A Framework for Software Maintenance

Chiang-Choon Danny POO

Department of Information Systems and Computer Science
National University of Singapore, Kent Ridge Road,
Singapore 0511

Abstract. In describing a software system, there are three elements that are always considered : objects, functional requirements and business policies. In the traditional approach to software development, these elements are often mixed with one another in a system's definition in such a way that their meanings are embedded into the software, making their identification very difficult. This has the knock-on effect of making maintenance, and hence evolution, difficult.

This paper suggests a framework for addressing software maintenance and it calls for a clearer separation between the business policies, functional requirements and object models.

1. System Evolution is Inevitable

Software investment cost has been recognised by many as expensive. The high cost incurred is attributed not to the complexity of creating systems but to the maintenance efforts required to accommodate changes in inflexibly-designed systems [11,13].

The design of an information system often changes throughout its lifecycle; this has been attributed to changing users' requirements. These changes have been recognised to be intrinsic to software, often unpredictable and cannot be accommodated without iteration in the definition and development phases. Thus, the key to successful systems development, lies not in designing systems that satisfy the initial users' requirements but in the continuous provision for system evolution, particularly in the area of accommodating changes due to users' requests [1, 2,11].

2. The Object-Oriented Approach

The functional approach to information systems development has been the primary approach used for the past two decades. This approach is characterised by data flow and operations upon them. The structure of systems produced is based mainly on system functional activities. This approach has two major problems. First, system functionalities are highly volatile elements, and will inevitably lead to more frequent changes in system structure, which in turn translates to increased maintenance efforts. Second, systems produced using the functional approach have an architecture whose structure is characterised by a string of sequential functional operations. Any changes to these functionalities will create a chain reaction and the effect will be propagated throughout the system. Thus, the complexity involved in making such changes is usually not proportional to the requirements.

In recent years, there has been a shift in the way systems are designed: from a function-oriented to an object-oriented approach. The latter focuses on the objects that describe the problem domain and their mutual interactions. Unlike the functional approach, the primary

design issue of the object-oriented approach is no longer *what functionalities* the system does, but what *object* it does it to.

The primary motivation for adopting the object-oriented approach is the stable model of reference upon which a problem space can be examined. Objects in a problem domain are comparatively more stable than functional requirements [10]. For instance, the objects in a library or order processing environment today will probably be the same as the objects in such an environment a few years from now. Thus books and library users, and customers and orders will continue to be the principal objects in library and order processing environments, respectively, even though the functionalities of the applications may have changed over the years.

The object-oriented approach has several advantages over the traditional functional approach in terms of the *correctness, robustness, extensibility, reusability,* and *compatibility* of the final delivered system [12]. These key aspects of software quality are especially important because of the difficulties experienced with present-day systems development practices -- that is, programs often do not do what they are supposed to do; they are not equipped enough to deal with abnormal situations; they are not amenable to change; their construction does not rely on previous efforts; and they do not combine well with each other.

3. Object-oriented Software Structure

The advantages of adopting an object-oriented approach in software system definition have been well documented elsewhere [e.g. 3, 12, 23]. The structure of a system developed using the object-oriented approach is different from that produced using a function-oriented approach. How then should we model our problem domain in terms of objects? and How are functional requirements which are part and parcel of an information system description be considered in the modelling activity? The next few sub-sections will elaborate on these two issues.

3.1. Characteristics of an Object Model

Actions

Since objects in the real world participate in a set of events, their actions in the events would indicate what can happen to them. For example, a Customer in a banking environment participates in events such as Deposit money, Withdraw money, Open an account and Close an account, the actions that are relevant to a customer in such a situation would be those actions relating to the opening and closing of accounts, and depositing and withdrawing of money from the account. Nevertheless, a Customer in an order-processing environment would participate in events[1] that are different from those in the banking environment, thus his actions would be completely different from customers in the banking application. In other words, the meaning of a customer, and hence objects in a problem domain, is defined by the actions that it performs or the events with which it undertakes. That is, object actions characterise the behaviour of the object.

Ordering of Actions

It is obvious that an object does not conduct its actions in the real world randomly. There is a pattern by which the sequence of actions follows. For example, when a user wishes to borrow some items from the library, he first has to register himself as a member

---

[1] The events are Order, Cancel, MakePayment, etc.

of the library. Only when he has become a member of the library, may he performs other actions like borrow a book, renew a book and return a book. He may also repeatedly renew other books which he has initially borrowed, etc. He continues to perform these actions until such time when he decides to terminate his membership with the library, during such time he executes a Terminate action which formally ends his membership. Thus, we see a pattern of actions for a library member i.e. Register, Borrow, Renew, Return, (and repeated Borrow, Renew and Return) and Terminate. This pattern of actions describes the life history or dynamic behaviour of the member. Modelling the dynamic behaviour of an object provides us with a more accurate representation of an object since events in the real world are constraint by a certain time ordering.

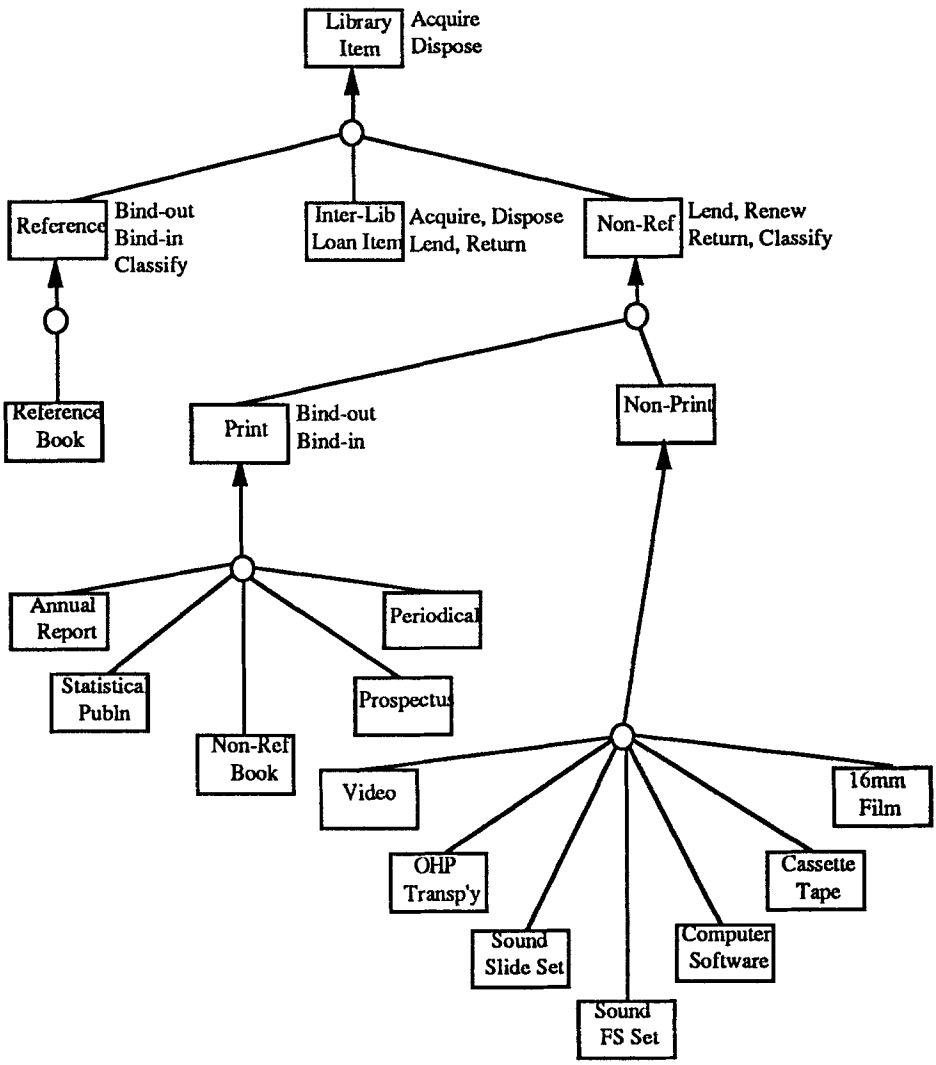


Figure 3.1 : Object Classification Diagram for Library Items

Attributes

In addition to actions, an object is also characterised by a set of properties known as attributes. The latter describes the state of an object. An example of attributes for a library member might be : Name, address, age, and loan-limit.

Classification

Objects in the real world are usually classified. For instance, books, periodicals, annual reports, prospectus and statistical publication may be classified as print items in a library environment; and these items may either be of reference or non-reference materials. A reference item may only be used in the library premises and unlike non-reference items may not be loaned out to any library users. The computational model representing similar informations of objects in the real world should thus include the concept of classification in its description.

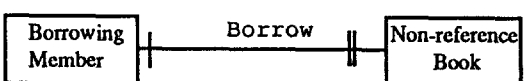
An object is classified by factoring common properties of similar objects into a class. The class then specifies the behaviour of all instances. Classes can be partitioned into subclasses, in which case the superclass is known as a *generalisation* [24] of the subclasses (and the subclasses are termed *specialisations* of the superclass). Figure 3.1 is an *Object Classification Diagram (OCD)* illustrating a classification hierarchy of library items in a library circulation application. Notice the action Classify has been defined in the respective subclasses; this is because Classify does not apply to Inter-Library Loan Item.

From the figure, we see that actions Acquire and Dispose have been repeated in Inter-Library Loan Item; the reason is due to different semantic requirements of these actions on the part of Inter-Library Loan Item from the others. For instance, we need to know the source library from which the inter-library loan item is acquired but this piece of information is not relevant to the other library items.

Relationships

Besides the *vertical relationship* of objects through classification, there is another form of relationship known as *horizontal relationship*. This kind of relationship describes the affiliation between two or more objects. Modelling relationships between objects provides us with an overview of the affiliation of objects with one another. For instance, a relationship between a borrowing member and a non-reference book is Borrow[1].

We can further impose upon a relationship a static restriction that dictates the cardinality of participation of object instances in the relationship. If the cardinality of the Borrow relationship between a borrowing member and non-reference book is one-to-many, it indicates that a member can borrow many non-reference books but only one non-reference book can be borrowed by one member. Expressing this form of relationship can be achieved with an *Object Relationship Diagram (ORD)* as shown in figure 3.2.



Rectangles represent objects; a line connecting the two objects indicates a relationship (in this example, Borrow). Double vertical bar denotes multiple relationship and single bar indicates single relationship.

Figure 3.2 : Relationship between Member and Non-reference Book (1:M)

---

[1] Borrow or Lend could be used depending on the direction in which the relationship is specified.

3.2. A Proposed Object Model

In summary, we can define objects as having five characteristics :

1. Objects have actions that describe their behaviour.
2. The actions of objects are time-ordered.
3. Objects have attributes that describe their states at a particular point in time.
4. The definition of objects can be organised into a class-instance relationship in a classification hierarchy.
5. Objects are affiliated to one another through relationships.

Figure 3.3 summarises this object model.

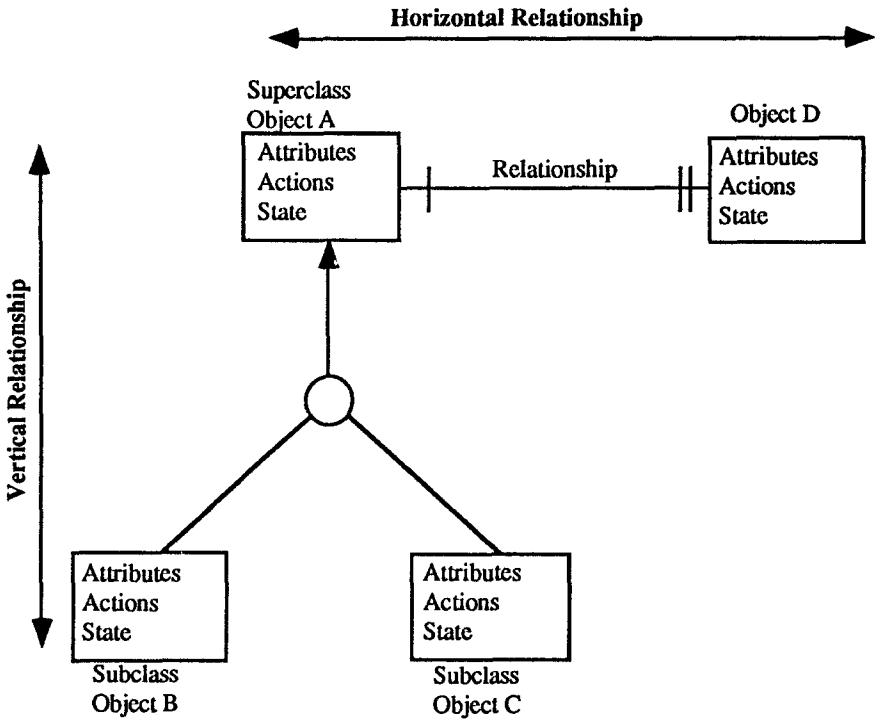


Figure 3.3 : Characteristics of Objects in an Object Model

4. Functional Requirement Modelling

Functional requirements are those input, output and processing requirements; they are part and parcel of any information systems description. They are specified in order to satisfy users' functional requirements. An example of a functional requirement in a library circulation application might be : *On request, lists all books currently on loan to staff member A.* This functional requirement is related to two objects : *Non-reference Book* and *Staff Member.* Contemporary object-oriented modelling approaches suggest the encapsulation of this functional requirement as an operation of a model object, i.e., define it as an operation of either Non-Reference Book or Staff Member. While this solution is plausible, it is not favourable for the following reasons:

1. It violates the essence of the definition of *object*. An operation defined for an object should be applicable to the object, the whole object and nothing but the object [3]. Since a functional requirement is a statement which is related to more than one object *(in this case, Non-Reference Book and Staff Member)*, we cannot categorically say that a functional requirement belongs to a particular object. Hence, functional requirements should not be defined as operations of objects in the object model.
2. It weakens the visibility of functional requirements, making changes to functional requirements difficult. For instance, in which of the two objects in the above example *(Non-Reference Book and Staff Member)* is the functional requirement represented, since representation of the requirement in both objects is possible.
3. Functional requirements are highly volatile elements. The principle of good software development practice [16, 17, 25] advocates the separation of volatile elements from the stable ones; this is to reduce the effect of change in one part of a system to another (possibly not related to the change at all).
4. To model functional requirements as operations of model objects has effectively forced us to consider the way a system is to be designed and implemented since there are many ways in which a requirement can be satisfied in the design of a system at a later stage. For instance, the functional requirement above can be satisfied in design by defining an operation in either Non-Reference Book or Staff Member, alternatively, we can define it as an operation of another third object that interact with Non-Reference and Staff Member to fulfill what is required.
5. This solution will lead to maintenance complexity in future changes to functional requirements. To illustrate, let us assume that the above functional requirement is encapsulated as an operation in one of the objects, say Non-Reference Book, and consider how a symmetric problem such as : *On request, lists all books currently on loan to staff members who have joined the library between 1 January 1990 and 1 January 1991* would affect the definition of the model. If this problem were to be satisfied as before, it would require a separate service module. The service module would have to interact with all instances of Staff Member to select the relevant instances (i.e. satisfying the constraint that he joined the library between the 1 January 1990 and 1 January 1991); and then looking up all instances of Non-Reference Book[1] and selecting those that are on loan to those relevant members. We note that a simple change in the functional requirement has resulted in a major change in the way the problem is resolved.
6. Finally, to insist on the definition of functional requirements as operations of model objects would go against the way a user would generally express functional requirements. They view functional requirements in a functional way and do not consider them as being part of a model object (i.e. it should not be considered as operation of either Non-Reference Book or Staff Member).

Hence, in conclusion, we state that it is necessary to separate the definitions of the object model from its functional requirements. If objects have been correctly modelled they are likely to change infrequently, if at all. Modification of objects should only be effected by major changes to the basic meaning of the system and generally reflects a change in the business environment.

---

[1] We may define an object keeping a record of the books that the member has borrowed; this is certainly much better than searching through all instances of books to derive the answer. But this is more of a design issue of describing how the solution could be achieved, and is not suitable in the analysis phase where we aim to understand what the solution is.

5. A Partial Framework for System Evolution

The discussion so far suggests that a system's framework consists of two layers as shown in figure 5.1. At the kernel of the diagram is the object model.

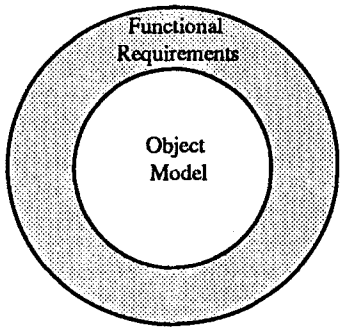


Figure 5.1 : Object Model and Functional Requirements

6. Business Policies

Consider the entries of Table 1. The latter contains a business policy pertaining to the loan quota for each class of member in a typical library environment (refer also to figure 3.1).

Member Class	Restrictions	Loan quota
Staff (Producer)	All items	4 print and 5 non-print items
Staff (Non-Producer)	All items except 16mm films	4 print and 5 non-print items
External Individual	All items except 16mm films	4 print and 5 non-print items
External Institutional	All items except 16mm films	10 print and 10 non-print items
Approved Training Centres	All items except 16mm films	10 print and 10 non-print items
Reader	Use library facilities only	0 item

Table 1 : Loan quota business policy

For instance, in Table 1, a staff member (Producer) may borrow all types of library items subject to a maximum of 4 print items and 5 non-print items. For Readers, they may only use the library facilities (such as using the library for reference work) but may not borrow any library items.

Consider the following policies : A borrowed item is said to be overdue when it is not returned to the library a day after the due date. Also, any overdue item is subjected to a fine of 50 cents per day per item. These policies read :

FineMember IF BorrowedItemIsOverdue.
BorrowedItemIsOverdue IF TotalFine > 0.

The total fine is governed by a computational policy indicated by the following formulae :

TotalFine = TotalDaysOverdue * 50 (cents)
TotalDaysOverdue = NumberOfDaysBorrowed - 1(day grace period) - TotalSundays - TotalPublicHolidays.

NumberOfDaysBorrowed = ReturnDate - LendDate[1].

These high-level declarative business policy statements are what a user would generally define in their context and are what they would view them as; unfortunately, they are usually not represented as they are. Instead, they are commonly transformed into low-level computational representation in its final specification. That is, the set of business policies, with which the definition of a system depends, is buried deep within the system program code, totally obscured from the users who define them in the first place and who are the ones that will request for changes in the future.

This mode of representation suffers from a number of problems. Firstly, programs become complex because the order of the procedural statements determines much of the logic of the program. Secondly, it is difficult to check the correctness of a program as few people with the knowledge of the policies will be able to understand the implementation. Finally, maintenance of programs is difficult, since programs describe a procedure of carrying out these policies rather than containing the policies themselves. Any changes to the business policy would require a re-ordering of the program logic and this could be costly [7]. In other words, procedural representation of business policies requires the pre-determination of the order of execution (of the program logic), whereas if they have been represented declaratively, then it is the environment that chooses the policy to be applied whenever the situation arises.

6.1. Taking a Synergistic Approach to Structuring a System

Based on the above, we can conclude that business policies of a system specification should be explicitly specified in software development and identifiably maintained throughout the development process and subsequent evolution, so as to permit immediate and flexible response to changes in users' requirements. Hence, in addition to separating object model from its functionalities, we also need to explicitly represent business policies such that the definition of the policies and their corresponding operational application are separated. Thus, in addition to the two layers, we have a third layer, the business policy layer as shown in figure 6.1.

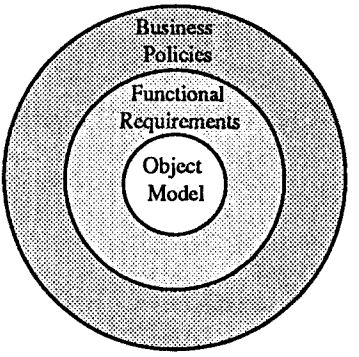


Figure 6.1 : The three layers of a system structure

[1] ReturnDate and LendDate are attributes of library items; they record the date upon which the item is returned and borrowed respectively.

This synergistic approach to structuring a system can potentially alleviate the deficiencies of present-day systems development methods, with particular emphasis on software maintenance.

7. Explicit Representation of Business Policies

To explicitly represent a business policy is to raise the level of representation of the business policy. As business policies are declarative statements about objects, their attributes and their actions, we thus expect them to be represented in the same manner. Let us now examine the areas in which business policies can be explicitly represented given the framework as illustrated in figure 6.1.

There are 4 areas in which explicit representation of business policies is applicable :
1. Object action
2. Object attribute
3. Condition derivatives
4. Computation derivatives.

7.1. Object Action

An action denotes a participation on the part of an object in an event. The execution of an action changes the state of an object. For instance, a Non-reference book issue is an event participated by two objects - Non-reference Book and Member. The action is a common action between the two objects (let's call the action 'Lend'), and when this event occurs, the Lend actions of the two objects are executed. The completion of the actions update the states of the two objects (i.e. the data values of the relevant attributes such as loanCount and currentBorrower will be updated).

7.1.1. Pre-Action Constraint and Post-Action Triggering Policies

However, before any actions can be executed, certain conditions may have to be satisfied. For example, before a book can be loaned to a member, it must first be available for loan; also, that the member has not exceeded his loan quota. For a book renewal, the book must not be initially reserved by another member, etc.

The above constraints are related to the point in time before the execution of an action. This is known as *pre-action conditional constraint*. There are also cases where upon the completion of an action, certain functionalities must follow. For instance, when a book which has been initially reserved by a member is returned, a notification card must be printed, to be sent to the reserving member. The action concerned is "Renew[1]", and the functionality is "Print a notification card".

In other words, for each action, there are pre-action condition and post-action triggering business policies associated with it. The pre-action condition business policies serve as a gate to the execution of an action; opening only when the conditions are satisfied and barring any execution if the situations are not consistent with the constraints. The post-action triggering business policies are links that connect actions to other functionalities which are regularly performed upon the completion of an event.

---

1 Renew is a common action of library item and member.

The pre-action condition business policies for the Lend action (Staff Member object) could be expressed as :

BEFORE Lend CONDITION PrintloanCount < PrintloanQuota.

For the action Renew, the pre-action condition business policy may be specified as :

BEFORE Renew CONDITION noPriorReservation

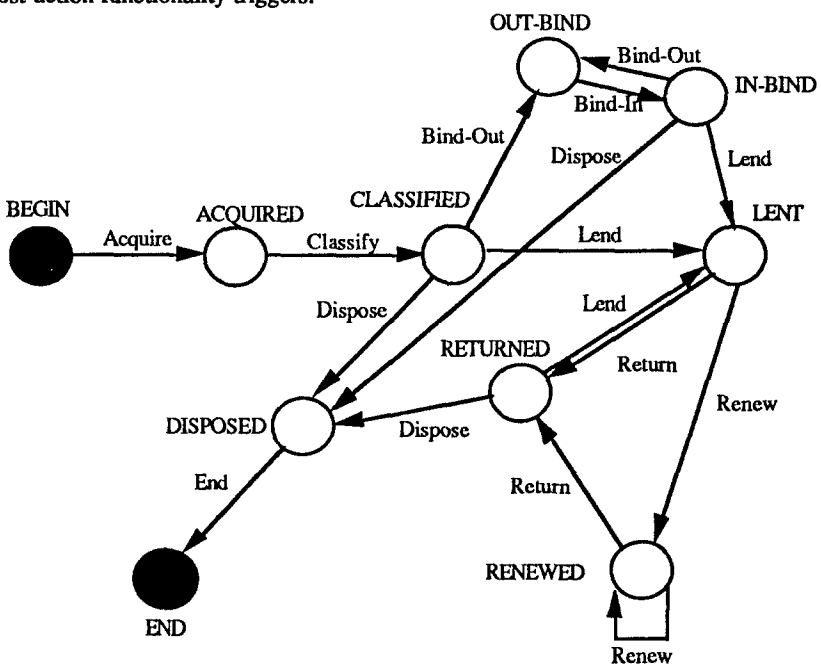
(where noPriorReservation is a functional module that returns a boolean value)

The post-action trigger business policy "When a book which has been initially reserved by a member is returned, print a notification card" can be expressed as follows :

AFTER Return AND ResvnCount > 0 CALL PrintNotificationCard(BookId).

This policy brings together the three related elements : Action (Return), attributes (ResvnCount) and functionality (PrintNotificationCard).

Explicit representation thus makes clear the policies relating to pre-action constraints and post-action functionality triggers.



A circle denotes a state. A darkened circle indicates the beginning and ending states of an object; they are special states applicable to all objects. A line connection indicates an action; the activation of the action moves the object from a state to another. For example, if the object is currently at CLASSIFIED state, it may accept actions Bind-Out (which will lead it to the OUT-BIND state) or Lend (which will lead it to the LENT state). The diagram also indicates the set of actions that the object may accept at a given state. In other words, the Non-Reference Book can only accept Bind-Out or Lend (and nothing else), if it is at the CLASSIFIED state.

Figure 7.1 : State-transitions of Non-Reference Book

7.1.2. Action Sequencing Policies

We saw in the earlier sections that an action is time-ordered. An object changes its state upon the completion of an action. The state transition is represented in the object by a

change in attributes' values. State-transitions can be depicted graphically using the standard state-transition diagram as in figure 7.1, or textually, in the following form :

FROM source-state WHEN action TO destination-state

For the object Non-reference Book, we may tabulate its state-transitions (or temporal constraints) as in Table 2. The FROM column consists of all source states. Given the set of respective actions in the WHEN column, the TO column defines the set of corresponding destination states. Since a state denotes the point in time when an action event has taken place, we use past-tense verb phrase to denote state. For example, the state for action Lend is LENT and for actions Renew, RENEWED, and Return, RETURNED etc.

Table 3 is another example but it is for *Reference* Book. The difference between tables 2 and 3 is that there are no state-transitions entries for the loan of book in the case of the Reference Book.

FROM	WHEN	TO
BEGIN	Acquire	ACQUIRED
ACQUIRED	Classify	CLASSIFIED
CLASSIFIED	Lend	LENT
CLASSIFIED	Bind-Out	OUT-BIND
CLASSIFIED	Dispose	DISPOSED
LENT	Renew	RENEWED
LENT	Return	RETURNED
RENEWED	Renew	RENEWED
RENEWED	Return	RETURNED
RETURNED	Lend	LENT
RETURNED	Dispose	DISPOSED
DISPOSED	End	END
OUT-BIND	Bind-In	IN-BIND
IN-BIND	Dispose	DISPOSED
IN-BIND	Lend	LENT

Table 2 : Temporal constraints for Non-reference Book

FROM	WHEN	TO
BEGIN	Acquire	ACQUIRED
ACQUIRED	Classify	CLASSIFIED
CLASSIFIED	Bind-Out	OUT-BIND
CLASSIFIED	Dispose	DISPOSED
DISPOSED	End	END
OUT-BIND	Bind-In	IN-BIND
IN-BIND	Dispose	DISPOSED

Table 3 : Temporal constraints for Reference Book

Action calls in the WHEN part of the expression correspond to action modules of objects. These modules are independent modules performing the tasks required of the objects in fulfilling the events. These tasks include the following kinds of operations :

1. attributes updates operations
2. value verification operations and
3. derivative operations where the derivation is based on other attributes.

Combining the state-transition expressions and action modules, we derive a situation as illustrated in figure 7.2 where action modules and their temporal constraints are loosely coupled and shared through the inheritance hierarchy as and where applicable[1]. Thus, the above suggests that temporal constraint policies *(business policy definition)* of actions and their corresponding action modules *(mechanism)* should be separately defined, so that changes in the temporal constraints[2] (state-transitions) are confined to changes in the set of state-transition expressions as tabled above. For instance, if for some reasons, the renewing policy for library items has changed as follows :

All items which are available for loans are renewable
except Non-Reference Books which must be returned when due.

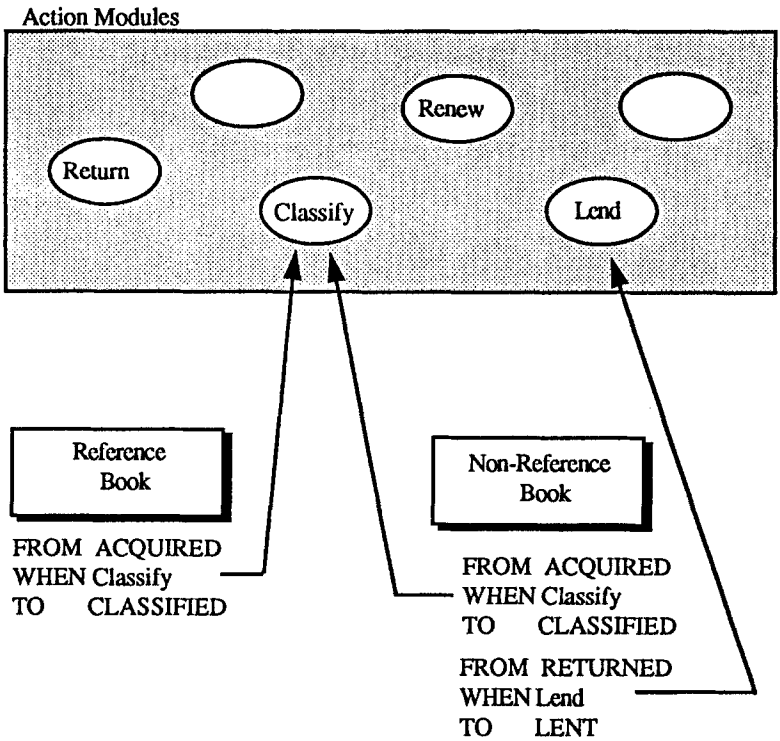


Figure 7.2 : Loosely coupling Action modules and their temporal constraints

This is a case of a change in the temporal constraints on actions and in this case, it applies to a Non-Reference Book. What is required to fulfill the change in the policy is to amend the set of state-transition expressions. In this instance, it entails the deletion of the followings from the set (see Table 2), all others remaining the same.

FROM	RENEWED	WHEN	Renew	TO	RENEWED
FROM	LENT	WHEN	Renew	TO	RENEWED

1 where an action is a specialised action then it has to be specifically defined in the specialisation class where it applies.

2 denoting a change in the life-history of an object.

Hence, explicitly representing the temporal constraints and separating their definitions from the action modules would thus provide a platform upon which changes in business policies could be made.

7.2. Object Attributes

An object has attributes and attributes affect one another in two ways. First, an attribute may only take a certain value depending on the object class. For instance, the loanQuota attribute of each member class is based on the following policies :

IF Producer	THEN	PrintLoanQuota = 4 and NonPrintLoanQuota = 5
IF NonProducer	THEN	PrintLoanQuota = 4 and NonPrintLoanQuota = 5
IF ExtIndivlMember	THEN	PrintLoanQuota = 4 and NonPrintLoanQuota = 5
IF ExtInstitnMember	THEN	PrintLoanQuota = 10 and NonPrintLoanQuota = 10
IF MemberATC	THEN	PrintLoanQuota = 10 and NonPrintLoanQuota = 10
IF Reader	THEN	PrintLoanQuota = 0 and NonPrintLoanQuota = 0

Second, the value of an attribute may be constrained by another attribute. For example, the attributes DisposedDate and ReturnDate of the Non-Reference Book are constrained in the following manner :

DisposedDate >= ReturnDate.

This means that a book cannot be disposed unless it has been returned earlier. Also, loan quota and loan count are similarly constrained in the following manner :

PrintLoanCount <= PrintLoanQuota
NonPrintLoanCount <= NonPrintLoanQuota

The above are constraint policies pertaining to object attributes; these policies do change. The meaning of these policies is usually encoded into low-level computational statements often obscured by the complexity of program code. Of course, constant parameters could be used; in which case, PrintLoanQuota and NonPrintLoanQuota could be stored in a parameter table and changed as and when the quota value for each of the member category changes. This solution is feasible but limited. For instance, the second kind of constraint which is also related to the first constraint cannot be represented using a parameter table. In fact, the second constraint has to be encoded within program logic, obscuring the definitions. Hence, to reduce the maintenance effort, we also need to consider the explicit representation of attribute value constraints as discussed above.

7.3. Condition Derivatives

Another area of concern is in the factoring of derivatives of condition policies. For example, the condition BorrowedItemIsOverdue was first expressed as :

BorrowedItemIsOverdue IF TotalFine > 0

Alternatively, we may express BorrowedItemIsOverdue as :

BorrowedItemIsOverdue IF (ReturnDate - DueDate) > 1

That is, if an item is returned more than a day after the due date then the item is considered as overdue. The digit "1" on the right-hand side of the expression indicates a one-day grace period for the member.

This policy (BorrowedItemIsOverdue) may be applied in various part of the system and possibly hidden in the complexity of the system, leading to difficulty in making changes to the policy when the need arises. However, if the definition of the policy is abstracted and

explicitly represented in a specification, then changes to the policy applies only at the policy definition rather than at *where* the policy occurs.

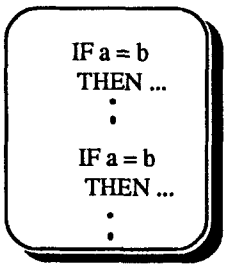


Fig 7.3

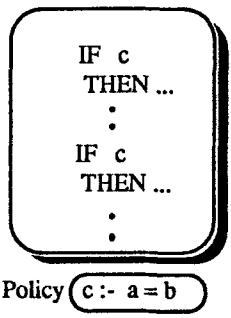


Fig 7.3b

Figure 7.3 illustrates the advantage of explicitly representing this type of condition derivative policies. In figure 7.3a, we have a policy (c IF a=b) which occurs at various parts of a system being represented as it is (a=b) in the system. Changing this policy would require the search for their occurrences and then making the required changes. However, in figure 7.3b, abstracting and explicitly representing the policy would confine the change only to the definition of the policy without affecting the entire system. Certainly, the latter approach is much better.

Explicit representation of condition derivative policies is particularly useful in system simulations where the change in conditions, representing different situations, could be facilitated easily. It is also useful for the monitoring of systems particularly when the system is implemented in a new environment. For instance, the above OVERDUE policy may include the one-day grace period for the initial two months of implementation but would have to be removed once the period is over. In this case, since the change is a policy change, then only the meaning of the policy should be changed from

BorrowedItemIsOverdue IF (ReturnDate - DueDate) > 1

to

BorrowedItemIsOverdue IF (ReturnDate - DueDate) > 0

and the others should remain the same as before.

7.4. Computational Derivatives

A condition derivative policy is related to another kind of policy known as *computational* policy. The term *computational* indicates that the policy has an arithmetical flavour. Indeed, the policy connects object attributes and other computational policies via an arithmetical formula. While the condition derivative policy suggests that a certain condition is true, it is the computational policy that stipulates how the value is to be calculated. For instance, we could tell if a Non-Reference Book is overdue via the condition derivative policy but we need a computational policy to indicate the total fine due. The formula for the calculation is stated in a computational policy as follows :

TotalFine = TotalDaysOverdue * 50.
TotalDaysOverdue = ReturnDate - DueDate - 1 - TotalSundays - TotalPublicHolidays.

ReturnDate and DueDate are attributes of the object Non-Reference Book; TotalSundays and TotalPublicHolidays could be pre-defined functions that return integer values relating to total Sundays and public holidays during the overdue period respectively.

The above computational policy not only makes clear the meaning of the formula but also brings together all related components associated with the computation. Since users generally express computational policies in such a declarative manner, representing them in a computer system in similar form would greatly enhance the maintenance effort.

Consider, for instance, the formula for the calculation of fine :
TotalFine = TotalDaysOverdue * 50.
TotalDaysOverdue = ReturnDate - DueDate - 1 - TotalSundays - TotalPublicHolidays.

These formulae may be used in various parts of a system. Hence, when there is a change in the definition of the formulae, there is a need to first locate where they have been used and change them accordingly. To alleviate this problem, one may package the calculation within a single module so that changes to the formula could be limited to the module. Although this solution is possible, it falls short of making the definition explicit. What this solution has achieved is to transform the meaning of the policy into a set of low-level program logic, albeit in a single module; often making the policies unrecognisable.

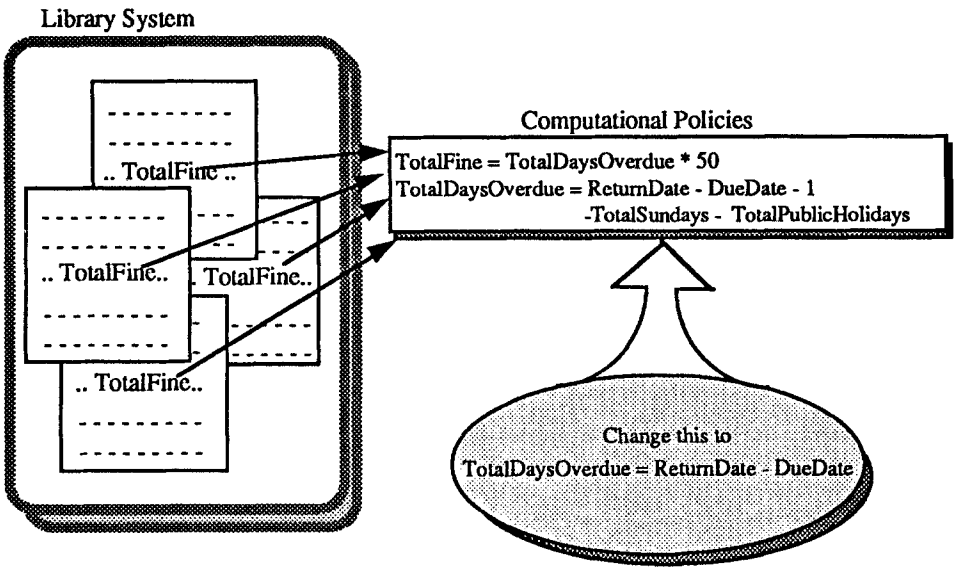


Figure 7.4 : Explicit Representation of Computational Policies

To illustrate, let's consider the previous computational policies on TotalFine. The TotalFine policy takes into consideration the fact that the library is not open on Sunday and Public holidays; that is why the calculation of the total fine amount does not include these days. Also, there is a one-day grace period given to the member for returning items that are due. Let's assume now that the library is open to members on Sundays[1], then the policy has to be amended to :

TotalDaysOverdue = ReturnDate - DueDate - 1 - TotalPublicHolidays.

Furthermore, grace period may be revoked and if such a situation occurs, the policy will have to be changed to :

TotalDaysOverdue = ReturnDate - DueDate - TotalPublicHolidays.

---

1 Highly possible in Singapore.

and if need be, the policy may be reduced to :

TotalDaysOverdue = ReturnDate - DueDate.

Even the rate of fine may be changed from 50 cents to say $1 to encourage members to return the items on time.

Although the changes may seem trivial, the efforts required to make the change may not be so using the conventional approach; and certainly the efforts would not be proportionate to the complexity of change if the policy is not represented explicitly as in figure 7.4.

Based on the above, we can conclude that there is a need to factor out the derivation of condition and computational policies from the environment in which they are used. In so doing, we aim to enhance the maintenance of the policies when the need arises.

8. Conclusion

This paper recognises the need to continuously provide for software system evolution throughout its entirety. It also suggests a framework upon which software system should be structured for easier maintenance. The framework is characterised by 3 elements : object model, functionalities and business policies. The paper concludes that these elements should be separated from one another and in particular the representation of business policies should be raised to a level where they can be explicitly recognised for changes when the business environment evolves.

References

1. vanAsschc F., Layzell P.J. and Anderson M. RUBRIC : A rule-based approach to the development of information systems, Proceedings of the 1st European Conference on Information Technology for Organisational Systems, Athens, 16-20 May 1988.
2. Bochm B.W., Software Engineering Economics, Prentice Hall Inc., 1981
3. Booch G. Object-oriented Development, IEEE Transactions on Software Engineering, Vol SE-2, No. 2, Feb 1986, pp 211-221.
4. Cameron J.R. An overview of JSD, IEEE transactions on Software Engineering Vol SE-12, No 2, Feb 1986 pp 222-240.
5. Cox, B.J., Object Oriented Programming: An Evolutionary Approach, Addison-Wesley Publishing Co., August 1986
6. Davis R. and Buchanan B. Production rules as a representation for a knowledge-based consultation, Artificial Intelligence 8, 1977 pp 15-45.
7. Fjeldstad R.K et al, Application program maintenance in [15], pp 13-27.
8. Gustafsson M.L., Karlsson T. and Bubenko J.A. A declarative approach to conceptual information modelling in [14].
9. Hayes-Roth F. Rule Based Systems, Communications of the Association of Computing Machinery, Vol 28, No. 9, Sept 1985, pp 921-932.
10. Jackson M.A. System Development, London : Prentice Hall International Inc., 1983.
11. Lientz B.P. & Swanson B. Problems in Application Software Maintenance, Comm. ACM, Vol 24, No. 11, pp 763-769, 1981.
12. Meyer B. Object-oriented Software Construction, Prentice-Hall, 1988.
13. Morris E.P. Strengths and Weaknesses in Current large scale data processing systems, Alvey/BCS SGES workshop, Jan 1985.

14. Olle T.W. et al (eds). CRIS1 - Information System Design Methodologies : A Comparative Review Amsterdam : North-Holland, 1982.
15. Parikh G. and Zvegintzov N. Tutorial on Software Engineering, IEEE, 1983.
16. Parnas D.L. On the criteria to be used in decomposing systems into modules Communications of Association of Computing Machinery, Vol 15 No 12, 1972 pp 1053-1058.
17. Parnas D.L. Designing software for ease of extension and contraction IEEE Transactions on Software Engineering SE-5, March 1979, pp 128-138.
18. Poo Chiang-Choon Danny. The integration of Rules into the Object-oriented Paradigm to facilitate Software Maintenance Ph D thesis, Dept of Computation, UMIST, Manchester, May 1988.
19. Poo Chiang-Choon Danny et al. Information Systems Development - A new direction, Proceedings of SEARCC 90, Dec 1990, Manila (Philippines).
20. Poo Chiang-Choon Danny. Adapting and using JSD modelling technique as a front-end to object-oriented systems development, Journal of Information and Software Technology, Vol.33, No. 7, Sept 1991, pp 466-476.
21. Poo Chiang-Choon Danny. An Object-oriented Software Requirements Analysis Method, (accepted for publication in International Journal on Software Engineering and Knowledge Engineering in June 1992).
22. Poo Chiang-Choon Danny. TarTan : An object-oriented System Modelling Method for MIS applications, Technical report, Dept of Information Systems and Computer Science, National University of Singapore (same as author's correpondence address).
23. Rumbaugh James et al. Object-oriented Modelling and Design, Prentice-Hall, 1991.
24. Smith J. and Smith D. Database Abstractions : Aggregation and Generalisation, ACM Transactions on Database Systems, Vol 2, No. 2, 1977, pp 105-133.
25 Yourdon E. and Constantine L., Structured Design : Fundamentals of a discipline of computer program and systems design, Yourdon Press, 1979.