bound-Periodical).
7. staff-rate-10days-Book = 50.
8. staff-rate-10days-Periodical = 300.
9. FineMoreThan10days = staff-rate-MoreThan10days-Book * (number-of-days-overdue -
   10) IF (member is a Staff) AND (item is a Non-Reference Book).
10.FineMoreThan10days = staff-rate-MoreThan10days-Periodical * (number-of-days-
   overdue - 10) IF (member is a Staff) AND (item is a Bound-Periodical).
11.staff-rate-MoreThan10days-Book = 100.
12.staff-rate-MoreThan10days-Periodical = 600.
13.FineUpTo10Days = post-underGrad-rate-10days-Book * number-of-days-overdue IF
   (member is a Post-Graduate Student OR Under-Graduate Student) AND (item is a Non-
   Reference Book).
14.FineUpTo10Days = post-underGrad-rate-10days-Periodical * number-of-days-overdue
   IF (member is a Post-Graduate Student OR Under-Graduate Student) AND (item is a
   Bound-Periodical).
15.post-underGrad-rate-10days-Book = 30.
16.post-underGrad-rate-10days-Periodical = 100.
17.FineMoreThan10days = post-underGrad-rate-MoreThan10days-Book * (number-of-days-
   overdue - 10) IF (member is a Post-Graduate Student OR Under-Graduate Student)
   AND (item is a Non-Reference Book).
18.FineMoreThan10days = post-underGrad-rate-MoreThan10days-Periodical * (number-of-
   days-overdue - 10) IF (member is a Post-Graduate Student OR Under-Graduate
   Student) AND (item is a Bound-Periodical).
19.post-underGrad-rate-MoreThan10days-Book = 50.
20.post-underGrad-rate-MoreThan10days-Periodical = 200.
```

Pre-Event Conditions: This section states the conditions that must hold before the event in consideration can take place. Only pre-event conditions are specified in event scripts. Post-Event Conditions have been omitted since statements in the "Changes" section implicitly define post event conditions. In the above example, the condition states that a

Member object must have previously participated in a "Borrow a Non-Reference Book/Bound-Periodical" event.

Inputs: This section states the information that is required for the execution of the event. The information in this example suggests that membership number is required to indicate which member is involved in the event. Similarly, accession number is used to identify the library item that participates in the event. The other inputs (`due-date, loan-count,` and `status`) are information that are updated by the event. From the above example, it is clear that the `Inputs` section information can be used to define object attributes of classes identified from event scripts.

Changes: This section states the changes effected as a result of the event. It includes statements highlighting the effects of the event. These effects are translated into changes on object attributes. Such state-changing statements can later be used to identify objects and their attributes. In this example, the `loan-count` of a member is decremented by 1 and the `status` of an item indicates that the item is now available for loan.

Post-Event Triggers: This section states the *conditions* for triggering an *operation* after the event is successfully completed. In this example, the operation "compute fine amount" is triggered if the item is returned later than when it was supposed to be returned. Also, if there is a reservation on the item just returned a notification letter is produced.

Derivative Policies: This section states how a value is computed or derived. For example, the first derivative policy states that a loan is overdue if the number of days overdue is greater than 0; the value "number of days overdue" is computed as the difference between today's date and the due date of the loan. Note that policies are chained with one another in definition. This will provide a means of ensuring all required definitions are available from the users. Derivative policies have the following structures:

```
A IF B where B is a boolean expression consisting of b1 AND b2 AND, etc.
C = D where D is an arithmetic expression that may include other derivative policy.
```

It is clear that event scripts are not formal specification of events. It is not intended to be so since event scripts are meant to help analysts have a better understanding of the business policies connected with events. Analysts should be able to write them in the manner that they have gathered from users. Although such an approach to specifying events may invite ambiguities, it is more structured and more conducive for communication between analysts and users. Martin in [5] suggested the use of diagrams as another means of depicting the relationships between events and rules.

3. Object/Class Identification

Object/class identification is an important activity in business modeling. Objects or classes identified form the fundamental structure of the final system. Information in event scripts is used to identify objects/classes, their attributes and operations, and their relationships with one another.

Information in the Source, Participant Sets, Pre-Event Conditions, and Changes sections are used to identify objects/classes. Objects that participate in an event are candidate objects for the object model. The rationale is that an event has an effect on the state of a system, and the state of the system is reflected by the aggregate state of the objects that constitute the system. Any changes as a result of the event are translated into

changes on the state of objects involved with the event. The participants in an event therefore are suitable candidates for the object model.

In the above example, Staff Member, Post-Graduate Student, Under-Graduate Student, Non-Reference Book, Bound-Periodical have been identified as candidate objects/classes. Member and Item in the Participant Sets are considered as potential superclasses for structuring the class hierarchy at a later stage.

Information for attribute identification can be found in the Pre-Event Conditions, Input, Changes, Post-Event Triggers and Derivative Policies sections. The following attributes are identified:

- for Staff Member, Post-Graduate Student Member and Under-Graduate Student Member – `membership number` and `loan-count`.
- for Non-Reference Book, Bound-Periodical – `accession number`, `due-date`, and `status`.

For each event an object participates in, an operation, known as an "action", is included in the class specification (see Section 4 below). This operation is a realization of the changes that are effected on the state of the object as a result of the object's participation in the event. At the class level, the action becomes the basis for representing business policies associated with the event. We thus have Pre-Action Conditions and Post-Action Triggers in place of Pre-Event Conditions and Post-Event Triggers in class specifications.

4. Class Specification

Information on objects, their attributes and operations are defined in a class specification. Textbox 3 contains the class specification for the class Non-Reference Book. A class specification is incrementally built up from information derived from event scripting as more events are considered. A class specification for Non-Reference Book is given in Textbox 3; it includes information derived from the above example and other event scripts. For obvious reason, the other event scripts are not elaborated here.

Class Name: name of the class.

Class Type: abstract or concrete class; an abstract class defines properties (attributes and operations) that are common and hence reusable by other classes; no instance of abstract classes is intended. Concrete classes are the actual classes defining the domain model; instances of object are created from them.

Superclasses: the classes that this class inherits properties from.

Subclasses: the classes that inherit properties from this class.

Whole: the class to which the objects of this class are a part of.

Parts: the classes of objects that are parts of this class.

Class Attributes: attributes specific to this class.

Object Attributes: attributes of objects of this class.

Actions: state-changing operations; there is one action for each event the objects of this class participate in.

Pre-Action Conditions: conditions that must be satisfied before an action can execute.

Post-Action Triggers: conditions for triggering an operation and the operation to be triggered after an action.

```
Class Name:
Non-Reference Book
Class Type:
Concrete
Superclasses:
Item
Subclasses:
nil
Whole:
nil
Parts:
nil
Class Attributes:
nil
Object Attributes:
 title, author, returnDate, dueDate, borrowCount
Actions:
 acquire()
 borrow()
 return()
 renew()
 discard()
Pre-Action Conditions:
 State-Transition Policies
  START, acquire, ACQUIRED
  ACQUIRED, borrow, BORROWED
  BORROWED, return, RETURNED
  BORROWED, renew, RENEWED
  RETURNED, borrow, BORROWED
  RENEWED, renew, RENEWED
  RETURNED, discard, DISCARDED
  DISCARDED, end, END
 Condition Policies
  BFORE acquire ENSURE true
  BEFORE borrow ENSURE true
  BEFORE renew ENSURE noReservation()
  BEFORE return ENSURE true
  BEFORE discard ENSURE true
Post-Action Triggers:
  AFTER acquire THEN DO NOTHING
  AFTER borrow THEN DO NOTHING
  AFTER renew THEN DO NOTHING
  AFTER return THEN IF hasReservation() THEN printNote()
  AFTER discard THEN DO NOTHING
Derivative Policies:
  overdue IF isOverdue()
  popular IF borrowCount > 50
  daysOverdue = numberOfDaysOverdue()
  reservationLimit = 5
Services:
  overdue()
  printNote()
  noReservation()
  hasReservation()
  isOverdue()
  numberOfDaysOverdue()
```

Textbox 3. Class Specification for Non-Reference Book

Derivative Policies: policies stating how a value is computed or derived.

Services: non-state-changing operations.

Note that Pre-Action Policies are made up of State-Transition Policies and Condition Policies. State-Transition Policies specify the valid state transitions an object may traverse through. For example, the state-transition "`RETURNED, borrow, BORROWED`" indicates that a "borrow" action can be executed after a "return" action has been successfully executed.

```
Class Name:
Undergraduate Member
Class Type:
Concrete
Superclasses:
Member
Subclasses:
nil
Whole:
nil
Parts:
nil
Class Attributes:
nil
Object Attributes:
name, faculty, matricNumber, loanCount, itemsBorrowed
Actions:
register()
borrow()
return()
renew()
terminateMembership()
Pre-Action Conditions:
 State-Transition Policies
  START, register, REGISTERED
  REGISTERED, borrow, BORROWED
  BORROWED, return, RETURNED
  BORROWED, renew, RENEWED
  BORROWED, borrow, BORROWED
  BORROWED, terminateMembership, TERMINATEDMEMBERSHIP
  RETURNED, borrow, BORROWED
  RETURNED, renew, RENEWED
  RETURNED, return, RETURNED
  RETURNED, terminateMembership, TERMINATEDMEMBERSHIP
  RENEWED, borrow, BORROWED
  RENEWED, renew, RENEWED
  RENEWED, return, RETURNED
  RENEWED, terminateMembership, TERMINATEDMEMBERSHIP
  TERMINATEDMEMBERSHIP, end, END
Condition Policies
  BEFORE Register ENSURE true
  BEFORE Borrow ENSURE loanCount < loanQuota
  BEFORE Return ENSURE true
  BEFORE Renew ENSURE true
  BEFORE TerminateMembership ENSURE noOutstandingFine() AND noOutstandingLoan()
Post-Action Triggers:
  AFTER Register THEN DO NOTHING
  AFTER Borrow THEN DO NOTHING
  AFTER Renew THEN DO NOTHING
  AFTER Return IF item.overdue THEN displayFine()
  AFTER TerminateMemberShip IF hasReservation() THEN cancelAllReservations()
Derivative Policies:
  loanQuota = 6
  fineAmount = item.daysOverdue * 10
  fineAmount = fineAmount10 + fineAmount10Plus IF item.daysOverdue > 10
  fineAmount = daysOverdue * 30 IF item.daysOverdue <= 10
  fineAmount10 = 10 * 30
  fineAmount10Plus = (item.daysOverdue - 10) * 50
Services:
  displayFine()
  noOutstandingFine()
  noOutstandingLoan()
  hasReservation()
  cancelAllReservations()
```

Textbox 4. Class Specification for Undergraduate Member

This means that a reference book cannot traverse through some of the states that a non-reference book can. This basically prevents the reference book from being borrowed by a member. The class specification as indicated here is clearly not formal but it does contain sufficient information for structuring a class, including the policies that are associated with it. How a class can be represented and implemented has been discussed in [9]. For completeness, we have included the class specification for an Under-Graduate Member class in Textbox 4. Note that the state-transition policies indicate whether an Under-Graduate Member can borrow or renew a book/bound-periodical:

```
BORROWED, borrow, BORROWED
```

However, how many book/bound-periodical a member may borrow is not specified in state-transition policies. That number is specified as a loan quota in terms of a derivative policy, e.g. loanQuota = 6.

From the above examples, we note that business policies can be incorporated into class specifications. This approach makes clear the association policies have with a class. Certainly, a policy may be connected with more than one class. In such a situation, the conventional dot notation is used to indicate membership. For instance, `fineAmount = item.daysOverdue * 10` indicates that the value fineAmount can be derived by taking the daysOverdue of the item (which can be a non-reference book or a bound-periodical) and multiply it by 10.

Operations are distinguished into actions and services. An *action* is a state-changing operation of an object. It exists because of the object's participation in an event. A *service*, on the other hand, is a non-state-changing operation. Its existence is not directly connected with events. An example of a service is an operation that enables other objects to access the data values of an object. Distinguishing actions from services provides a level of abstraction for distinguishing design-related functionality from domain-related requirements.

5. Relationships

Relationships are links connecting objects. Identification of relationships has been carried out intuitively in the conventional approach by examining the association that a class has with other classes. In this paper, we shall identify relationships based on objects' participation in events (as is practiced in the Jackson's System Development Method). From the "Return a Non-Reference Book" event script, we note that a Staff Member *returns* a Non-Reference Book. A Staff Member is thus said to be linked to a Non-Reference Book because of their participation in the "Return a Non-Reference Book" event (as shown in Figure 1 left diagram). Similarly, for a "Borrow a Non-Reference Book" event, a link is also established between a Staff Member class and a Non-Reference Book class as illustrated in Figure 1 right diagram.

Since these two links are semantically related, an associative relationship called "has loan" is used to represent the relationship between Staff Member and Non-Reference Book class instead (see figure 2 left diagram). In like manner, events such as "Reserve a Non-Reference Book" and "Cancel a reservation on a Non-Reference Book" result in a "has reservation" relationship (in this case, a member has reservation for a specific copy of a Non-Reference Book). The relationship model can be further enhanced to include cardinality constraints (as shown in Figure 2 right diagram). Further modeling may

translate “has loan” and “has reservation” relationships into classes, or as is recommended by Blaha et al [1], into association classes.

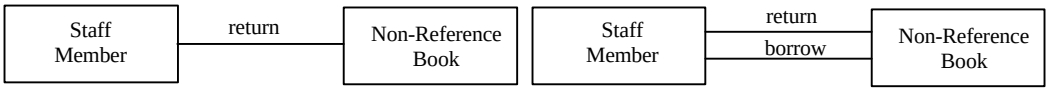

Figure 1. Relationship link between Staff Member and Non-Reference Book

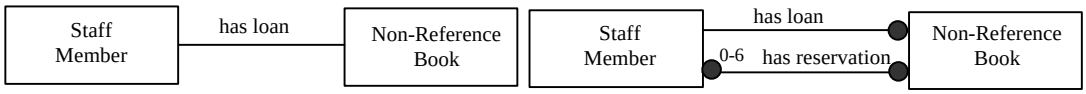

Figure 2. Relationship link between Staff Member and Non-Reference Book

6. Conclusion

This paper recognizes the importance of object identification in business modeling using the object-oriented development approach. It also recognizes the limitation of the commonly practiced approach of underlining nouns from any documentation or requirements specification. This limitation stems from the lack of consideration for business policies at the modeling stage. In this paper, we propose the extension of the Use Case Modeling approach to include business policies modeling. Events in use cases form the basis for identifying and specifying classes and business rules. A process known as Event Scripting is used to document event and from it objects and their relationships are identified. Business rules identified with the events are attached to objects as part of their definitions in class specifications. The paper describes this process in more detail using an example as an illustration.

References

[1] Blaha M. and Premerlani W. 1998. Object-Oriented Modeling and Design for Database Applications, Prentice-Hall.
[2] Jacobson I. 1992. Object-Oriented Software Engineering, Addison-Wesley.
[3] Jacobson I. 1994. The Object Advantage: Business Process Reengineering with Object Technology, Addison-Wesley.
[4] Kilov H. and Harvey W. (eds) 1996. Object-Oriented Behavioral Specifications, Kluwer Academic Publishers.
[5] Martin J. 1993. Principles of Object-Oriented Analysis and Design, Prentice-Hall.
[6] Martin J. and Odell J.J. 1995. Object-Oriented Methods: A Foundation, Prentice-Hall.
[7] Morgenstern L. 1996. Specifying and Reasoning about Business Rules in a Semantic Network in [4].
[8] Palmer J. 1996. Specifying Business Policy Using Agent-Contract Meta-Constructs, in [4].
[9] Poo, C.C.D., 1998. Policy Definition in Application-Domain-Related Classes, Proceedings of TOOLS Asia 98, Sept. 22-25, Beijing.
[10] Ramackers G. and Clegg D., 1995. Object Business Modeling, Requirements and Approach, Proceedings of the OOPSLA 95 Workshop, Oct 16, Texas, pp. 77-86.
[11] Rosca D., Greenspan S., Feblowitz M. and Wild C. 1997. A Decision Making Methodology in Support of the Business Rules, Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, Jan 6-10, Maryland, pp. 236-246.
[12] Stanford J. 1996. Enterprise Modeling with Use Cases in [4].