From Relations to Objects

Javad Heshmati

Vrije Universiteit Brussel (1996)
Department of Computer Science

Thesis submited in view of obtaining the
degree of Master in Computer Science

Promoter: Prof. Dr. Theo D'Hondt

Contents

List of Figures

Abstract

Applying object-orieted technology to systems development is widely recognized as improving productivity and reducing system maintenance and costs. At the same time, relational technology has gained leverage in most businesses. There exist already several proposals to combine object-oriented programming with relational database systems. Yet, existing approaches do not support necessary com binations of object-oriented and relational technology in concert, like reengineering of existing relational data in an object-oriented way. In this thesis, I present a pattern *oriented approach which enables the application designer and programmer to map a given relationl schema to an object-oriented schema.*

Chapter 1

Introduction

The state of art of data-intensive application development environment is dominated by two technologies, namely relational database systems and object-oriented programming languages. Concerning relational database technology, it has been developed for over twenty years, it has proved stable, and as a consequence relational database systems are in wide spread use in industrial and commercial areas. Together with relational database systems there exists a legacy of relational applications, i.e., applications written in some procedural languages and manipulating data in traditional SQL-like way. Concerning the object-oriented development paradigm, it has evolved out of the need for improving productivity and reducing system maintenance costs [7]. Object oriented languages go already a long way in providing support for building such systems. Among the key features which object-oriented programming support are encapsulation of data and functions to objects, user-defined object type, message passing, inheritance and polymorphism [15].

The existence of two greatly varying technologies for application development raises the following problems:

- existing relational applications represent a vast investment for each company. No company may afford to throw away its data and its applications and build every thing from scratch. Thus a requirement when combining object-oriented and relational technologies is that the solution supports relational applications further on.
- the concepts behind the object-oriented paradigm are more expressive than the ones of the relational models. This semantic mismatch has to be tackled in some way or the other.

Existing approaches coping with the above problems can be divided into two groups: First, *reverse engineering of data* which attempts to extract abstract, semantic meaningful information from the concrete database contents to gain a conceptual understanding of the stored information [14, 4]. The target model of the abstraction process is in general a semantic data model which has more

expressive power than the implementation model on the basis of which the data has been stored. There exist approaches for reverse engineering relational databases to Entity Relationship (ER) or NIAM models, and from the obtained ER or NIAM model to object-oriented schemas [3]. Consequently, it is possible to abandon the relational technology to use object-oriented technology instead [5] and to incorporate the existing data of the old database into the new system by reverse engineering of the old data. At least two problems, however, remain with this approach:

- existing relational applications are no longer supported and have to be rewritten in some object-oriented way.
- most commercially available object-oriented database systems are not stable enough, yet[1].

Thus, for companies with major investments in relational technology, both in terms of money and development time, this isn't an attractive solution. The second group may be characterized as *combining object-oriented technology and relational technology* (see figure 1.1). Most existing solutions in this group provide predefined wrapper object classes which hide access to a relational database system [1]. These wrapper object classes model relational entities such as tables, tuples, and fields. Since an object-oriented model is more expressive than the relational model, the decomposition of an object into several tuples and vice versa must be hand-coded by the application programmer. This hand-coding process is complex. There is no cookbook method to achieve it. It takes experience and taste of design and programming to translate a relational schema into an object-oriented schema efficiently (some might say properly). In this thesis, I propose a translation methodology in design *pattern* form which provides the application programmer with a set of patterns (as shown in the middle layer of figure 1.1) for mapping a relational schema into an object-oriented schema.

1.1 What Is a Design Pattern?

In general, patterns help reduce the complexity in many real-life solutions. For example, in some solutions the sequence of actions is crucial in order to accomplish a certain task. Instead of having to choose from an almost infinite number of possible combinations of actions, patterns allow for the solution of problems by providing time-tested combinations that work. For example, car drivers apply a certain behaviour pattern to set a manual transmission car in motion. This balance of clutch and gas is applied no matter whether the car is a Volkswagen Beetle or a Porsche 959.

Patterns appear in a vast variety of domains in daily life. In fashion, various patterns exist that describe good-looking combinations of various aspects like

[1] New releases of object-oriented databases include major changes compared to their predecessors, and it is not yet predictable which system will survive in the market [12].

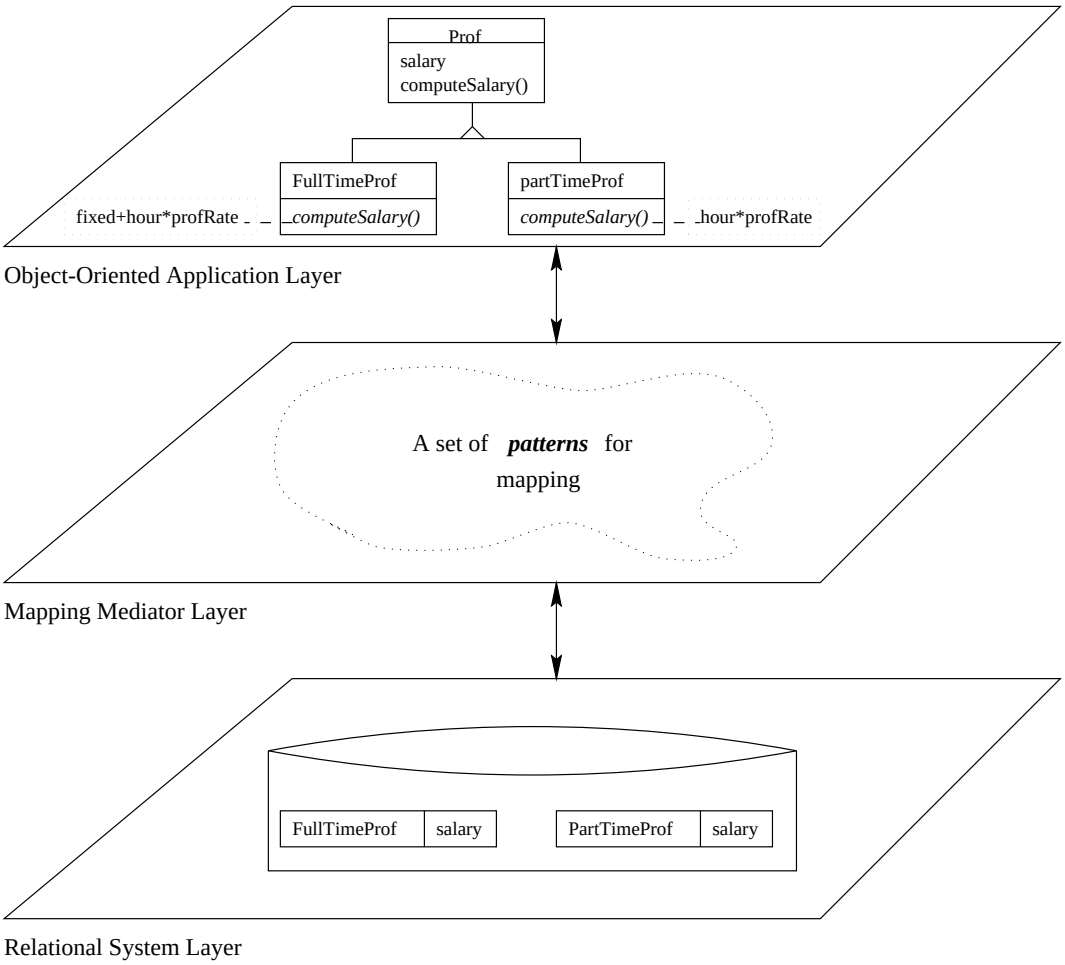


Figure 1.1: Combining Relational and Object-Oriented Technologies

colors. More or less mandatory rules (for example, behaviour patterns and traffic rules) might be derived from usable patterns.

Each culture has a set of behaviour patterns outlining how to behave in certain situations. For example, behaviour patterns describe how to eat a meal. In the Iranian culture, spoon and fork are used together for eating while in Europe these tools are used together only for certain tasks and Chinese use different tools for eating. Having said all this about patterns in general, How can the pattern concept be applied to software development, and specially to object-oriented software development?

Programmers tend to create parts of a program by imitating, though not directly copying, parts of programs written by other, more advanced programmers. This imitation involves noticing the *pattern* of some other code and adapting it

to the program at hand. Such imitation is as old as programming.

The design pattern concept can be viewed as an abstraction of this imitation activity. In other words, design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development. You might even include books on algorithms into the category of general design patterns. For example sorting algorithms describe how to sort elements in an efficient way depending on various contexts [18].

1.1.1 A Brief Historical Overview

In the 1960s, building architects were exploring the issues of design in its broadest sense, and investigating automated, computerized building design. There was keen interest in rules and algorithms that could transform requirements into a configuration of building modules. The mainstream of this movement moved on to what we know as a modular construction. The architect Christopher Alexander broke with that movement, noting that the great architecture of history were not made from rigorous, planned designs, but that their pieces were custom-fit to each other and to the building's surroundings. He also noted that some buildings were aesthetically pleasing than others, and that these aesthetics were often attuned to human needs and comforts. He found recurring themes in architecture, and captured them into descriptions (and instructions) that he called *patterns*. According to him,

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [2].

Software designers have discovered (and rediscovered) analogies between Alexandrian patterns and software architecture patterns over the past decade [6].

1.2 Why the Design Pattern Approach?

Design patterns are a promising technique for achieving widespread reuse of software architectures. Design patterns capture the static and dynamic structures and collaborations of components in solutions to problems that arise when building software in domains like databases[2]. Patterns aid the development of reusable component frameworks by expressing the structure and collaboration of participants in a software architecture at a level higher than source code or object-oriented design models which focus on individual objects and classes. Thus, patterns facilitate reuse of software architectures, even when other forms of reuse are infeasible.

[2]Design patterns are applied in various fields such as client/server programming, distributed processing, organizational design, software reuse, real-time systems, business and financial systems and human interface design [20].

- *Patterns description explicitly record design and implementation trade offs and design alternatives.* Because pattern descriptions explicitly enumerate consequences and implementation trade-offs (see section 1.3), they can be used to record why certain design choices were selected and others were rejected.

- *patterns improve communication within and across software development teams* because they provide developers with shared vocabulary and concepts. Patterns capture essential properties of software architecture, while suppressing details that are not relevant at a given level of abstraction. Thus, they provide a comprehensible way of documenting complex software architectures (like the one proposed in figure 1.1) by expressing the structure and collaboration of participants at a level higher that source code or object-oriented design models that focus on individual objects and classes.

- *patterns explicitly capture knowledge that experienced developers already understand explicitly.* Pattern descriptions explicitly codify knowledge the experts understood intuitively. The use of patterns help experts document, discuss, and reason systematically about sophisticated architectural concepts [20].

1.3 Describing Design Patterns

I used the following template to describe all the patterns presented in this thesis.

- **Pattern name** conveys the essence of the pattern.
- **Motivation** illustrates the design problem and how the OMT structures in the pattern present a solution to the problem.
  - **Examples** clarify the of the problem in the *motivation* section.
- **Structure** represents the pattern using the the Object Modeling Technique (OMT) notation.
- **Participants** name classes, objects and other OMT structures involved in the structure of the pattern and describe their responsibilities.
- **Discussion** tells you how to make practical use of the pattern and includes to subsection:
  - **Consequences** tells you what are the the trade-offs and results of using the pattern.
  - **Examples** illustrate how the pattern can be applied.
- **Preceding pattern** (optional) describes which pattern precede this one and why.
- **Related patterns** (optional) suggests other design patterns closely related to this one.
- *Side Trips* (optional) *describe issues that are closely relevant to the subject.*
- Marginal notes are used to emphasize important points.

1.4 How to Select a Design Pattern

To find the design pattern that's right for your problem, you can start by looking at the "Intent" column of the following table. Then scan the motivation section of the referred pattern, read trough the section to find out if it sounds relevant to your problem.

- If it does
  - see the example of the Motivation section and try to compare it to your case.
  - also see the example of the Discussion section.
- If it doesn't

* see the Related Patterns section to see if there is any related pattern which is relevant to your problem.

Pattern Name	Intent
Class Maker (9)	Creating classes
Inheritance Mapper (13)	Making super class subclass Relationship
Reference Mapper (17)	Making relationships among objects
Relationship Object Builder (24)	Creating explicit relationship objects

Chapter 2

Mapping Lexical and Non-Lexical Object Types

Lexical and non-lexical object types are the building blocks of NIAM diagrams from which elementary deep structure sentences and diagrams are built. In general, a NIAM diagram is built up of non-lexical object types, lexical object types, and fact type roles [22]. This chapter provides you with several patterns whose solutions can be used to map the NIAM Lexical and non-lexical object types to the appropriate OMT structures.

2.1 Class Maker

2.1.1 Motivation

Given a NIAM schema, it isn't unusual to find a non-lexical object which is related to one or more lexical object(s). For example, a person has a name, an ID and an address, this is depicted in figure 2.1. You can bundle these non-

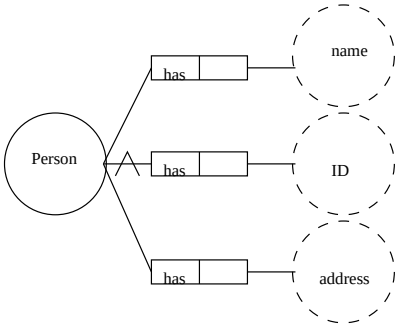


Figure 2.1: The Class Maker Motivation Example

lexical and lexical objects into an OMT class in which you also define some methods to perform the necessary operations.

2.1.2 Structure

The *initialize()* method is there to make sure that all the constraints (if there are any) are mapped at the creation time of the object.

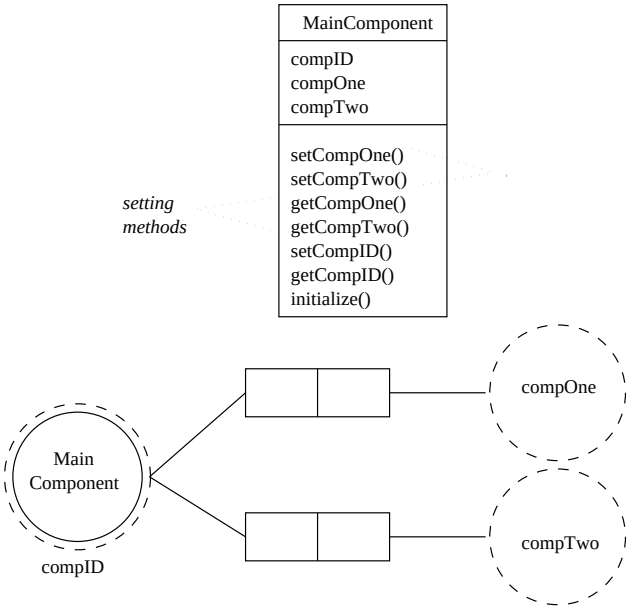


Figure 2.2: The Class Maker Structure

2.1.3 Participant

- **MainComponent** defines the interface for the MainComponent class by defining
  - **setting** methods set the instance variables.
  - **getting** methods are used to access the instance variables.
  - the **initialize** method is used to initialize the object ID and the instance variables with their initial value at the creation time of the object.

2.1.4 Discussion

The structure of this pattern is very clear and doesn't need much explanation. However, it should be stressed that the initialize() method must always be invoked at the creation time of the object. Because the initialize method initializes

the object with its identity (ID) and also makes sure that all the constraint associated with the object are mapped at the creation time of the object. Although object-oriented languages generate a unique identity per object [1], but it is still a good practice to define an explicit ID per object. There are many advantages associated with the usage of IDs. (see the Consequences section)

Side Trip

The manual call to the initialize() method to provide initialization for class objects is inelegant and error prone. Because a programmer can forget to call the initialize() method when creating the object, or (often with equally disastrous results) call the method twice. A language like C++ lets the programmer declare a function with the explicit purpose of initializing objects. Such a function is called a constructor [21]. The object constructor which is responsible for the initialization of the object, gets called automatically by the C++ complier, at the creation time of the object.

Consequences

There are benefits to using IDs:

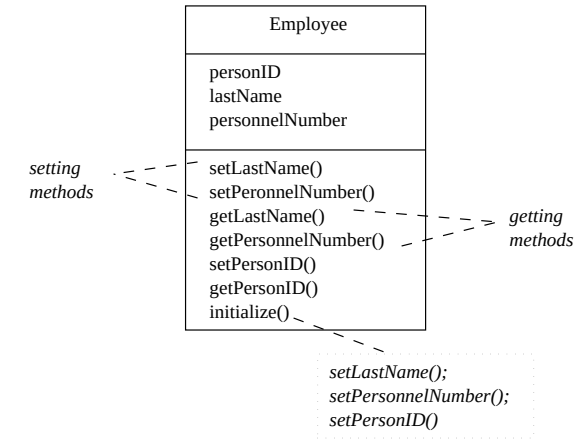
- IDs are immutable and completely independent of changes in data value and physical location. The stability of IDs is particularly important for associations since they refer to objects. Contrast this with referring to objects by name. Then changing a name requires update of many associations.
- IDs provide a uniform mechanism for referencing objects.

Of course, generating IDs is a nuisance, for which RDBMS provide no inherent support. Therefore, use IDs for applications where users directly access the database, because people think in terms of descriptive properties such as names and not in terms of artificial numbers. However, if the database access is restricted via programs, you can use the generated object identities that are generated and managed by your object-oriented environment [19].

[1] Object-oriented languages implement identity with pointers or look-up tables into pointers; an ID is the equivalent database construct.

Example

As you see in the figure below, every employee must have a last name, an ID and a personnel number. Therefore, we need to call the initialize method[2] to initialize the employee object with the initial values of its instance variables, last name, ID and personnel number.



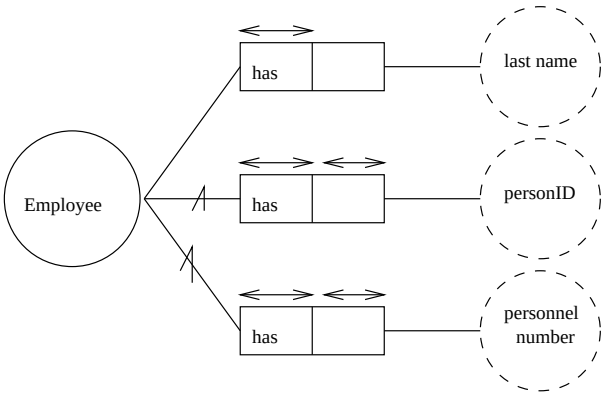


Figure 2.3: The Class Maker Example

2.1.5 Relevant Patterns

The Relationship Object Builder (see section 3.1 on page 24) can be used to handle constraints at run time (see figure 3.4).

[2]Of course this method should get called at the creation time of the object.

2.2 Inheritance Mapper (Subtyping)

Fortunately, The NIAM method of analysis includes the notion of subtypes and in so doing spells out clearly what is a subtype of what. This is achieved by displaying subtypes *outside* their supertype(s) and depicting the subtype-supertype connections by means of an *arrow*.

The process of mapping the NIAM subtype-supertype relationship to the corresponding OMT relationships which is often called *inheritance* is a straight forward process, but it is important to understand the difference between an object's class and its type.

> An object class defines how the object is implemented. The class defines the object's internal state and the implementation of its operations. In contrast, an object's type only refers to its interface-the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type. Of course, there's a close relationship between class and type. Because a class defines the operations an object can perform, it also defines the object's type. When we say that an object is an instance of a class, we imply that the object supports the interface defined by the class [9].

Side Trip

> *In a statically typed language like C++, class is used to specify both an object's type and its implementation . Whereas, in Smalltalk, programmers* don't *declare the types of variables, the consequence is that sending a messages to a Smalltalk object only requires checking that the class of the receiver implements the message, but there is no requirement for the object to be an instance of a particular class. It should be mentioned that JAVA clearly distinguishes between an object's type and its implementation by introducing the notion of* interfaces *[8]. JAVA interfaces are data types, just as classes are, and when a class implements an interface, instances of that class can be assigned to variables of the interface type.*

2.2.1 Motivation

Subtyping is often used in NIAM schemas to generalize different types into a common supertype. Figure 2.4 shows how the Employee type is derived from its super type Person. The NIAM subtype/supertype relationship can be mapped

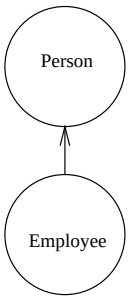


Figure 2.4: The Inheritance Mapper Motivation Example

into the OMT subclass/parent class structure. New parent classes and subclasses can be defined in terms of existing supertypes and subtypes.

2.2.2 Structure

In practice however, an object type may have many subtypes which overlap and can be generalized.

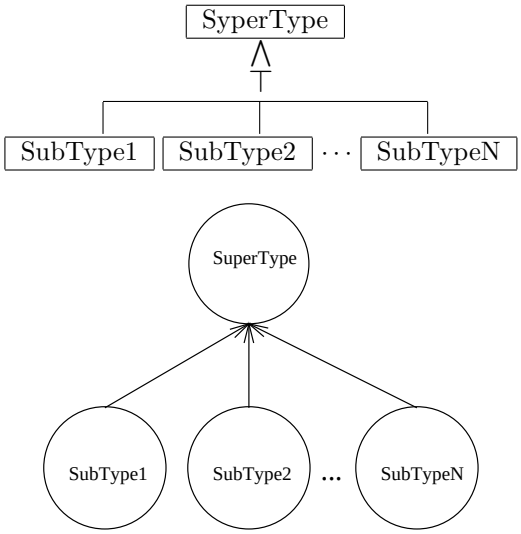


Figure 2.5: The Inheritance Mapper Structure

2.2.3 Participants

- **SuperType** defines and implements a generic interface for the super class.
- **SubType(s)**
  - define their own interfaces.
  - customize their parent's interface by adding new methods and overriding the super class method(s), if needed.

2.2.4 Discussion

Class inheritance is a mechanism that you can use to extend your application's functionality by reusing functionality in parent classes. You get new implementations almost for free, by inheriting from existing classes. Of course subclasses can also customize the functionality of parent classes by overriding the parent classes' methods.

Consequences

Using inheritance, you can

- extend your application's functionality by
  - adding operations to those already offered by parent classes.
  - override operations of parent classes.

Because of inheritance along the class hierarchy, a change to class in general does impact other classes and their instances, as well as its own instances. For example, if an attribute is dropped from the SuperType class, the values of the attribute in all existing instances of the SuperType must logically be dropped; and the attribute must also be dropped from all the direct and indirect subclasses of the SuperType class which had inherited it, and of course the values of the attribute must also be dropped from the instances of the subclasses [13]. If you intend to avoid this, then you should look for some alternative approaches (see section 2.2.6).

Example

In the example shown in figure 2.6, the Prof type is the super type of both FullTimeProf and PartTimeProf. Applying the Subtyping pattern we obtain the corresponding OMT inheritance relationship. As you see, both FullTimeProf and PartTimeProf override the computeSalary() method of their super class.

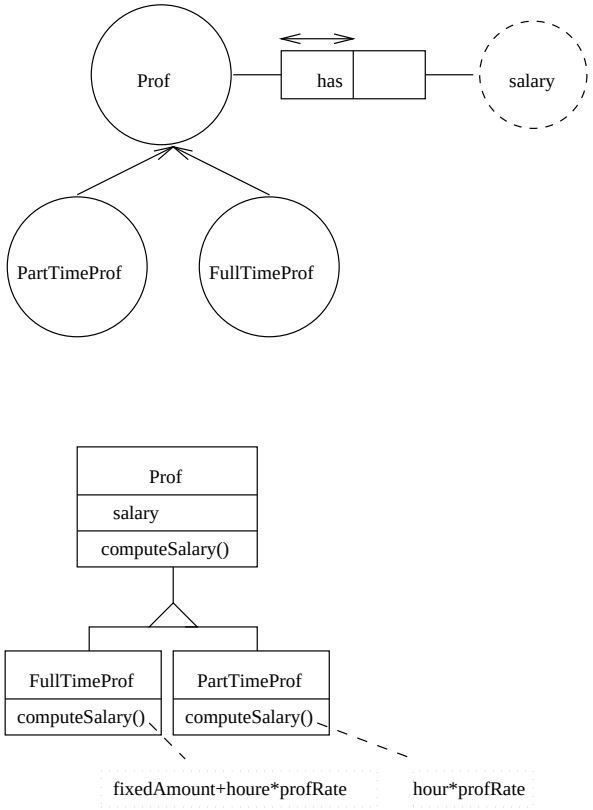


Figure 2.6: The Inheritance Maker Example

2.2.5 Preceding Patterns

The Class Maker pattern (see section 2.1 on page: 9) should precede this pattern.

2.2.6 Related Patterns

The Reference Maker pattern (see section 2.3 on page: 17) is closely related and might be considered as an alternative approach.

2.3 Reference Maker

2.3.1 Motivation

You might come across a lexical object type related to a non-lexical object type through a number of relationships (see figure 2.7). This pattern combines two approaches and presents it as a solution for mapping these object types and their relationships to the OMT structures.

Example

The following NIAM diagram illustrate the description of a loan card:

- "loan made on date"
- "loan due on date"
- "loan terminated on date"

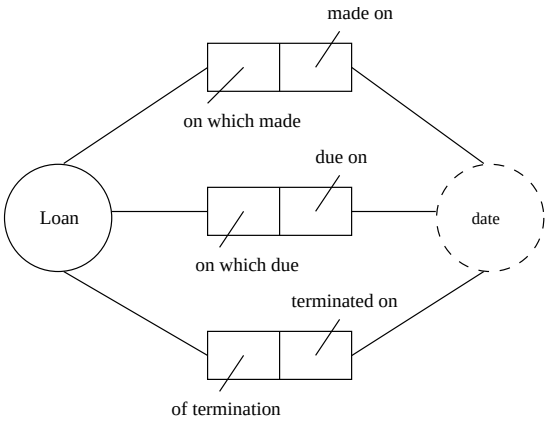
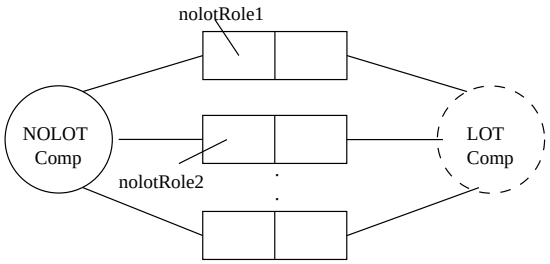
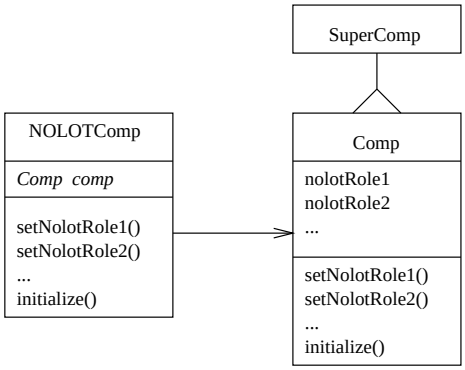


Figure 2.7: The Reference Maker Motivation Example

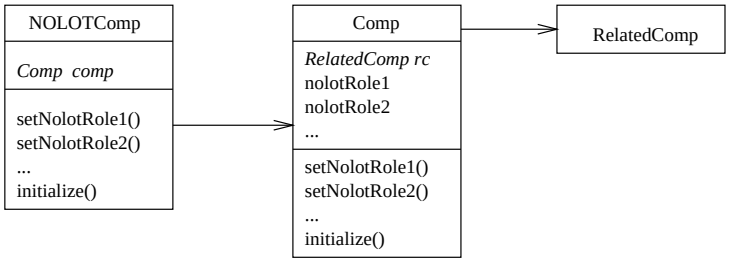
2.3.2 Structure



Choose the *White-box approach*, if inheritance makes sense and you can obtain a lot of functionality by subclassing. Choose the *black-box approach*, if the objects that you are dealing with have well-defined interfaces.



White-box approach



Black-box approach

Figure 2.8: The Reference Maker Structure

2.3.3 Participants

- **NOLOTComp**
  - defines the interface.
  - keeps a reference to the Comp object.
- **Comp**
  - implements the interface defined by the LOTComp.
  - depending on the selected approach, either it keeps a reference to the SuperComp or inherits from it. Therefore, if
    - **White-box** approach is chosen, the Comp class becomes the subclass of the SuperComp.
    - **Black-box** approach is chosen, the Comp would only keep a reference to the SuperComp.
- **SuperComp** is the super class of the Comp class and provides the Comp object with the basic functionalities through the inheritance mechanism.
- **RelatedComp** provides the Comp object with the basic functionality through the delegation mechanism.

2.3.4 Discussion

Using this pattern you should always decide which of the two approaches (White-box or black-box approaches) is most appropriate for your case. Of course, there is no *clear cut* set of criteria available to base your decision upon, but there are some trade-offs that you should be aware of before making your design choice.

Consequences

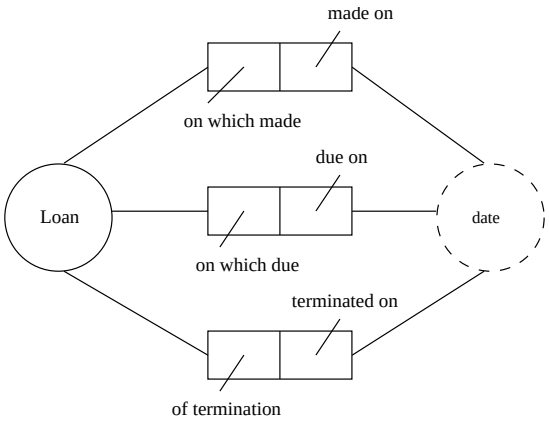
- The White-box approach is partly based on inheritance which means that each instance of the Comp is an instance of the SuperComp; thus all the operations and instance variables of the SuperComp must apply to the Comp. You should choose this approach if you think that the superclass/subclass relationship makes sense and you can benefit from this relationship by reusing the superclass code in the subclass. However, improper use of this approach leads to programs that are hard to *maintain* and *extend*.
- Sometimes it appears that use of inheritance would increase code reuse within a program, when a true superclass-subclass relationship doesn't exist. If this is the case at hand, *do not* go for the White-box approach, use *delegation* (Black-box approach) instead. This approach which is based on delegation provides a good mechanism for achieving the desired code reuse. Actually, in the Black-box approach, delegation is done at least twice (depending on your case):

1. The Comp object provides the NOLOTComp object with the desired operations (Needless to mention that the actual execution is done in the NOLOTComp object).
2. The SuperComp object provides the Comp object with the desired operation.

- using this approach you can avoid unexpected side effects, since each operation is explicitly forwarded.
- this approach makes it possible to encapsulate the behaviour of the NOLOTComp into the Comp object. This means that you can easily modify the implementation of the NOLOTComp class, just by changing it in the Comp class.
- obviously, this approach increases the complexity of your system. A design that uses this approach might end up with a number of little and big Lexical objects referenced through one another. Although, the resulting system can be easily maintained and customized by the designer himself, an outsider might have a difficult time understanding the system.

Example

The top part of figure 2.9 shows a NIAM diagram which describes how the Loan and LoanDate object types are related to each other through a number of relationships. The bottom part of the figure shows the OMT diagram obtained after applying the Reference Maker pattern based on the black-box approach.



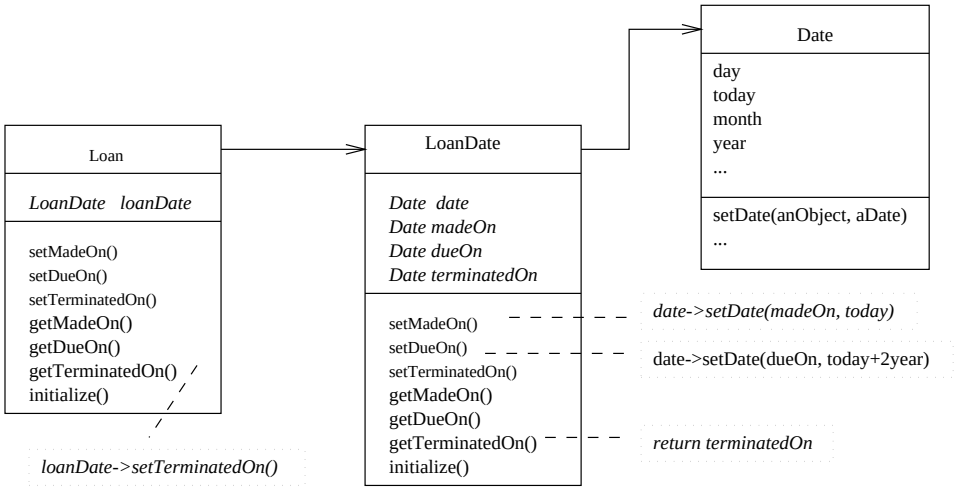


Figure 2.9: The Reference Maker Example

2.3.5 Preceding Patterns

The Class Maker pattern (see section 2.1 on page: 9) must precede this pattern to build the class structures and initialize the related constraints at the creation time of the object.

2.3.6 Related Patterns

See the Relationship Object Builder pattern (section 3.1 on page: 24).

Chapter 3

Mapping Relationships

3.1 Relationship Object Builder

In the NIAM method of analysis, the relationships between object types are expressed directly. Object-oriented analysis techniques such as OMT also construct models using both objects and relationships between objects. However, object-oriented programming languages (such as Smalltalk[1] and C++) don't support relationships directly, they are only based on objects. Therefore, relationships obtained from analysis are usually lost during design and implementation.

3.1.1 Motivation

Consider the following NIAM diagram in which "Account" and "Transaction" are related to each other through many relationships. We can convert this NIAM diagram to the corresponding OMT model by creating two classes (Account and Transaction) which also contain the association between these two non-lexical objects. However, by associating all the system behaviour with the objects and non with the relationships between objects, we introduce a lot of complexity to our objects. Using the Relationship Object Builder pattern, we can avoid this complexity by transforming the relationships between these non-lexical objects into explicit objects. These explicit objects can then be implemented directly in an object oriented language.

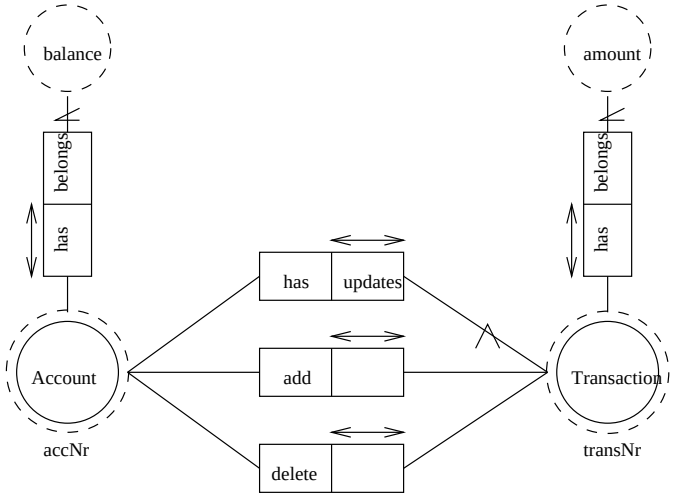


Figure 3.1: The Explicit Object Builder Motivation Example

[1] Although, Smalltalk doesn't support relationship objects directly, it does provide several examples of objects being used to explicitly represent relationships (see the *Side Trip* at the end of the section).

3.1.2 Structure

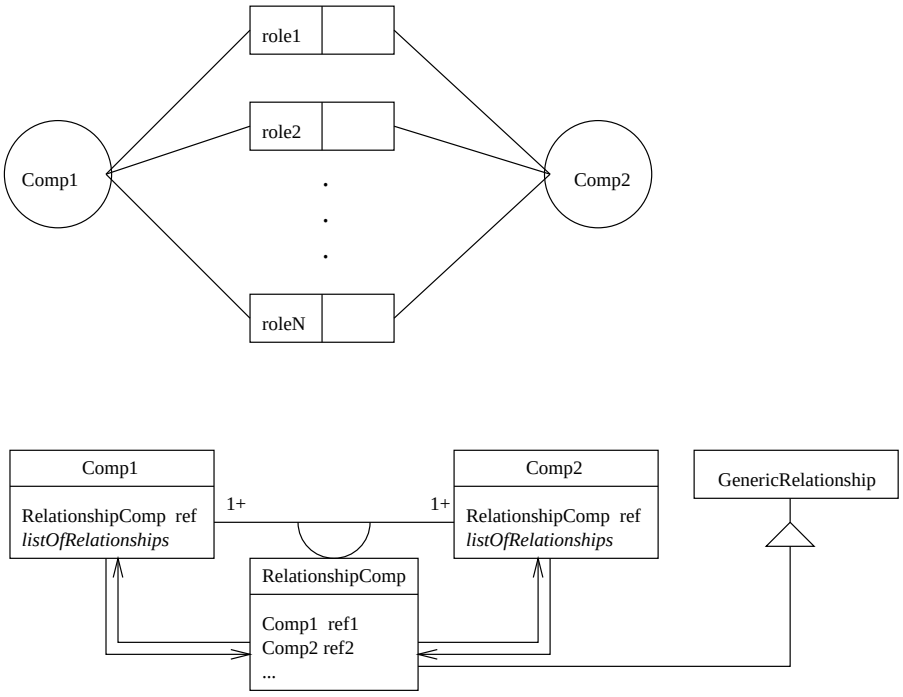


Figure 3.2: The Relationship Object Builder Structure

3.1.3 Participants

- **Comp1**
  - defines its own interface and implements its own specialized methods.
  - keeps reference(s) to the RelationshipComp object(s).
  - keeps a list of its dependent objects for notifications. Namely, each object which participates in relationships stores their relationships in a list of dependents, when a participating object is changed, it sends a single "update()" message to all relationships on its list of dependents. [2](see figure 3.4)
- **Comp2** does the same thing as the Comp1.
- **RelationshipComp** incorporates
  - references to the objects participating in the relationship.
  - methods to manage adding, deleting, modifying the objects being related.
- **GenericRelationship** implements some general methods such as *checkConstraint(), objectDeleted(), objectModified()*, and so on.

3.1.4 Discussion

Using this pattern your design includes object relationships explicitly. Behaviour which is associated with relationships can then be put into the relation ship objects. Of course, it is possible to omit the RelationshipComp and associate all of the system behaviour with the Comp1 and Comp2 objects. However, doing that means, Comp1 and Comp2 classes quickly become large and complex, because they should take care of all the functionalities associated with the inter-object relationships.

Consequences

There are a number of advantages associated with the solution which is presented by this pattern:

- objects tend to be smaller and more usable. For example in the following example, the behaviour associated with the relationships is abstracted into methods of the relationships object (AccountTransaction) new constraints and functionality associated with this relationship object is easier to add to the evolving system.
- explicit relationship objects improve the traceability of object oriented development. Every object in the design (see the structure) can be traced back to either single object or single relationship from analysis.

[2]You can also use the *Observer* [9] pattern for notifying the dependent objects (see the related pattern section).

- relationship objects increase the cohesion of the participating objects, as those objects contain the core of data and behaviour, while reducing the coupling, as the behaviour within the participating objects relating to each other has been moved into the relationship objects.
- there are also several objections to the solution presented by this pattern:
  - using relationship object may be considered as "not being pure object orientation".
  - using this pattern might make your design more complex than an equivalent design without relationship objects, because as a result of applying this pattern, you'll end up with a design which include more objects than the equivalent design without relationship objects. However, as it was mentioned before, your objects become smaller and simpler.

Example

Consider the "AccountTransaction" relationship. This object handles the addition of new transactions into the relationship; the deletion of a transaction; updating the balance of the account if a transaction is added, deleted or updated (see the figure below). Of course the AccountTransaction class could be extended to do more operations such as sorting, printing and copying for the account.

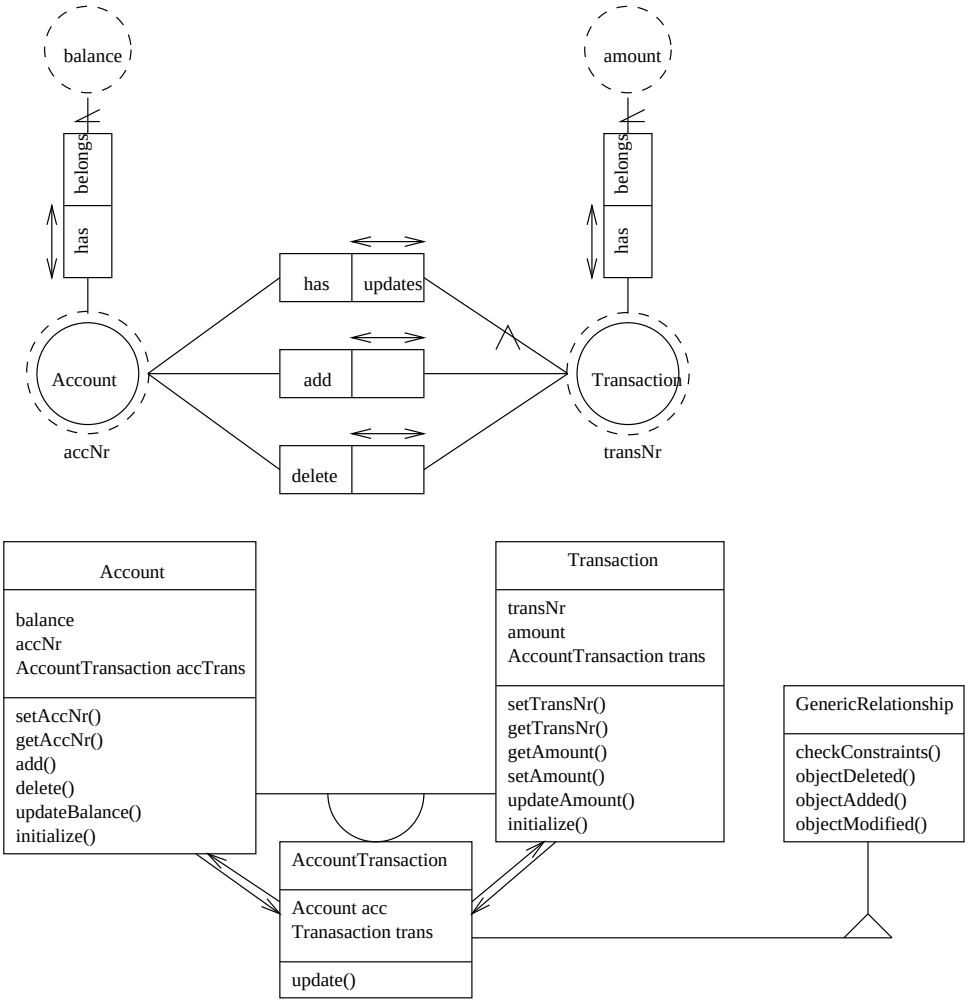


Figure 3.3: The Relationship Object Builder Example

For example, if a transaction object is added, deleted or modified, the AccountTransaction relationship's *update()* is called and this then updates the Account's balance appropriately (see figure below).

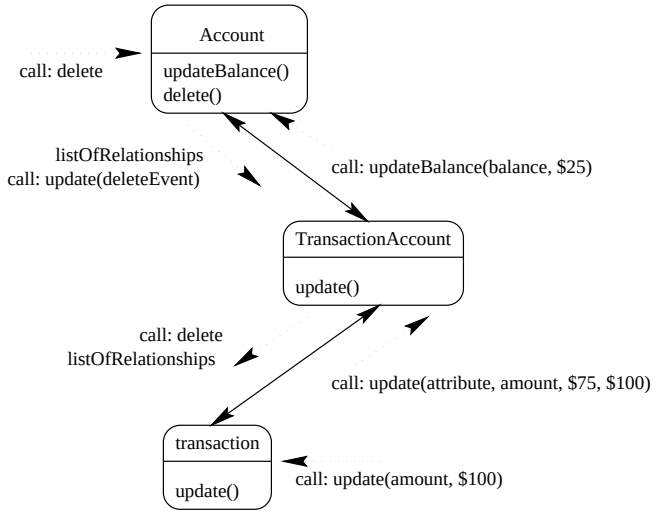


Figure 3.4: The Transaction/Account Interaction

3.1.5 Proceeding Pattern

Class Maker (see section 2.1 on page: 9) should precede this pattern to map Comp1 and Comp2 and their related LOTs to the corresponding OMT classes.

3.1.6 Related Patterns

Reference Maker (see section 2.3 on page: 17) can be applied to those relationships which are left as reference and are not converted to relationship objects. The Observer pattern [9] can be used so that each object which participates in relationships can store its relationships in a list of dependents, rather directly refering to its relationships. When a participating object is changed, it sends a signal "update:" message to all relationships on its list of dependents, rather than sending a message to each relationship individually.

Side Trip

The evolution of the Smalltalk system also provides several examples of objects being used to explicitly represent relationships, resulting in a simpler design. In original Smalltalk-80 system [10], a Smalltalk object can have dependents- other objects which are notified if the primary object changes. In Smalltalk-80, a behaviour to add and

remove an object's dependents, and to broadcast change notifications is included in the object class, which also maintains a global database of objects' dependents.[16]

Chapter 4

A Case Study: Mapping a Sample NIAM Schema to an OMT Schema

This chapter presents a case study in which a sample NIAM schema is mapped to an OMT schema by applying some of the patterns which were discussed in the previous chapter. The last section, presents a piece of Smalltalk code which implements the obtained OMT schema shown in figure 4.4.

4.1 The Sample NIAM Schema

Equipment such as computers can be organized into part-whole hierarchies. For example a computer has a chassis which can contain a hard disk, a mother board which in turns contain a CPU and some SIMMs. Figure 4.1 depicts the NIAM schema which describes the above case. Mapping this NIAM schema to the corresponding OMT schema can be done in three steps:

1. Apply the *Class Maker* (9) pattern to map the lexical and non-lexical object to the corresponding OMT classes.

2. Apply the *Inheritance Mapper* (13) to map the NIAM subtying relationships to the OMT inheritance relationships.

3. Apply the *Reference Maker* (17) to configure the references among the objects.

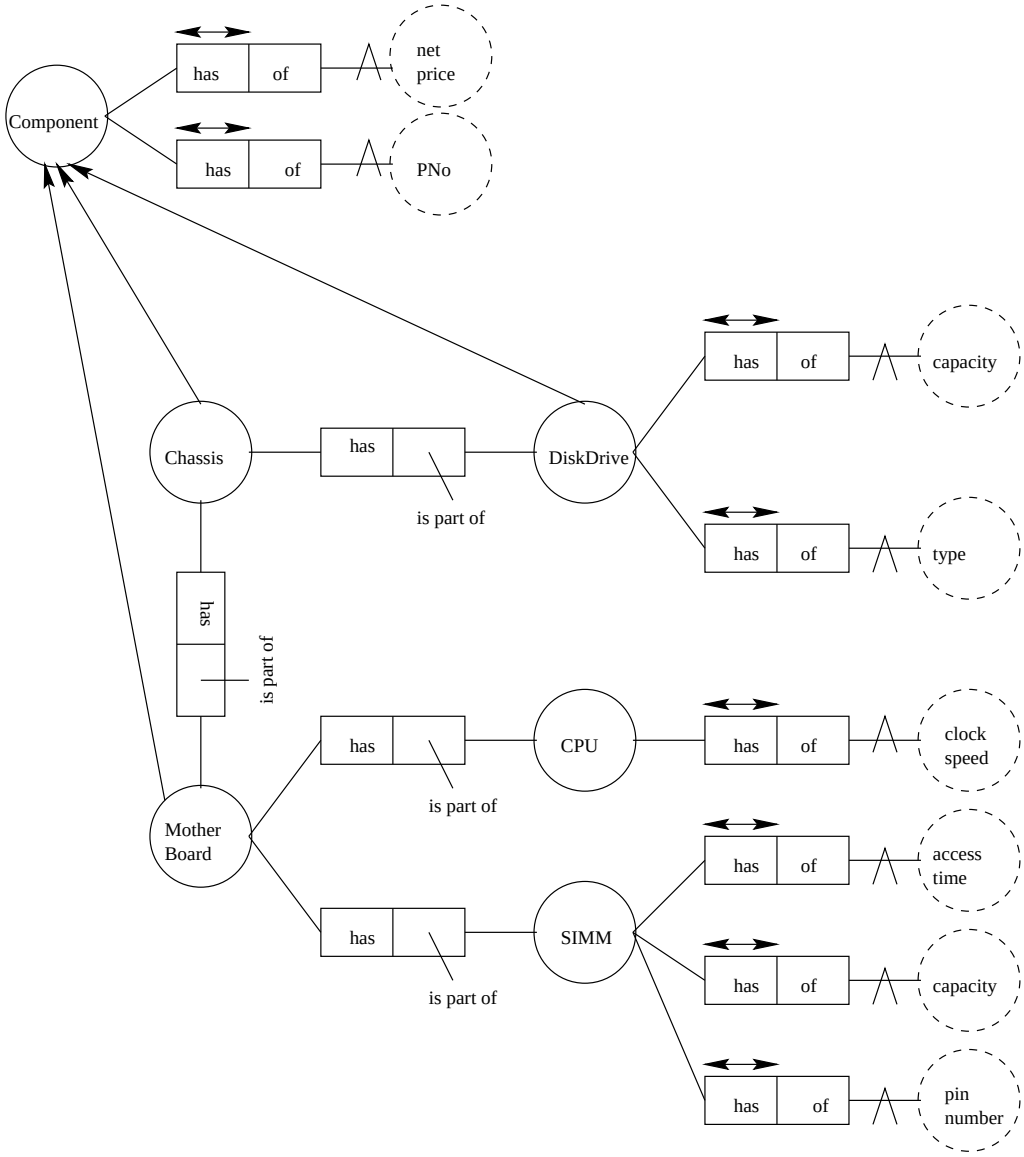


Figure 4.1: The NIAM Schema

4.2 Applying the Class Maker Pattern

As an initial strategy for dealing with non-lexical objects, we may consider them as objects in the OMT schema. The lexical objects of these non-lexical objects will become instance variables of the class instance. This can be achieved by applying the Class Maker pattern (see the figure below).

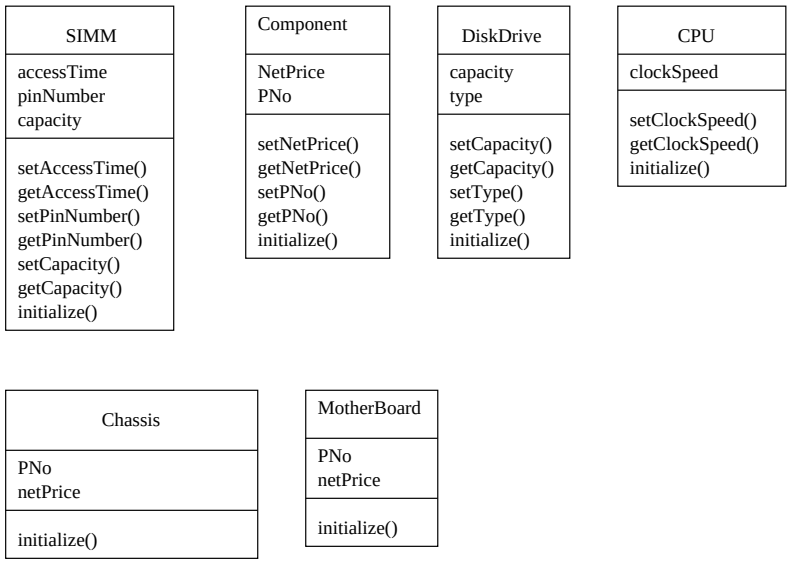


Figure 4.2: Applying the Class Maker pattern

4.3 Applying the Inheritance Mapper Pattern

The process of mapping the NIAM subtyping relationships to the OMT inheritance relationship is a straightforward process. By applying the Inheritance Mapper pattern, we obtain the following class hierarchy (see the figure below) where the Component class is the superclass of all the other classes.

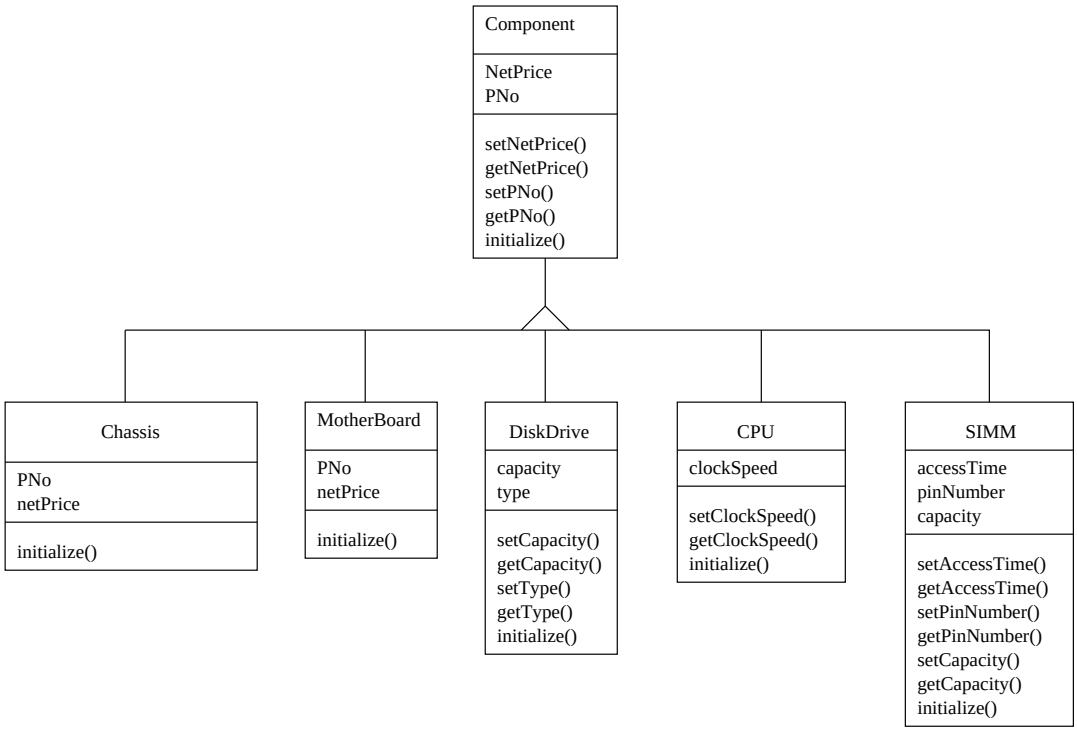


Figure 4.3: Applying the Subtyping pattern

4.4 Applying the Object Reference Maker

Having obtained the class hierarchy of our schema we need to figure out how these classes are related to one another. Namely, how these classes are referenced through one another. The Object Reference maker offers two solutions to this task: the White-box approach and the Black-box approach. Here, the black-box approach is chosen because:

1. The interfaces of the involved objects are well-defined.
2. The encapsulation of behaviour is needed, namely, Chassis encapsulates the behaviour of the MotherBoard and DiskDrive by keeping references to them.

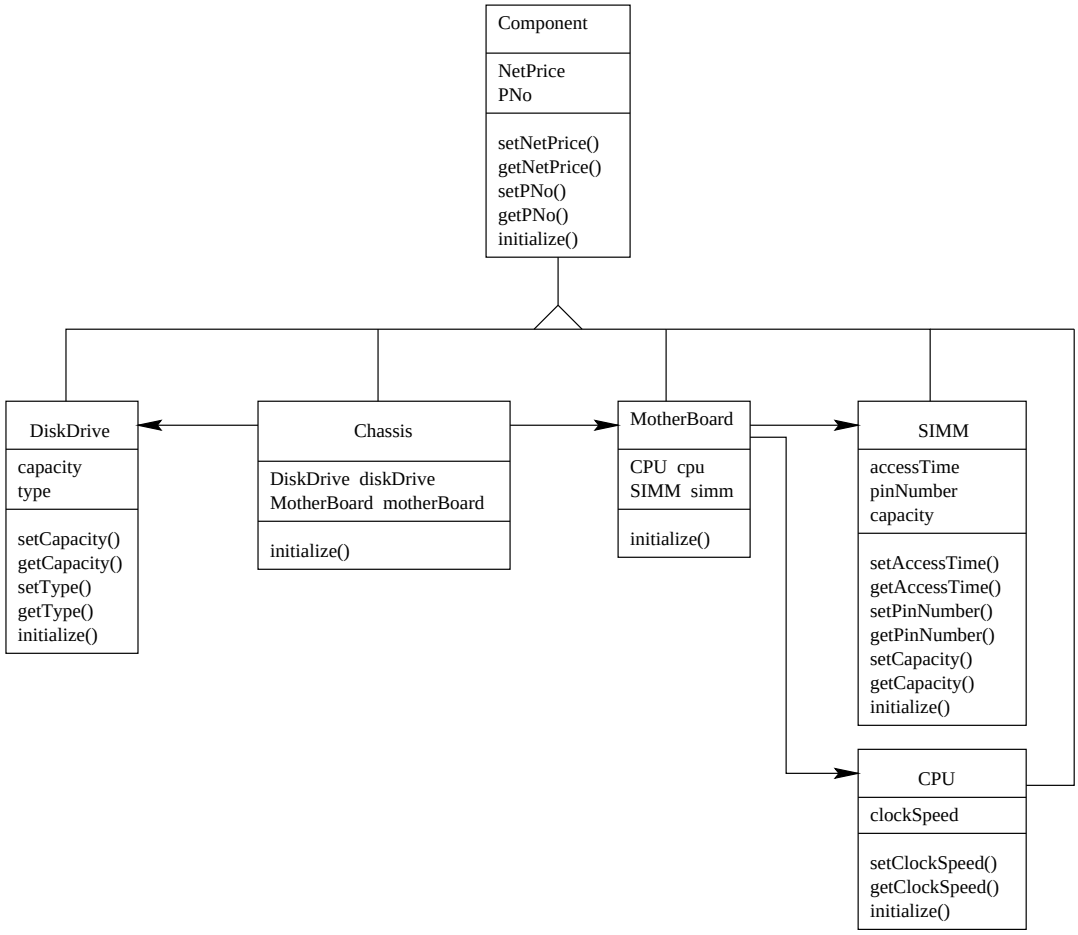


Figure 4.4: The Obtained OMT Schema

4.5 Sample Code

```
| chassis drive motherBoard cpu simm |

    "Creating an instance of Chassis"

        chassis := Chassis new.
        chassis initialize: '$50' partNumber: 'PN-001'.

    "Creating an instance of Mother Board"

        motherBoard := MotherBoard new.
        motherBoard initialize: '$500' partNumber: 'PN-002'.

    "Create an instance of Disk Drive"

        drive := DiskDrive new.
        drive initialize:'$300' partNumber:'PN-003' type:'HD' capacity: '2GB'.

    "Creating an instance of CPU"

        cpu := CPU new.
        cpu initialize: '$200' partNumber: 'PN-004' clockSpeed:'200MZ'.

    "Creating an instance of SIMM"

        simm := SIMM new.
        simm initialize: '$200' partNumber:'PN-005' pinNumber: '72' capac-
        ity:'8MB' accessTime: '60 ns'.

    "Adding the simm and the cpu to the mother board"

        motherBoard addEntity: cpu.
        motherBoard addEntity: simm.

    "Adding the hard disk and the motherboard to the chassis"

        chassis addEntity: drive.
        chassis addEntity: motherBoard.
        chassis getDependentEntity: drive.
```

Chapter 5

Conclusion

The relational database model was created to provide a simple abstraction that allowed the representation of large classes of data using small set of principles. Likewise, the object oriented model of programming was designed to allow the creation and representation of complex data structures in a coherent and uniform way. There exist a large class of applications for which relational database systems are too limited. These applications can be characterized as complex, large scale and data intensive programs. One example is a World Wide Web application. For companies with major investments in relational technology, both in terms of money and development time, it would be almost impossible to abandon the relational technology to use object-oriented technology instead. Thus, there is a need for combining relational and object-oriented technologies.

In this thesis I presented an object-oriented approach based on design *patterns* for mapping a relational schema to an object-oriented schema. The most salient advantages of this approach are the following:

1. an efficient mapping of existing relational schemas to an object-oriented one.

2. providing the application programmer with a set of patterns for converting a relational schema to an object-oriented one.

3. further applicability of existing relational applications.

Certainly, I don't presume to have covered all the possible design patterns, needed for achieving the above mentioned mapping. Rather, the design patterns presented in this thesis has captured a fraction of patterns needed to achieve a full mapping from a relational to an object-oriented schema. It would be worth while for someone to establish new patterns and add them to the set of patterns presented in this thesis.

Appendix A

The Smalltalk Implementation of the NIAM Case Study

```
Object subclass: #Component
        instanceVariableNames: '_net_price _p_number _parent'
        classVariableNames: ''
        poolDictionaries: ''
category: 'EntityWrapper'
Component methodsFor: 'printing'
         printOn: aStream

 Component methodsFor: 'accessing'


getNetPrice
        ^_net_price.

getPartNumber
        ^_p_number

parent
        ^_parent.

parent: aParent
        _parent := aParent.

setNetPrice: aPrice
        _net_price := aPrice.

setPartNumber: aNumber
        _p_number := aNumber.

 Component methodsFor: 'initialize-release'

initialize: netPrice partNumber: aNumber
        self setNetPrice: netPrice.
        self setPartNumber: aNumber


Component subclass: #CPU
        instanceVariableNames: '_clock_speed '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'EntityWrapper'


 CPU methodsFor: 'accessing'

getClockSpeed
```

```
        ^ _clock_speed

setClockSpeed: aNumber
        _clock_speed := aNumber.

 CPU methodsFor: 'printing'

 printOn: aStream
        super printOn: aStream.
        aStream nextPutAll: '===CPU:'; nextPutAll: 'Net price: '.
        aStream nextPutAll: self getNetPrice printString;
        nextPutAll: 'Part number: '; nextPutAll: self
getPartNumber printString; cr.

 CPU methodsFor: 'initialize-release'

initialize: aPrice partNumber: aNumber clockSpeed: aClockSpeed
        self setClockSpeed: aClockSpeed.
        self setPartNumber: aNumber.
        self setNetPrice: aPrice


Component subclass: #MotherBoard
        instanceVariableNames: '_cpu _simm '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'EntityWrapper'


 MotherBoard methodsFor: 'printing'


printOn: aStream
        super printOn: aStream.
        aStream nextPutAll: '==Mother Board:'; nextPutAll: 'Net price: '.
        aStream nextPutAll: self getNetPrice printString;
        nextPutAll: 'Part number: '; nextPutAll: self
getPartNumber printString; cr.
        self simm isNil
                ifTrue: [aStream nextPutAll: 'NO SIMM'; cr]
                ifFalse: [self simm printOn: aStream].
        self cpu isNil
                ifTrue: [aStream nextPutAll: 'NO CPU'; cr]
                ifFalse: [self cpu printOn: aStream].

 MotherBoard methodsFor: 'removing'
```

```
removeEntity:  anObject
        "removes entities according to their tags."

        anObject class = SIMM
                ifTrue:[_simm := nil].
        anObject class  = CPU
                ifTrue:[_cpu := nil].

 MotherBoard methodsFor: 'accessing'

cpu
        ^_cpu

getDependentEntity: aName
                aName class =  CPU
                        ifTrue:[self cpu].
                aName class = SIMM
                        ifTrue:[self simm].

simm
        ^_simm

 MotherBoard methodsFor: 'adding'

addEntity:  anObject

        anObject class =  CPU
                ifTrue:[_cpu := anObject. _cpu  parent: self].
        anObject class = SIMM
                ifTrue:[_simm := anObject. _simm parent: self].


Component subclass: #SIMM
        instanceVariableNames: '_access_time _pin_number _capacity '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'EntityWrapper'


 SIMM methodsFor: 'printing'

initialize: aClockSpeed
        self setClockSpeed: aClockSpeed
```

```
printOn: aStream
        super printOn: aStream.
        aStream nextPutAll: '===SIMM:'; nextPutAll: 'Net price: '.
        aStream nextPutAll: self getNetPrice printString;
        nextPutAll: 'Part number: '; nextPutAll: self
getPartNumber printString; cr.

 SIMM methodsFor: 'accessing'

getAccessTime
        ^_access_time.

getCapacity
        ^_capacity.

getPinNumber
        ^_pin_number

setAccessTime: aFloat
        _access_time := aFloat.

setCapacity: aNumber
        _capacity := aNumber.

setPinNumber: aNumber
        _pin_number := aNumber.

 SIMM methodsFor: 'initialize-release'

initialize: aPrice partNumber: aNumber pinNumber: aPinNumber
            capacity: aCapacity accessTime:
anAccessTime
        self setPinNumber: aPinNumber.
        self setCapacity: aCapacity.
        self setAccessTime: anAccessTime.
self setPartNumber: aNumber.
        self setNetPrice: aPrice


Component subclass: #Chassis
        instanceVariableNames: '_disk_drive _mother_board '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'EntityWrapper'
```

```
 Chassis methodsFor: 'removing'

removeEntity:  anObject
        "removes entities according to their tags."
        anObject class = DiskDrive
                ifTrue:[_disk_drive := nil].
        anObject class = MotherBoard
                ifTrue:[_mother_board := nil].

 Chassis methodsFor: 'adding'

addEntity: anObject
        "adds entities according to their tags."

        anObject class = DiskDrive
                ifTrue:
                        [_disk_drive := anObject.
                        _disk_drive parent: self].
        anObject class = MotherBoard
                ifTrue:
                        [_mother_board := anObject.
                        _mother_board parent: self]

 Chassis methodsFor: 'accessing'

diskDrive
        ^_disk_drive

getDependentEntity: aName
                aName class = DiskDrive
                        ifTrue:[self diskDrive].
                aName class =  MotherBoard
                        ifTrue:[self motherBoard].

motherBoard
        ^_mother_board

 Chassis methodsFor: 'printing'


printOn: aStream
        super printOn: aStream.
        aStream nextPutAll: '=Chassis:'; nextPutAll: 'Net price: '.
        aStream nextPutAll: self getNetPrice printString;
        nextPutAll: 'Part number: '; nextPutAll: self
getPartNumber printString; cr.
```

```
        self diskDrive isNil
                ifTrue: [aStream nextPutAll: 'NO DISK DRIVE'; cr]
                ifFalse: [self diskDrive printOn: aStream].
        self motherBoard isNil
                ifTrue: [aStream nextPutAll: 'NO MOTHER BOARD'
                         printString; cr]
                ifFalse: [self motherBoard printOn: aStream]


 Chassis methodsFor: 'initialize-release'

initialize: aPrice partNumber: aNumber
        super initialize: aPrice partNumber: aNumber


Component subclass: #DiskDrive
        instanceVariableNames: '_disk_drive _type _capacity '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'EntityWrapper'


 DiskDrive methodsFor: 'accessing'

diskDrive
        ^ _disk_drive.

diskDrive: aDiskDrive
        _disk_drive := aDiskDrive.

getCapacity
        ^_capacity

getType
        ^_type.

setCapacity: aCapacity
        _capacity := aCapacity.

setType: aType
        _type := aType.

 DiskDrive methodsFor: 'printing'


printOn: aStream
```

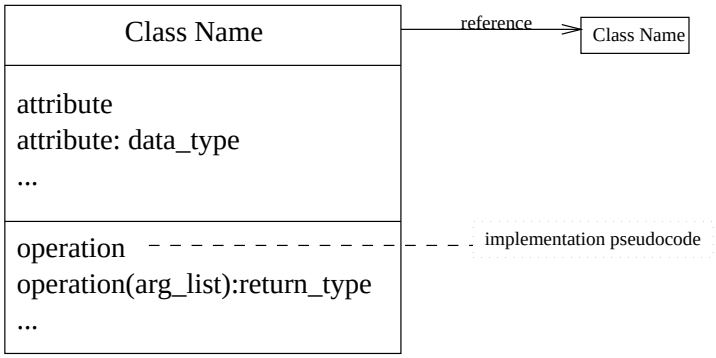
```
        super printOn: aStream.
        aStream nextPutAll: '==Disk Drive:'; nextPutAll: 'Net price: '.
        aStream nextPutAll: self getNetPrice printString;
        nextPutAll: 'Part number: '; nextPutAll: self
getPartNumber printString; cr.

 DiskDrive methodsFor: 'initialize-release'

initialize: aPrice partNumber: aNumber type: aType
            d
capacity: aCapacity
        self setType: aType.
        self setCapacity: aCapacity.
        self setPartNumber: aNumber.
        self setNetPrice: aPrice
```

Appendix B

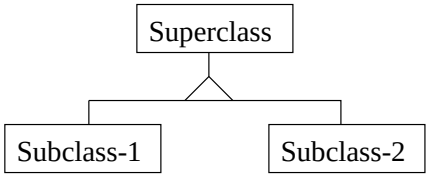
Guide to Notations

B.1 The OMT Guide

Class:



Generalization (Inheritance):



Aggregation:

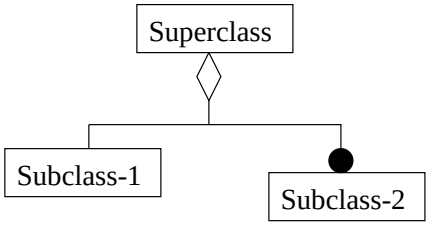
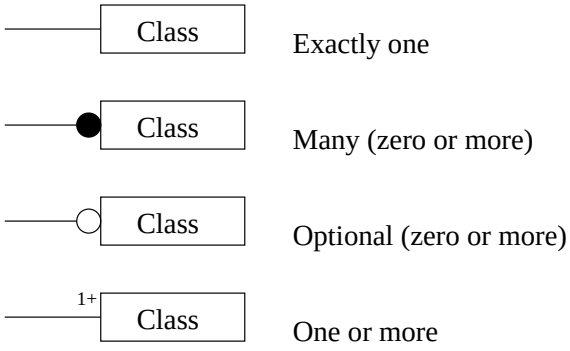
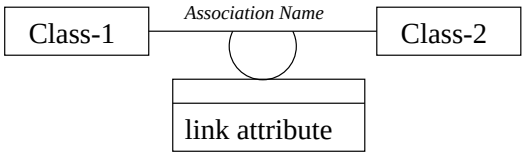


Figure B.1: The OMT Object Model



Link Attribute:



Association as class:

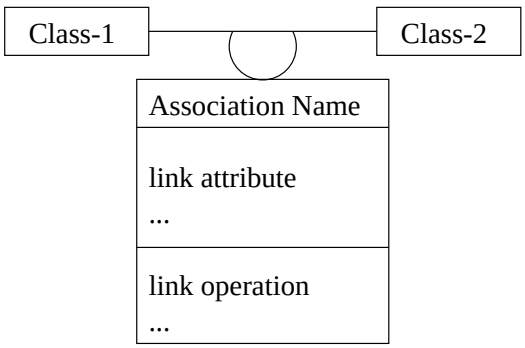
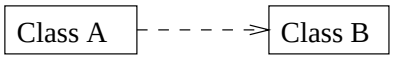
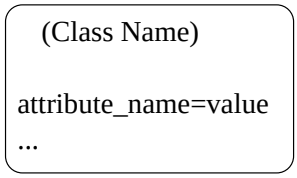


Figure B.2: The OMT Associations

Another useful thing to show is which class instantiate which others. I use dased arrowheaded line to indicate this, since OMT doesn't support it, this is called the *"creates"* relationship [9]. The arrow points to the class that's instantiated. In the following example, class "A" instantiates class "B".



Object Instances:



Instantiation Relationship:

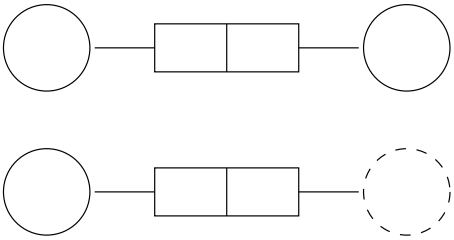
B.2 The NIAM Guide

Object type:

non-lexical object type

lexical object type

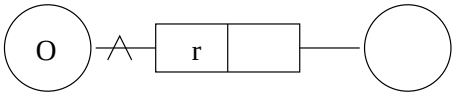
Fact type symbol:



idea type

bridge type

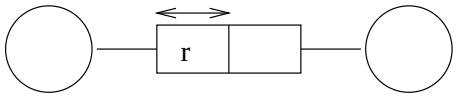
Totality symbol:



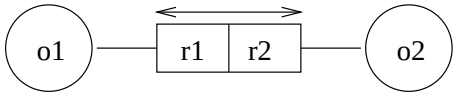
Role r of object type O is total

Figure B.3: The NIAM Object Types

Uniqueness symbol:

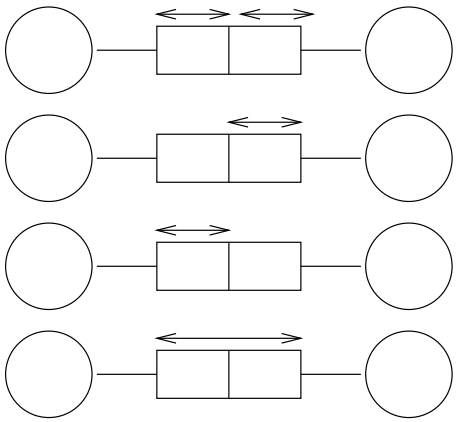


Role r of object type O is unique

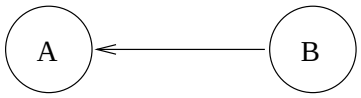


The combination of roles r1 of object type O1 and r2 of object type O2 is unique.

The four possible uniqueness constraints in binary fact types:



Subtype symbol:



Non-lexical object type B is a subtype of non-lexical object type A.

Non-lexical object type A is a supertype of non-lexical object type B.

Figure B.4: The NIAM Roles

Bibliography

[1] *Smalltalk(SQL) 2.0.* Synergistic Solutions Inc., Kendall Park, New Jersey, 1990.

[2] Christopher Alexander, Sera Ishikawa, Murry Silverstein, Max Jacobson, Ingrid, Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* University Press, New York, 1977.

[3] Sundaresan Jayaraman Badri Narashimhan, Shamkant B. Navathe. Entity-relationship approach-er '93. In *Entity-Relationship Approach-ER '93*, pages 401–414, 1993.

[4] C. Batini and S.B. Navathe. *Conceptual Database Design.* The Benjamin/Cummings Publishing Company, Inc, California, 1992.

[5] R.G.G. Cattell. *Object-Oriented and Extended Relational Database Systems.* Professional Computing Series. Addison Wesley Publishing Company, Massachusetts, 1991.

[6] James O. Coplien. Software design patterns: Common questions and answers.

[7] G. Engels and G. Kappel. Will the new approach solve old problems? volume 3 of *Object-Oriented systems Development.* IFIP94 World Congress, North Holland, 1994.

[8] David Flanagan. *Java in a Nutshell.* O'Reilly & Assiciate, Inc, 1996.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Professional Computing Series. Addison Wesley, 1995.

[10] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its implementation.* Addision-Wesley, 1983.

[11] Terry Halpain. *Conceptual Schema & Relational Database Design.* Prentice Hall, secound edition, 1995.

[12] G. Kappel, S. Preishuber, E. Prol, S. Rausch, W. Retschitzegger, R. Wagner, and C. Gierlinger. Coman - coexistence of object-oriented and relational technology. Technical Report Austria, University of Linz.

[13] Won Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.

[14] C. McClure. *The Three Rs of Software Automation - Re-engineering, Repositoty, Reusablility.* Professional Computing Series. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[15] O.M. Nierstrasz. Object-oriented concepts: A survay of object-oriented concepts. In *Databases and Applications.* ACM Press and Addision Wesley, 1989.

[16] James Noble and John Grundy. Explicit relationships in object oriented development. 1994.

[17] ParcPlace Systems Inc., 999 E. Arques Avenue, Sunnyvale. *Visual Works*, October 1994. Revision 2.0.

[18] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley Publishing Company, 1995.

[19] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[20] Douglas C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communication of the ACM*, 38(10):73, October 1995.

[21] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, second edition, 1995.

[22] J.J.V.R. Wintraecken. *The NIAM Information Analysis Method.* Kluwer Academic Publishers, 1991.