JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 5, September-October 2003

Toward Better Logical Models in UML

P. V. Reddy, Independent Consultant, Bangalore, India

Abstract

In this paper we present that the logical models in UML can be made better in three ways. Firstly these can avoid the limited ways of using UML to overcome two deficiencies: i. the ambiguous interpretation of classes, and ii. a lack of one to one correspondence with the textual representations of the models.

Secondly the models can be hierarchical based on the notions of entity, collection, relation, and control classes. The collection, and relation classes are for overcoming the first two deficiencies mentioned above.

Thirdly relation driven design can be used as an alternative to behaviour driven design, which is clumsy and leads to spaghetti logical models. The latter views computation as interactions of objects and is very widely used in UML based design methods. Relation driven design is a novel contribution of this paper. On the contrary it is based on an alternate view of computation as interactions of relations rather than objects. Hierarchical logical models resulting from relation driven design are easier to understand, and maintain than the spaghetti ones resulting from behaviour driven design.

1 INTRODUCTION AND BACKGROUND

The present research is an outcome of our investigation to study the benefits from the representation of relations as classes as in [Soukup94, Soukup99] along with the RAISE method's strategy for hierarchical specifications [George92, George95]. These ideas were explored and tested in software industry on commercial projects. At very early stages of our investigation we found the benefits of such representations in RAISE's formal specifications in text. Later when we started using UML as a visual notation for the logical models, we found that there are two deficiencies in the use of UML: 1. ambiguous semantics of classes, and 2. a lack of one to one correspondence between UML based specifications, and the RAISE's textual specifications. We consider that the latter deficiency is a serious issue which blocks hierarchical thinking and also hierarchical logical models.

The main purpose of this paper is to illustrate by a simple example how logical models in UML can be improved in three ways. Firstly by avoiding the limited ways of using UML to overcome the above mentioned deficiencies. Secondly by making models

Cite this article as follows: P. V. Reddy: "Toward Better Logical Models in UML", in *Journal of Object Technology*, vol. 2, no. 5, September-October 2003, pp. 101-121. http://www.jot.fm/issues/issue_2003_09/article2

to be hierarchical based on the notions of entity, collection, relation, and control classes. Thirdly by using a new notion of relation driven design along with the hierarchical models for identifying operations. Relation driven design is a novel contribution of this paper.

To our surprise we have found that very few object oriented design methods [Booch93, Coad91, Cole94, Embley92, Jacob92, Rumb94, Reensk96, Shlaer89, Shlaer91, Wrifs90] in the literature discuss hierarchical logical models. As an exception, HOOD is a hierarchical design method for building systems using Ada [Heitz88] and is not actually object oriented, since it has no idea of classes or inheritance. HOOD consists of decomposing a parent object into several child objects which act together to provide the functionality of the parent.

The design methods [Booch93, Coad91, Cole94, Embley92, Rumb94, Reensk96, Schlaer89] visually model relation information. Though some of these methods [Embley92, Shlaer91] and the other authors such as [Champ93, Bock97a, Bock97b, Susch03] discuss relations as classes for analysis purposes, they do not exploit relation information for driving design to build hierarchical logical models. By and large all these methods include relation information in the states of objects, and use behavior driven design in one form or the other for identification of operations.

Behaviour driven design comes in two versions, one as in responsibility driven design [Wirfs90], and the other as in data driven design methods [Booch93, Cole94, Embley92, Reensk96, Rumb94, Shlaer91]. The method of the former version does not give importance to data modeling, whereas the methods of the latter version do. On the surface the methods of the second version seem to be different from the first version method. However all these methods primarily bury relation information in the states of objects, and go about identifying operations because these methods view computation as interactions of objects to change/assess their states. In doing so these methods do not explicitly exploit relation information for design purposes. Both the versions differ only in their ways of identifying operations.

UML assumes behavior driven design for identification of operations. To facilitate the design it provides interaction diagrams such as collaboration and sequence diagrams. Subsequently we will discuss how behavior driven design is clumsy and leads to spaghetti designs, and poor abstractions of operations.

To overcome the problems arising out of the three limited ways of using UML, we view computation to be interactions of relations. Such view is the basis of our relation driven design.

The organization of the paper is as follows. In Sec. 2, we provide an intuitive understanding of hierarchical logical models based on diagrams. In Secs. 3 and 4, we discuss the first two deficiencies due to a limited usage of UML for logical models. In Sec. 5, we also show how the latter deficiency goes hand in hand with the behavior driven design which makes logical models to be non hierarchical(rather spaghetti). In Sec. 6, we suggest our solution to overcome the first two deficiencies mentioned in using UML based on the notions of collection and relation classes, and additionally introduce the syntactic notions of entity and control classes to syntactically define hierarchical logical models. In Sec. 7, we introduce relation driven design to find operations in hierarchical logical models. In Sec. 8, we provide our experiences using relation driven design, and in Sec. 9, we conclude the main contributions of the paper, and the future work.

2 HIERARCHICAL LOGICAL MODELS

Logical model organizes all data and computation of system in terms of classes which are smaller units of both data and computation. Even then it is ideal to have a hierarchical view of both data and computation for easier understanding, development and maintenance of the model and its respective system. For this reason, we introduce the notion of hierarchical logical model.

For data in a logical model to be hierarchical, we shall be able to view the objects of its classes as boxes. Each such box can be recursively decomposed into smaller boxes, as shown below in Figure 1, for an instance. This means we have objects within objects.

In the diagram in Figure 1, boxes a and e are unrelated by their decomposition. Hence these are shown outside each other. Box a has boxes b and c at the immediate next level. This means that the object a has two objects b and c. Box b has box d. Once again boxes b and c are unrelated by their decomposition.

Similarly for computation of a logical model to be hierarchical, we shall be able to view the operations of its classes as boxes. For the purposes of viewing, we abstract from the details of the objects to which these operations belong to. Each such box can be recursively decomposed into smaller boxes, as shown in Figure 2. Note that an operation u can be composed from the operation v which may belong to other classes than u's. Similarly v in return can be composed from *x* and *w*.

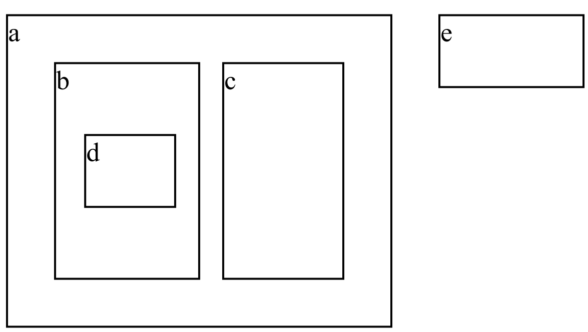

Figure 1. Hierarchical objects as boxes within boxes

A hierarchical view of operations allows their analysis to be based on a strategy of divide and conquer to handle their complexity. An operation is composed from less complex operations. But these need not belong to the object to which the parent operation belongs.

These may also belong to the objects which are inside the object of the parent operation. For promoting better abstractions of operations, it is usually recommended that a parent operation is composed from operations that belong to the objects at the immediate next level in the object of the parent operation if it is depending on the operations of inner objects.

The challenge is to enforce hierarchy of both data and computation into the logical models. When we do that we have hierarchical logical models. For this purpose we introduce two notions: i. logical models with hierarchical data and ii. classes with single focus.

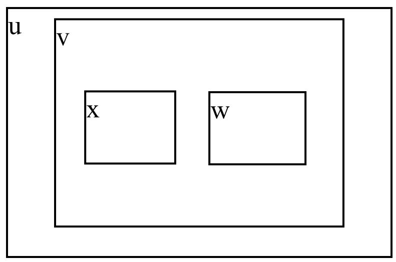

Figure 2. Hierarchical operations as boxes within boxes

Definition 1. A logical model has hierarchical data if the following conditions hold.

1. Each object of each class can be seen as a box.
2. Each box can be recursively decomposed into smaller boxes.
3. There are no boxes for the attributes of the basic types.

The second notion of class with single focus applies to the classes in logical models with hierarchical data. Intuitively class with single focus means that its operations can be understood solely in terms of its attributes and messages to them. This also implies that any of its operations can be also understood in terms of other operations in the class or those belonging to objects which are inside the object to which the operation belongs.

Definition 2. A class in logical model with hierarchical data has a single focus if the operations of its object can send messages only to itself or to its inner objects.

Definition 3. A logical model is hierarchical if it has hierarchical data, and each of its classes has a single focus. Otherwise it is spaghetti one.

For an explanation of Definition 2, once again we refer to the objects of Figure 1. Figure 3 visually shows possible messages between them. For instance a directed line from box a to b means there is a message from object a to b. All *acceptable* messages are in thick lines and are from an outer box to an inner box. The *unacceptable* ones are in dashed lines, and from a box to an outside box.

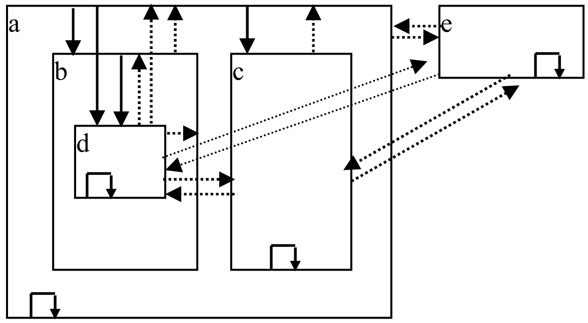

Figure 3. Message flows

Assume that the boxes *a*, *b*, and *c*, are respectively of classes A, B, and C. Hence the class A will have the following definition.

```
class A {
  b:B
  c:C

  // other attributes of basic types, and operations
  ...
}
```

Note that for instance the objects of the following classes in a logical model can not be represented as boxes within boxes.

```
class X {
  a: Y

  //other attributes and operations
  ...
}

class Y {
  b: X

  //other attributes and aoperations
  ...
}
```

Because of that the logical model is not hierarchical. Usually such classes arise because of burying relation information as part of their states. Moreover the classes can have multiple focus because of not observing constraints on message flows as given in

Definition 2. Because of such message flows, a logical model with hierarchical data, may not be hierarchical.

3 CLASS HAS AMBIGUOUS SEMANTICS

In this section we discuss that the classes in the logical models expressed in UML have ambiguous semantics. We consider a simple and familiar example of a college's library information system from where the students of the college borrow books. We have the following class model expressed in UML.

Figure 4. Logical model-Version 1

Let us assume that the problem domain has several students, and books. Then the class Student represents several students, and the class Book the several books. The above logical model does not explicitly mention whether it has indeed several students, and books. At this stage one might assume, whatever classes that are there in the logical model represent several objects in the problem domain. But such assumption cant generally hold. To counter the assumption, let us include an additional information: the library has a honorary librarian, who is not a student of the college. As a result, we have the following logical model.

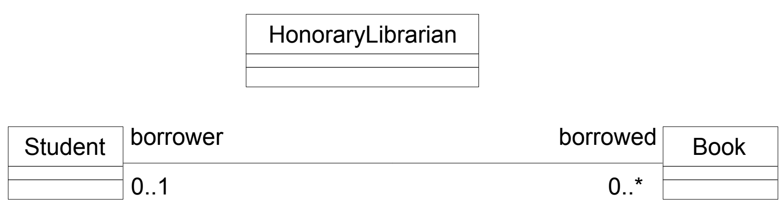

Figure 5. Logical model-Version 2

In the logical model in Figure 5, we notice that the new class represents only one object. This is in contrast to the other two classes which represent several objects.

Hence the question is: *When does a class in logical model expressed in UML represent* a single or several objects? In fact, it is not explicitly understood when a class represents one or more objects in the logical models expressed in UML. However it may be argued that one can establish the numbers of objects of each class actually occurring in the system/problem domain by studying the associations between classes in the logical model.

4 VISUAL AND TEXTUAL MODELS WITHOUT 1-1 CORRESPONDENCE

In this section we discuss that the logical models in UML lack 1-1 correspondence with their textual representations. Once again we consider the library information system for the discussion. We have the following textual representation for the above logical model in Figure 5. Note that we have not included the details of operations and their visibility details to keep the discussions simpler.

```
class Student {
  name:String
  idNo: Integer
  borrowed:Set<Book>

  //operations
  ...
}

class Book {
  title: String
  accNo: Integer
  borrower:Student

  //operations
  ...
}

class HonararyLinbrarian {
  name: String
  address: String

  //operations
  ...

}
```

In Figure 5, the relation information between Student, and Book is explicitly shown as an association. However there is no such explicit representation of relation information in the textual representation. We have buried the relation information pertaining to the association as attributes of the classes Student, and Book. Because of that the logical model has non-hierarchical data.

In the next section we discuss the disadvantages of behavior driven design method which is used by UML.

5 BEHAVIOUR DRIVEN DESIGN IS CLUMSY AND LEADS TO SPAGHETTI MODELS

Burying of relation information has been the basis of *behavior driven design* for identification of operations. Incidentally it has been a part in one form or the other of almost every design method that has been published in the literature [Booch93, Coad91, Jacob92, Shlaer91[1], Odel98, Rumb94, Reensk96, Wrifs90]. In fact UML's accommodation of the design method by providing collaboration and sequence diagrams indicates its wide spread popularity.

The behavior driven design is based on a very commonly held view of computation. As per the view a computation is nothing but interactions of objects by sending messages from one to another. The purpose of the messages is to change/assess the states of objects.

The behavior driven design has the following features:

1. It makes no assumptions on the data of the classes in logical models. Hence the question of the models with or without hierarchical data does not arise.
2. All objects are peers to each other. Hence it allows any object to send message to any other objects.
3. It may identify operations without a good level of abstractions, and details.

Because of the first two reasons, the behavior driven design results in contributing spaghetti logical models. Such models may not have hierarchical data as we defined in Sec. 2. Secondly such models have classes without a single focus because of their operations. For these reasons, the classes in such logical models are also clumsy. Hence we claim that the behavior design is clumsy. In the rest of the section we explain the above features.

Non-hierarchical data

Behaviour driven design comes in two forms – i. responsibility driven and ii. data driven. Data driven design methods have an expectation that an object shall have a reference to another for the former to send a message to the latter. The expectation of references alone does not pose any hierarchy on data. Responsibility driven design does not have such expectations. Hence we can say that behavior driven design does not assume hierarchical data.

[1] Though Shlaer & Mellor do not have responsibility driven design they still use the view of computation held by the behaviour driven design.

All objects are peer to each other

Consider once again the same library example and the use case, issuing a book to a student. Assume that the operation is possible only if the book is not issued to anyone. We show below a collaboration diagram based on the logical model in Figure 6.

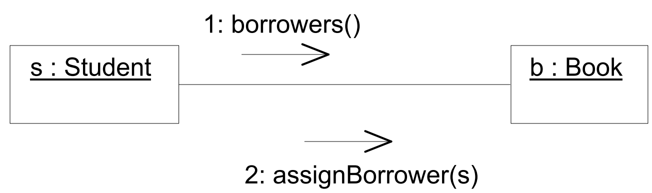

Figure 6. Collaboration Diagram

The sequence diagram in Figure 7 shows the sequential flow of messages for realising the use case.

These objects can send each other messages since these are assumed to be peers.

Note that a student object can not have a book object inside it, and the vice versa. As a consequence the logical model in Figure 5 has non hierarchical data. Hence the question of restricting messages on the basis of their hierarchy does not arise.

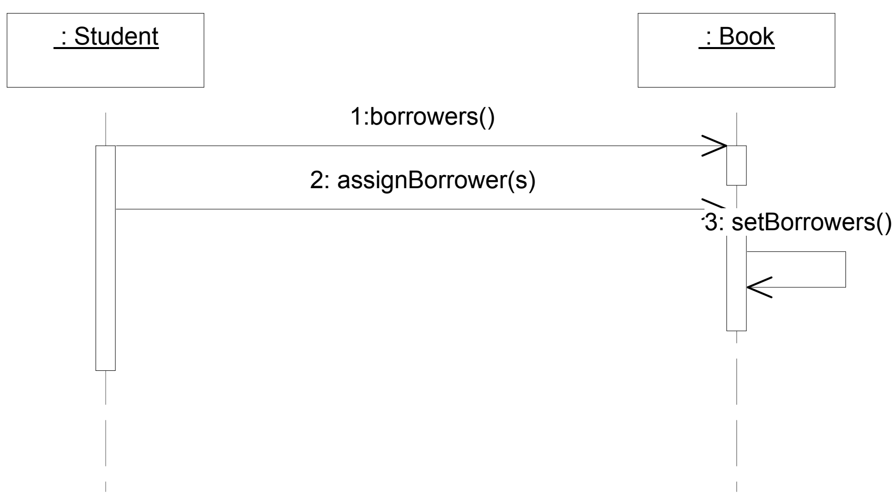

Figure 7. Sequence diagram

Lacks/promotes inadequate levels of abstractions

Because of its simple view on computation the behavior driven design is in terms of identifying behaviours without any botheration about data in the objects. Hence it can identify behaviours which can be with inadequate abstractions. Also the visual tools such as interaction diagrams for helping the design cant fully capture all the details. Hence when requirements change, managing the design to reflect the new requirements based on such interaction diagrams is often painful. For the appreciation of these claims, we continue below our discussion with the same use case of issuing book to a student with a few more additional details.

We further assume that the issue of book to a student also depends on reservations. In such a case there may be some students who have reserved for the book. So for a student s to be issued the book b, the conditions that should be obeyed are:

1. the book b is not issued to any one,
2. if there exist reservations for the book, the student s is the first one.

To take into consideration the additional requirements, we show a new logical model in Figure 8. It is a result of adding a new association for reservations between the classes Student, and Book of the logical model in Figure 5.

We show a sequence diagram for the extended use case in Figure 9.

We can see that the sequence diagram models the flow of messages. However it does not include business logic pertaining to the use case. If we put all the finer details into the diagram it becomes clumsy. Hence sequence diagrams may not be good for capturing all the details of business logic. For the idea of the details not covered by the sequence diagram in Figure 9, please study the textual specification for the operation pertaining to the use case in Sec. 7. In practice designers do not include all such missing details in the sequence diagrams. Hence they avoid changing them for incorporating changes in requirements. Where designers have time for incorporating changes, the resultant sequence diagrams often miss many details. If changes to requirements are often designers have a tendency to give up changing the sequence diagrams.

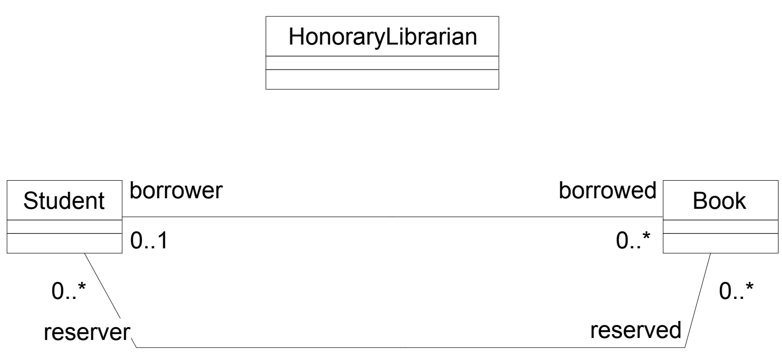

Figure 8. Logical model of Figure-7 additionally with reservations

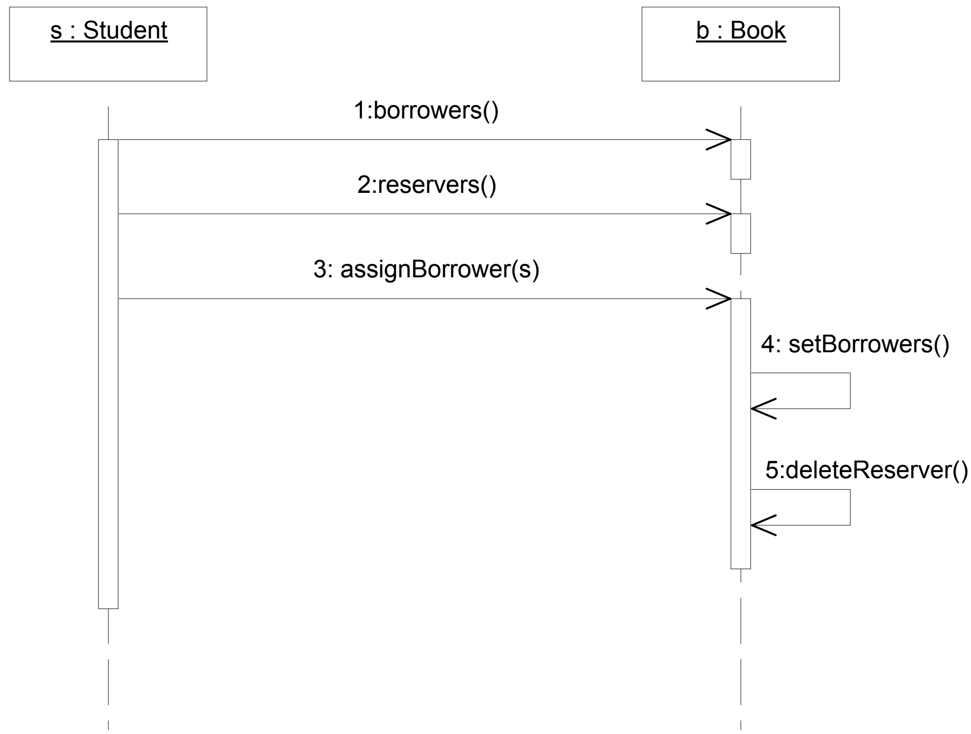

Figure 9. Sequence diagram for the extended use case

When design covers several use cases we will find objects sending messages back and forth because these do not have hierarchy. We may also find objects sending messages back and forth even in design for a single use case. This means that the logical models are inherently assumed to be spaghetti. In such a case hierarchical/stage-wise understanding, development, testing, and maintenance of classes do not arise.

6 OUR SOLUTION

To avoid the first deficiency of ambiguous semantics of classes in logical models specified in UML, we use collection classes which are defined using generic classes Set, Sequence, and Bag. To avoid the second deficiency of a lack of one to one correspondence between logical model in UML and its corresponding textual specification, we use relation classes which are defined using generic classes [Reddy2001]. Besides our solution uses the syntactic notions of entity, and control classes for hierarchically organizing the data in logical models.

Jacobson introduced the notions of entity and control classes [Jacob92]. His entity classes include relation information and it is not so in our case. Moreover, his control classes do not have relation objects as attributes. Champeaux introduced collection, relation and coordinator classes [Champ93]. Coordinator classes maintain collections, and relations as our control classes. However the distinction between his and our approach is in we forcing the entity classes to have attributes of basic types only, which is not the case with Champeaux. Secondly both Jacobson, and Champeaux do not have the notion of hierarchical logical models in which relations play a crucial role in identification of operations.

Relation classes

Table 1 summarizes generic classes for defining the relation classes based on generic classes, where Source, Target and Assoc are class parameters. The first four generic classes are for binary relations without associations, whereas the remaining ones are for binary relations with associations.

Relation name	Generic type
One to One	ONE2ONE<Source,Target>
One to Many	ONE2MANY<Source,Target>
Many to One	MANY2ONE<Source,Target>
Many to Many	MANY2MANY<Source,Target>
One to Many with association	ONE2MANY_A<Source,Assoc,Target>
Many to One with association	MANY2ONE_A<Source,Assoc,Target>
One to Many with association	ONE2MANY_A<Source,Assoc,Target>
Many to Many with association	MANY2MANY_A<Source,Assoc,Target>

Table 1. Generic types for binary relations

An explanation of the generic classes is as follows. For instance, the class ONE2MANY<Source, Target> defines relation class for associating an object of Source with 0, or more objects of Target. And conversely an object of Target with 0 or 1 object of Source. An object of the relation class maintains a set of ordered pairs of Source and Target objects. The class has typical add and delete methods for maintaining the relation objects, and the following query operations on the relation object.

forward(Source): Set<Target>	Outputs all targets, which are related to a Source
backward(Target): Set<Source>	Outputs all sources, which are related to a Target

Similar explanation applies to all generic relation classes with an association with an additional note that each linking of an object of Source with an object of Target is associated with an object of Assoc.

Relations based logical models

Relations based logical models have hierarchically organized data in the following way. They have entity classes whose attributes are only of basic data types. Typically collection and relation classes are defined using generic classes. Control classes have attributes which belong to already defined entity, relation, collection, and control classes. Two control classes are not allowed to be cyclically defined in terms of each other.

In these models the hierarchy of classes is as follows. Entity classes are at the lowest/first level. In the second level, all the collections, and relations based on the entity classes. In the third level, control classes. And so on.

We have ignored the details of sub typing of entity and control classes in the description of the hierarchy. More formal definition of the classes that occur in a hierarchical logical model can be specified using a recursive definition as in [Reddy2001]. Based on such definition we can also precisely define the level of a class in a model.

Note that it is not necessary to use relation classes to hierarchically organize data. Hence the specialty of our hierarchical logical models is in the use of relation classes for an explicit and syntactic representation of relation information. Studies on hierarchical logical models are hardly any in OO literature. Besides such logical models based on relations do not exist in the literature. Hence the comparison of the practical benefits of the hierarchical logical models with or without relations is beyond the scope of this paper, and shall be addressed by future research.

For the example in our discussion we present below a relations based logical model with hierarchical data.

```
//entity classes: Student, Book, HonararyLibrarian
class Student {
  name:String
  idNo: Integer

  //operations
  setName(String):void
  setIdNo(Integer):void

  getName():String
  getIdNo():Integer

  isName(String):Boolean
  isIdNo(Integer):Boolean
}

class Book {
  title: String
  accNo: Integer
```

```
    //operations
    ...
  }

  class HonararyLinbrarian {
    name: String
    address: String

    //operations
    ...
  }

  //collection classes: Students, Books
  Students = Set<Student>
  Books = Set<Books>

  //relation classes: Issues, Reservations
  Issues = ONEMANY<Student,Book>
  Reservations = MANY2MANY_S<Student,Book>

  //control class
  class Library {
    students: Students
    books: Books
    hLibrarian: HonararyLibrarian
    issues: Issues
    reservations: Reservations

    //operations
    ...

  }
```

Note that the class Student can have a single focus because its operations need only to manage its state, and not the state of any other class. This observation also applies to the class Book. Additionally we have relation classes Issues, and Reservations. These classes are by default with single focus, and explicitly represent the relation information between students and books. Their instances as attributes in the control class Library represent binary relations. These relations are respectively sets of ordered pairs of Student and Book objects.

Relations based logical models can be easily specified in UML. Relation classes are expressed by extending UML's association by using a stereo type relation.

Figure 10 gives a UML representation of the hierarchical logical model given in the textual specification above. It can be easily noted that the logical model in UML representation does not have the two deficiencies discussed earlier.

One may wonder whether it is computationally efficient to have relation classes implemented as it is. Our answer is that we shall defer from addressing such questions/issues while building logical models, since these models are expected only to be abstractions of the system/code and also the problem domain. Their transformation to the code/system shall be a separate issue which includes how these needs to be implemented.

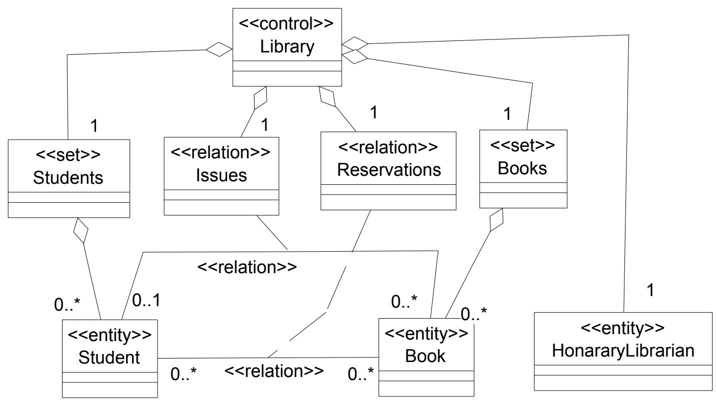

Figure 10. Hierarchical logical model for the library information system

7 RELATION DRIVEN DESIGN

As we have discussed earlier behavior driven design can be messy, and without good levels of abstractions of their operations. Also it leads to spaghetti logical models. To avoid these problems we introduce *relation driven design* for identification of operations in relations based logical models with hierarchical data. The design assumes syntactical notions of entity, collection, relation, and control classes as discussed earlier.

To define meaningful objects that correspond to the objects of the problem domain, each object needs to be defined on the basis of its relations with other objects and/or values of basic data types. It shall be noted that the attributes of basic data types also represent relation information. We classify it to be simple. Moreover relation objects capture binary relation information between objects, which do not belong to basic types. We classify such information to be complex.

We decide what relation information shall go into what classes by syntactically restricting entity classes to have attributes of only basic data types. Hence any relation information that is not covered by entities is covered by relation classes.

Relation driven design is based on an alternative view of computation as interactions of relations. As per the view complex relation information is maintained in relations, and separate from the attributes of objects. Hence computation is refined to be changing/assessing the states of objects, and/or their relations. Embley has taken this refined view of computation in his object oriented analysis [Embley92]. However he has not exploited relations for design purposes and also for building hierarchical logical models.

Description of the method

The relation driven design has inputs of use cases with goals [Coburn95, Coburn00]. Such goals are for actor fulfillment and by achieving certain states of objects; by forging or breaking relations between objects.

Relation driven design identifies operations belonging to classes in logical models in two different ways from behavior driven design. Firstly it assumes logical models to have hierarchical data based on the notions of entity, collection, relation, and control classes. Secondly it forces the operations such that the classes in logical models have a single focus. That is operations in an object can send messages to it or to the objects inside it.

As a result operations can be recursively viewed in terms of less complex operations and smaller objects. This is not the case with the behavior driven design. Relation driven design always results in hierarchical logical models.

To find operations pertaining to a use case we first identify entity and control classes of the logical model in question. For the entity classes we introduce the operations to realize the use case. Note that these operations manipulate the attributes of their classes, and nothing else. Thus these classes have a single focus.

For the control classes, we introduce one or more operations. So for each operation we identify the relevant entity, collection, relation, and control attributes in its control class, and specify it in terms of pre and post conditions based on them. Typically lower level control classes have operations manipulating the states of entities, and their relations. It shall be noted that the exercise of identifying the relation attributes is equivalent to finding collaborations. Writing pre and post conditions in terms of relations replaces the need for sequence diagrams. Hence we may not need such diagrams. In fact we have never felt the need for such diagrams to build the logical models. We ensure that the added operations keep the focus of their classes to be single. That is they do not change or access the state of objects which are not inside the object to which the operations belong to.

For the library example, the relations based logical model with hierarchical data, Library is the single control class. For the use case of issuing book to student, we add an operation issueBookToStudent(Student,Book) to the control class. We identify that the attributes of the control class that are related to the operation is the collection objects students, and books; relation objects issues, and reservations. We specify the operation below in terms of pre and post conditions using these collection and relation objects.

```
issueBookToStudent(Student s, Book b)
pre:
1.  s is in students, b is in books
2.  b is not issued to anyone
3.  there are no reservations for b or
    if there are reservations for b, e is
    the first one
```

```
post:
1. old(students) and students are identical
2. old(books) and books are identical
3. s has been issued the book b
4. there is no reservation of s for  b
```

In the post condition, old(students) refers to the value of students attribute before the operation is performed. Similarly old(books) represents the value of books before the operation is performed.

We can easily observe that the operation is specified on the basis of relations, and collections. This style of specification is declarative and purely on the basis of natural language descriptions. Such specifications can be also expressed in set theoretic notation mixed with the natural language. We show below the above specification in terms of set theoretic notation mixed with English. We have found that software engineers are very comfortable in adopting such notation. We have cautiously avoided using formal logic in the first place since many software engineers are unfamiliar with it.

```
issueBookToStudent(Student s, Book b)
pre:
1.  s ∈ students, b ∈ books
2.  issues.backward(b) = ∅
3.  reservations.backward(b) = [] or
    if reservations.backward(b) ≠ [], e is
    the first one in reservations.backward(b)
post:
1. old(students) = students
2. old(books) = books
3. b ∈ issues.forward(s)
4. reservations.backward(b) excludes s
```

It is interesting for instance to compare second condition in the pre condition of the above versions. The condition *b is not issued to any one* is expressed as *issues.backward(b)* = ∅. The latter expression is based on relation object. One might say the same could have been achieved even when relation information is pushed into b. No doubt that is possible. However it can be easily mapped to expressions in terms of relation objects.

Relation classes are useful to represent relation information explicitly represented in the model. Because of that the advantage will be to consciously figure out the specifications of operations and their changes in terms of relations as well.

The pre and post conditions of the above specification are explicitly based on conditions on relations. Such conditions reflect interacting relations. For some operations we may not have interacting relations. Such operations represent simple computations. Hence in general we view computation as interactions of relations.

At this stage it shall be noted that the identification of many operations pertaining to control classes are inspired by (user) goals which are understood in terms of relations and/or states of objects. Such operations often involve interacting relations, where the interactions are better expressed in natural language and more formally in set theoretic and/or formal logic notations. Sequence diagrams are not useful to express these interactions as these diagrams are meant for representing object interactions. As computation is viewed as interactions of relations sequence diagrams become less relevant to represent computation. In the case of computation involving more than one operation use case maps [Buhr96] and activity diagrams seem to be better alternatives than sequence diagrams to represent computation . All such operations are organized in terms of smaller goals thus allowing top down approach to design.

Specifications based on relations as discussed can be also specified in Object Constraint Language(OCL) which is OMG's standard for expressing the pre and post conditions. For instance the above specification in OCL is as given below.

```
issueBookToStudent(Student s, Book b)
pre:
   students->exists( x | x.idNo = e.idNo) and
   books->exists(x |x.idNo = b.idNo)  and
   issues.backward(b)= Set{} and
   reservations.backward(b) = Sequence{} or
   (reservations.backward(b)<> Sequence{} implies
   reservations.backward(b).first() = e)
post:
   students@pre = students and
   books@pre= books and
   issues =issues@pre->including(e,b) and
   reservations.backward(b)->excludes(e)
```

In summary we can easily notice that entity, collection, relation and control classes can be used to hierarchically organize data in logical models. Relation driven design can be used for identification of operations in such logical models. As a result the classes in those models will have a single focus and there by making the models to be hierarchical.

The advantage of using relation driven design is that the changes in requirements can be easily reasoned on the basis of relations. Because of that modifying the specifications is easier than in the case of operations identified on the basis of behavior driven design. Illustrating this point is out of the scope of this paper.

8 EXPERIENCES USING RELATION DRIVEN DESIGN

The author has been using relation driven design in the software development projects for the last four years in the industry. There have been more than 35 software engineers who used it under his supervision, and guidance. There is a positive feedback from the users of the approach, which is as follows: i. helps organization of analysis because of hierarchical models, ii. provides convenience to handle more details because of rigor, iii. provides easier understanding, coding, testing and maintenance of system iv. exploits the possibilities for code reuse because of generic classes, v. hierarchical logical models are good for the definition of work, vi. hierarchical structure of design is useful for allocation of work in a systematic way.

We have found that the first challenge in producing a hierarchical logical model is in getting the logical models with hierarchical data. Once that is achieved, introducing the operations in the classes based on relation driven design is an easier task without any need for the interaction diagrams. In fact freeing ourselves from the such diagrams on the basis of relation driven design has led to conservatively 3 – 5 times productivity in producing the designs and in the software life cycles. The productivity measures were in terms of man months, and not based on any other metrics. Hence we feel that the future research/experiences shall collaborate how cognitively relation driven design is better than behavior driven design.

There is a small percentage of people (15-20%), who are unsure about the treatment of relations. This may be due to a lack of abstraction skills.

We had opportunities to bench mark the benefits of hierarchical thinking, along with relation driven design. The bench marking is informal and is based on effort. It is given for three cases:

1. A 60 man month internet project on document review system was redone using relations based hierarchical logical models in 12 man months effort.
2. 15 man month design exercise on an interactive distance learning project was redone in 1 man month effort.
3. There is a 27 man month project on element management system. Similar project with same size, and complexity was completed in 9 man months effort.

9 CONCLUSIONS

After providing a brief view of hierarchical logical models, we have shown two deficiencies of logical models expressed in standard UML. The first one is the ambiguity in the semantics of classes and the second is a lack of one to one correspondence between a logical model in UML, and its corresponding textual representation. The latter deficiency arises because of burying relation information in the states of objects. We have also shown how the latter deficiency is linked with behavior driven design which many design methods in the literature have adopted. We have introduced collection, and relation classes to overcome these deficiencies. We have used a visual representation for relation classes in UML using a stereo type. Such representation has 1-1 correspondence to relation classes in textual specifications. We have also introduced entity, and control classes to build logical models with hierarchical data, which can be also expressed in UML.

We have introduced relation driven design for identifying operations in logical models hierarchical data. The process is based on an alternative view of computation as interactions of relations. We have also shown that the process leads to specifications of operations in terms of natural language. Such specifications can be easily expressed in set theoretic notation using syntactic terms for relations.

10 ACKNOWLEDGEMENTS

The author expresses his thanks to Professor Dines Bjorner, Technical University of Denmark, and Chris George, UNU/IIST for introducing him to the RAISE method; and Dr. Jiri Soukup, Codefarms Inc for introducing him to his data object library (DOL) which inlcudes generic relation types.

REFERENCES

[Bock97a] Bock, C., and Odell, J. "A More Complete Model of Relations and Their Implementation, Part 1", in *Journal Of Object-Oriented Programming,* Vol. 10, No. 3. June 1997, SIG Publications, NY

[Bock97b] Bock, C., and Odell, J. "A More Complete Model of Relations and Their Implementation, Part 2", in *Journal of Object-Oriented Programming,* Vol. 10, No. 6. October 1997, SIG Publications, NY

[Booch93] Booch, G. *Object-Oriented Analysis and design with Applications*. Benjamin/Cummins, Redwood City, Calif. 1993.

[Buhr96] Buhr, R.J.A., and Casselman, R.S. *Use Case Maps for Object Oriented Systems*. Prentice Hall, 1996.

[Chen76] Chen, P. "The Entity-Relation Model – Toward a Unified View of Data". *ACM Transactions on Database System*, 1(1):9-36, March 1976.

[Coad91] Coad, P., and Yourdon, E. *OOA - Object-oriented Analysis.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Coburn95] Cockburn, A. "Structuring use cases with goals" in *Journal of Object-Oriented Programming*, 1995.

[Coburn00] Cockburn, A. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[Cole94] Coleman, D., et al. *Object-Oriented Development – The Fusion Method*. Prentice Hall, Englewood Cliffs, NJ, 1994

[Champ93] de Champeaux D., et al. *Object Oriented Software Engineering*, Addison-Wesley, 1993.

[Embley92] Embley, D.W., et al., *Object-Oriented Systems Analysis: A Model-Driven Approach*, Yourdon Press, 1992.

[George92] George, C., et al. *The RAISE Specification Language*, Prentice Hall, 1992.

[George95] George, C., et al. *RAISE method*. Prentice Hall, 1995.

[Hietz88] Hietz, M. "A Hierarchical Object-Oriented Design Method". *Proc. 3rd German Ada Users Congress*, Munich 1988.

[Jacob92] Jacobson, I., et al. *Object Oriented Software Engineering - A use case driven approach*. Addison-Wesley, 1992.

[Odel98] Odel, J. *Advanced Object Oriented Analysis & Design Using UML*. Cambridge University Press, 1998.

[Reddy01] Reddy, P.V. "Structured Object Oriented Designs" in *Proceeding of NCOOT-2001*, Allied Publishers, Bangalore, India.

[Reensk96] Reenskaug, T. *Working with Objects: the OOram Software Engineering Method*, Manning Publications, Greenwich, England, 1996.

[Rumb94] Rumbaugh, J., et al. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs. N.J, U.S.A,1991.

[Shlaer88] Shlaer, S., and Mellor, S. *Object Lifecycles: Modeling the World in Data*, Prentice Hall, 1988.

[Shlaer91] Shlaer, S., and Mellor, S. *Object Lifecycles: Modeling the World in States*, Yourdon Press, 1991.

[Soukup94] Soukup, J. *Taming C++, Pattern Classes for Large Projects*. Addison-Wesley, 1994.

[Soukup99] Soukup, J. "Data structures as objects- Designing for 10 times efficiency" in *Dr. Dob's Journal*, October, 1999.

[Susch03] Suschek, C. A., and Sanden, B. "A Construct for Effectively Implementing Semantic Associations", in *Journal of Object Technology*, vol. 2, no. 3, May-June 2003, http://www.jot.fm/issues/issue_2003_05/article1

[Wirfs90] Wirfs-Brock, R., et al. *Designing Object-oriented software*. Prentice Hall, N.J, U.S.A, 1990.

About the author

Dr. P. V. Reddy is an independent consultant. He holds a PhD from Imperial College, London. He was a visiting scientist at the Indian Institute of Science. He earlier worked in Bangalore at Infosys Technologies, Hughes Software Systems, and Xalted Networks. His research interests include formal software specification methods and OO design methods. He can be reached at objecttrees@vsnl.net.