Rapid Prototyping with the OR Model

Yoav Intrator
and
Shmuel Rotenstreich

Dept. of EE&CS
The George Washington University
Washington, DC 20052
yoav@seas.gwu.edu

Abstract

An Object-Oriented model called the OR model is presented and shown to be powerful for rapid prototyping of Object-Oriented systems. The OR model is based on relations that replace the implicit interconnections that typically exist in every software system. These relations are defined in class hierarchies and can be used similarly to the way regular objects are used, e.g., inheritance and reuse. An example of a nontrivial data structure, the heap, is fully presented in the paper. The reader can compare our implementation with any other typical implementation of a heap and identify the advantages of our approach.

1 Introduction

Rapid prototyping has been the center of research, development and practice for quite some time. A special issue of IEEE computer magazine[1] show-cases the variety of disciplines using prototyping. In particular, software prototyping[6] is an example for a generic software approach. Software prototyping can also be attempted in an artificial intelligence framework[5]. Real-Time prototyping was undertaken also[7]. Of particular interest to us is object-oriented approach to prototyping[2][3].

Object-Oriented Programming (OOP) intends to increase modularity by defining software in terms of objects that are closely related to a view of the world. Commonly, relations among objects are defined in the objects themselves. For instance, with one object representing a company and another object representing an employer in that company, each one of these two objects must refer to the other object. This implies that objects are dependent on each other. When rapid prototyping of a software system in OOP is considered, the interrelation between objects has to be expressed by related objects. As a result, the use of class libraries for rapid prototyping must be associate with adaptation of related objects. Given the company-employer objects, if a company class exists, a person object has to be adapted to allow the relation with the company.

The Object-Relation (OR) model suggest a view of the world as collections of objects and relations among them. The model facilitates the expression of the relations among objects as a new type of objects, called relational objects. Analogous to the notion of a class and instances in OOP, we define relational classes and instances of relations. Thus, in an OR implementation of the company-employer objects, the employer will be an object of class person but there will also be a relation such as “works-for.” Similar to the object class hierarchy, the OR model implies a relation class hierarchy. The use of such hierarchy is similar to the use of an object class hierarchy.

Rapid prototyping may benefit from several advantages by using objects and relational objects conjointly.

- New or different relations can be expressed among objects without changing the objects themselves.
- the same type of relations can be applied (reused) to different types of objects.
- objects become simpler to design and to reuse.

Our implementation of the OR model, called Miori, is written in Smalltalk/80. This tool supports prototyping through direct manipulation techniques. The user is provided with a graphical tool, editors and browsers. Using the graphical tool a user can interactively manipulate relational objects in the relational class hierarchy. The tool supports operations such as
create, inspect or delete relational objects. Designers using Midori, can explore different system behaviors by assigning different relations among different objects. The ability to define relations among objects without modifying the objects themselves reduces the time and the cost of redesigning existing objects as commonly done in other systems.

The next section will provides an overview of the OR model. Section 3 describes Midori. The Heap example is in Section 4.

2 The OR model

We start our exposition by describing the general model our prototyping system uses. This exposition will allow the reader to relate his/her concepts of system building blocks to their counterpart in our model.

Relations can be seen as class objects with the following features:

1. a set of assertions defining the relation
2. an optional functionality defining how to maintain the relation
3. a protocol for observing changes within related objects with the ability to propagate changes to related objects

Next we describe the object model. A detailed description of the relational model and its integration with the object model forming the Object-Relation model follows.

2.1 The Object Model

In traditional OOP languages the object class defines a template for local state variables (instance variables), a set of operations (instance methods) sharing the state variables, class state variables (class variables), a set of operations sharing the class variables (class methods), and class relationships (inheritance).

Mili[9] proposed to add internal and external relations to the object model. Internal and external relations are just two additional attributes that can be mapped into instance variables placed in the root of the class lattice (Object class). An internal relation defines a relationship among the object’s components. Haumann[4] calls this property an internal constraint which describes how a physical body holds together. An external relation describes the relation between an object and its environment. The external relations of an object do not affect the object’s definition.

Suppose one wishes to model a table. The internal relation of Table defines it in terms of a board and a set of legs. This internal relation details the connectivity between the board and the legs and imposes positional constraints on the legs with regard to the board. We say that this internal relation “fails” when a leg is removed from the board.

The board and the legs that have been put together to form Table exist as objects independently of Table. The relations between the leg objects and Table are external relations. Similarly, the board has an external relation with Table. Figure 1 shows the three objects involved and the relations between them.

The failure of an internal relation to hold is the driving force behind our approach to system prototyping. Consider, for instance, a stack (First-In-Last-Out) data structure. The stack consists of objects (stored in the stack) and a relational class (called a stack) that holds those objects through an internal relation. This internal relation may impose the following constraints:

\[ 0 \leq \text{NumberOfElements} \leq \text{StackSize} \]

If at any time this constraint fails to hold an error condition should be raised to indicate a flaw in the design of the stack. This behavior shows the similarity between the constraints imposed on an object relations and class invariant as used in the Object-Oriented programming language Eiffel[8].

A major problem in software production centers around adapting existing software to fit new solutions. With rapid prototyping the problem is just more acute. Any rapid adaptation must attempt to minimize the amount of change to existing software. We demonstrate that point with two very basic classes that are widely used together. The classes are integer and string. It is common to convert integers to strings.
and string to integers. So much so that most C language systems, for instance, provide functions for the bidirectional conversion.

In OOP, integers and strings are objects and instances of the corresponding classes. Given a class Integer and a class String, the programmer is required to code the conversion. For instance, when converting from string to integer the programmer will have to create a class string with conversion to integer as one of its methods. The result is the need to modify an existing class. When rapid-prototyping a large system, such modifications are not necessarily easy and, therefore, not rapid.

In our approach there is no need to modify any of the classes involved. By implication, such an approach will speed up the prototyping. With the integer and string class under discussion, we will define a relation String-Integer that will produce integers from strings and a relation Integer-String that will produce strings from integers. Figure 2 provides a view of the mapping between class String and class Integer using the two internal relations.

### 2.2 The Relational Model

In this subsection we discuss relations in detail.

The following properties are built into relational objects:

- **N-Tuple** \((X_1, X_2, \ldots, X_N)\)
- **Status** POSITIVE or NEGATIVE
- **Type** INCIDENTAL, ENFORCEABLE, or INVARIANT
- **Other state variables**

A binary relation holds between two components, or in our case, two objects. A general relation holds between N objects, in which case these N objects are referred to as an N-Tuple. If a relation holds, that is the assertion about the N-Tuple is true, the relation is said to be POSITIVE. If the relation fails to hold, the relations is said to be NEGATIVE. The relation may be INCIDENTAL, i.e., it holds or fails to hold without further action. Some relations must hold but may dynamically change from holding to non-holding. In such a case we want force the relation to hold, hence the term ENFORCEABLE. Some relations are INVARIANT, i.e., they must always hold.

The following set of operations and properties are built into the relational class.

- Membership function
- Satisfier
- Attributes
- Other generic actions

Each relation has a **membership function** which is a predicate that returns true when objects are related. If an object fails to satisfy the membership function, a **satisfier** may be invoked in order to force the object to conform with the function.

To illustrate the usage of the operations or properties introduced above, consider an example of a line connecting the points P1 and P2. Point P3 is the middle point of the line as shown in Figure 3a. The three points may be represented as instances of the class Point. P3 is constrained by a relational object of the relation Mid.Point, M.P as seen in Figure 3b.

The membership function of Mid.Point can be expressed as the following manipulation of the x and y coordinates of the three points involved.

\[
MF = \left(\frac{P_{1x} + P_{2x}}{2} = P_{3x}\right) \text{ AND } \left(\frac{P_{1y} + P_{2y}}{2} = P_{3y}\right)
\]
If \( P_1 \) or \( P_2 \) were to move, then the satisfier will compute new coordinates for point \( P_3 \) as follows:

\[
\begin{align*}
P_{3x} &= \frac{P_{1x} + P_{2x}}{2} \\
P_{3y} &= \frac{P_{1y} + P_{2y}}{2}
\end{align*}
\]

A satisfier also can be used to add new members, or restore previous members that failed the membership function.

Possible generic type of operations on relations are: create-Instance, delete-Instance, etc. Attributes define a set of properties that we usually associate with binary relations. Attributes are useful for building and maintaining relations. Among the most commonly used attributes are those that carry the names and connotation of attributes of binary relation e.g. symmetric, antisymmetric, transitive and reflexive. Such attributes are useful for both the programmer constructing the prototype and the implementation of the relational model.

Status variables capture the dynamic nature of the relations. A relation instance is POSITIVE if the membership function \( MF \) holds for the underlying tuple, and is NEGATIVE otherwise. A relational object whose status is NEGATIVE and its type is ENFORCEABLE will be enforced by the relation. A relational object that is INVARIENT expresses a relation that should never fail. If it fails, an error should indicate a flaw in the design.

In the middle point example the properties of the relation instance \( M-P \) may be described as follow:

**Status:** POSITIVE  
**Type:** ENFORCEABLE  
**Tuple:** \( <P_1,P_2,P_3> \)

Moving \( P_1 \) might result in the following sequence of events:

1. \( P_1 \) informs the relational object of a change by sending it a "changed" message.
2. Upon receiving the changed message, the status of the relational object is checked against the class membership function.
3. The membership function returns \( false \), and the relational object status is changed to NEGATIVE.
4. Since the relational object type is ENFORCEABLE, the satisfier of Mid-Point's forces \( P_3 \) back to the center of the line.

Figure 4: The structure hierarchy of the class Relation

5. The status of the relational object becomes POSITIVE.

Note that the discussion above indicates that a relational object of type INCIDENTAL does not use its satisfier nor is one required. Although INCIDENTAL relational objects have no effect on the related objects, maintaining them was found useful for debugging and testing purposes.

In order to shed some light on the internal mechanism of the OR model and maintaining relations, we introduce the generic operation "delete."

Sending the "delete" message to any object results in:

1. The delete message is propagated recursively throughout the receiver internal relation, informing all the receiver external relations to be removed.
2. If the received message came through the object’s external relation the object takes no action.

Another example of generic message propagation is the "status" message. The internal relation returns true or false based on the result returned by its own membership function. The status message will be propagated recursively. In similar fashion failing components can be traced and analyzed.

2.3 Relations as Classes

Viewing relations as classes provides a simple and uniform model. We identify a class Relation which can be used as an abstract class in a relation class hierarchy. An abstract class is a class with no instances. It captures the commonalities among subclasses that
are derived from it. The class Relation is a subclass of the Object class as depicted by Figure 4, thus making relations objects. The instance variables: Internal Relations, and External Relations as defined in the Object class are inherited in the Relation class. The membership function, the satisfier and the generic actions are all defined as class methods in each relation class. The attributes property is defined as a class variable, since it is a property of the class and not of its instances. A relational object defines an instance of a relational class were all the properties of a relational object such as N-tuple and status are defined.

Viewing classes as objects enables us to express relations among classes or relations among objects and classes. The OR model is complete, were relations can be expressed among any one of the components in Object-Relation model as described by figure 5.

Figure 5 displays all the possible relations in the Object-Relation model. Since relations were mapped to classes, we can generalize and categorize three types of relations: relations among objects, relations among classes and relations among objects and class objects. The former type of relations can be used to represent additional relations other than the "Instance of" relationship, as suggested by Almrode[2]. Relations between objects and class objects can be useful in prototyping systems.

2.4 Object Composition

A composite object is built by defining a set of instance variables, which will reference other objects, and by providing a set of methods to operate on those objects. A composite object can also be built by using the object's internal relations. The internal relation of an object can be assigned different relations at different times, thereby introducing a dynamic form of composition. An internal relation can reference other objects, including relational objects and the relations among them. In addition, it can express type and cardinality constraints on its members.

3 Midori: The Environment

To demonstrate construction of a system using the Object-Relation model, we use Midori, a support environment built on top of the Smalltalk-80 environment. At the center of Midori is an object-relation based editor. The editor shows the use of relations in designing systems, and enables the user to manipulate objects and relational objects interactively.

The editor provides two separate views; the object view is in the upper part of the display and the relation view is in the lower part of the display. Figure 6 demonstrates this capability.

In the relation view a user can apply any of the following operations:

1. Enforced/Not enforced: ENFORCEABLE relational objects have a "+" sign in their representation, while INCIDENTAL relational objects have a "-" sign.

2. Removed: removing relational object affects only the relational view. The relational representation is removed while the associated objects are not affected.
3. **Inquired**: inquiry about a relational object results in a short description of the relation name, state (POSITIVE or NEGATIVE), type (ENFORCEABLE or INCIDENTAL), and the type (class) of objects that associated with this instance in the help window.

In the object view a user can:

1. Edit graphical objects: create, copy, delete and modify their internal states. Copying an object does not include a copy of the relations associated with it. Deleting an object implies removing also all the relations that associated with it. Similarly, modifying an object internal state result in message propagation to all the relational objects associated with it.

2. Identify relations among objects: defining new relations among objects is done from a selection of predefined relation classes and by identifying the associated objects.

Midori supports the following capabilities:

- Create a graphical representation of the composed model
- Edit the graphical representation of the model interactively
- Compose new relations out of existing ones
- Detect conflicting relations
- Browse through the relation class hierarchy
- Manipulate relational objects in the same way other objects can be manipulated

In Figure 6, the upper Dialog-Box window represents the actual designed model while the bottom OR-View window provides the relational representation of the designed model. Each object in the upper window (e.g., button, frames) is represented in the bottom window by a small circle. Relations among the dialog box's objects (relational objects) are represented in the bottom window as diamond shapes with set of lines radiating to the related objects. In the lower window, using the OR-Menu, a user can define new relations, remove relations, inspect relations, combine relations, or modify relations. The behavior of the objects with respect to each other can be tested and inspected using both views. Changes applied in the upper window's objects (operation on the dialog box) are reflected in the lower window and vice versa.

**Figure 7: A heap structure**

4. **An Example: A Heap Data Structure**

This section demonstrates the construction of a data structure, i.e., a heap, using nontrivial relations. The relations used are composed of other relations. A heap is a tree of node. Each node in the heap has a key. The key of each node in a heap is constrained by the keys of its children (if they exist). The parent key is larger than or equal to the keys of each of its children. We present a heap using the OR model in Figures 7A through 7C.

Figure 7A shows a heap with only one node, N1. Figure 7B shows a heap that has a node with one child. The relational object R12, is an instance of the general relation class Larger-or-Equal (≥). R12 enforces N1's key to be larger than or equal to N2's key. Larger-or-Equal is an example of a relation that is likely to be reused by other applications. The membership function of the Larger-or-Equal relational class is expressed by the following Smalltalk/80 like code:

```
Membership_function: tuple
    -- array of two <a,b>
    return ((tuple first key) >= (tuple second key)).
```

tuple can be represented as an array of two objects. The function returns true if the key of the first object is larger than or equal to the key of the second object, otherwise it returns false. We use R12 to represent a node with one child; the first element is the parent node while the second entry is its child. The satisfier of the Larger-or-Equal relational class is presented next.

Satisfier: rel_obj
H is composed of R12 and R13. The relational object R13 is similar to R12, it defines a relation between N1 and N3 of the class Larger-or-Equal. While R12 and R13 still reference <N1,N2> or <N1,N3> respectively, nodes N1, N2 and N3 do not reference R12 or R13. All references to R12 or R13 are done by H. Hence, H encapsulates R12 and R13. H’s state variables are as follow:

Relation: Heap
Status: POSITIVE
Type: ENFORCEABLE
N-Tuple: <R12,R13>

The Heap’s membership function follows.

Membership function: tuple
-- tuple is an argument <R1,R2>
-- Assumptions:
-- 1) The tuple is composed of two
-- relational objects of
-- class Larger-or-Equal.
-- 2) Both relational objects have
-- one object in common:
-- R1 = <a,b> and R2 = <a,c>
| R1 R2 | -- temp. vars
R1 := tuple first.
-- retrieve first relation <a,b>
R2 := tuple second.
-- retrieve second relation <a,c>
return(R1 Membership_function) AND
(R2 Membership_function)

The parameter tuple contains two relational objects, R1 and R2, which are instances of the relation class Larger-or-Equal. It is assumed that the first object in R1’s tuple and the first object in R2’s tuple represent the same object. For example, if R1’s tuple is <a,b>, a represents the parent node and b its child. If R2’s tuple is <c,d>, were c is parent and d is its child, then a and c represents the same parent node, where b is the left child and d is the right child. The parent has two independent Larger-or-Equal relations with its children. The membership function of the parent is the conjunction of the membership functions of the members of the tuples (last line in the code). This conjunctive membership function returns true if the membership function of both relational objects return true.

The Smalltalk-like code for the satisfier the Heap appear next.

Satisfier: rel_obj
-- relational object of the class

--- class relational object
-- Assumptions:
-- (tuple’s second obj.’s key)
-- > (tuple’s first obj.’s key)
| temp tuple |
-- temporary variables.
tuple := rel_obj nTuple.
-- get the tuple
temp := tuple first key.
-- swap the keys
tuple first key: (tuple second key).
tuple second key: temp.

This method assumes that the membership function failed. To satisfy the constraint, as specified in the membership function, the satisfier swaps the keys between the two objects. Next, we describe the state variables of R12:

Relation: Larger_or_Equal
Status: POSITIVE
Type: ENFORCEABLE
N-Tuple: <N1,N2>

N1 and N2 represent the two nodes as seen in Figure 7B. The type of this relational object is ENFORCEABLE, to assure that R12 will be maintained even if the related objects change their values. R12 is referenced by N1’s internal relation while it is referenced in N2’s external relation. Figure 7C describes a heap with three nodes. Nodes N1, N2 and N3 can all be instances of the same class. The relational object H is an instance of the relation class Heap. The relational class Heap is a composite relational object; it encapsulates two relational objects into one relational object (see Figure 8).
Assumptions:
1) The rel-obj tuple contains two rel. objects R1, R2 of the Larger_or_Equal relation class.
2) at least one relational objects in rel-obj’s tuple fails to hold.

| RI R2 tuple left_child right_child |
tuple = rel-obj nTuple.
R1 := tuple first.
-- first relational object
R2 := tuple second.
-- second relational object
left_child := R1 tuple second.
right_child := R2 tuple second.
-- find the appropriate relational object to be enforced.
(left_child key) >= (right_child key)
ifTrue: [ RI enforce.
-- left child is larger. Enforce RI ]
ifFalse: [ -- else
R2 enforce.
-- right child is larger. Enforce R2 ]

The parameter rel-obj is a relational object of class Heap. It is assumed at the entry point to the satisfier that rel-obj’s status is NEGATIVE. Hence, at least one relational object in rel-obj’s tuple failed; the key of the parent node is not larger than (or equal) to the keys of both its children. The satisfier, finds the child node with the largest key, then it requests the appropriate relational object to be enforced.

We use a composite relational object in the Heap to assure that if one node is deleted only the relevant relations are removed. This can be done by first decomposing the relational object into its components (other relations) and then deleting the related relations, while maintaining the rest. If N3, for example, is deleted the following sequence of actions are taken:

1. The relational object H will be decomposed into its components R12, and R13.
2. R13 is removed since it references an object (N3) that does not exist any more, while R12 continues its role.
3. The relational object H is deleted.

This sequence of operations represents the transition from Figure 7C back to Figure 7B.

Figure 9A describes a heap where one of its nodes, N2, receives a message to change its value to 84. All the relational objects in this example are ENFORCEABLE. After a change in N2's value, see Figure 9B, N2 notifies its associated relation H3 of the change. H3, subsequent to receiving the “changed” message, requests its class (Heap) to checks whether its N-tuple is still a member of the relation. H3 fails to satisfy the membership function (the key of node N3 is not larger than or equal to the keys of its children), therefore the status of the relation becomes NEGATIVE.

Since H3 is ENFORCEABLE, the Heap’s satisfier enforces H3 back into the relation; the keys of N2 and N3 are swapped in Figure 10A. Upon receiving a new value, N3 notifies its associated relations H1 and H3 of the change. Next, the Heap’s membership function checks the status of H1 and H3. While H1 fails to hold making its status NEGATIVE, H3’s status is POSITIVE since its membership check results in true. Similarly, since H1’s type is ENFORCEABLE the satisfier enforces H1 back into the relation by swapping N3 and N4 keys as seen in Figure 10B.

This heap model is self-organizing. It shows how data structures can be composed and can maintain themselves using the OR model. The use and reuse of relations demonstrates the ease with which prototyp-
The OR model is a promising model for the support of rapid prototyping within the OOP paradigm. Due to the prevalence of OOP software, the ability to produce fast and minimally changed systems can be of significant importance.

The OR model is based on the ability to create a hierarchy of relations that has many of the advantages of the regular class hierarchy. Specifically, relations can be inherited, reused, added and deleted the same way one is already used to with a regular OOP model. Clearly, this adds a considerable potential to attempts to use the OR model for rapid prototyping purposes.

We are currently at work on more complicated systems in which the relations will be more complex and in which the reuse of relations will be more extensive. Due to the existence of the Midori environment, we already have the infrastructure needed for this type of work.

References


