Java binding of the ODMG standard.

Ye. Tian

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/2532

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
Java Binding of the ODMG Standard

By

Ye TIAN

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2001
© 2001 Ye TIAN
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

0-612-67604-8
Abstract

The core of the ODMG (Object Database Management Group) standard is the ODMG object model. Two of them are important: OQL (Object Query Language) and ODL (Object Definition Language)[7]. The ODMG standard also includes bindings for well-known programming languages. Such bindings are defined for C++, Java, and Smalltalk [7]. However, there are some unforeseen and serious problems in matching OQL with the type systems of the programming languages bindings of the ODMG standard. Runtime errors are generally more critical in database systems than in programming languages and systems. An expensive recovery procedure may have to be invoked just because of a type error.

In this paper, I consider the result that OQL queries cannot be type-checked in the type system underlying the ODMG object model and its definition language ODL. A really disturbing result is that OQL queries cannot be type-checked in the type system of the Java binding of the ODMG standard either. This problem has a solution within the framework of the Java type system, but it is not an attractive one. If explicit type casts, type-checking OQL queries become possible. However, It requires specifying redundant type information in a query. At the same time, type checking must be carried out dynamically, which is contrary to the intent of OQL and ODL. In contradistinction to the situation described above, a positive result established is that type-checking OQL queries presents no problem for a type system with the basic form of parametric polymorphism (universal type quantification). A corollary to this result is that the type system of the Java binding of the ODMG standard allows static type-checking of OQL queries. This is made possible if we add parametric polymorphism facilities to Java.

Key words: OQL, ODL, ODMG, object model, type casts, frameworks, parametric polymorphism, Java binding, Smalltalk, Runtime errors, static type-checking.
To my parents and my sister, with love.
Acknowledgments

I would like to express my deep appreciation to my advisor, Dr. Liwu Li. His patience and guidance make this work possible.

Also, I am appreciative of Dr. Xiaobu Yuan for being my internal reader. His invaluable suggestions and comments are the great helps to the thesis.

I would specially thank Dr. Xiang Chen for being the external reader from his busy schedules.

Specially, thank Dr. S. Bandyopadhyay for serving as the chair of the committee.
# Table of Contents

ABSTRACT .................................................................................................................. III

ACKNOWLEDGEMENTS ............................................................................................ V

TABLE OF CONTENTS .............................................................................................. VI

LIST OF FIGURES ..................................................................................................... IX

CHAPTER 1 INTRODUCTION ...................................................................................... 1

1.1 OVERVIEW OF THIS THESIS ............................................................................ 1

1.2 ORGANIZATION OF THIS PAPER .................................................................... 3

CHAPTER 2 PARAMETRIC POLYMORPHISM ......................................................... 4

CHAPTER 3 WHY NEED PARAMETRIC POLYMORPHISM ..................................... 6

3.1 COLLECTION CLASSES .................................................................................... 6

3.2 OQL QUERIES .................................................................................................... 7

3.3 ODL CLASSES .................................................................................................. 8

3.4 OQL QUERIES IN THE ODMG OBJECT MODEL .............................................. 9

3.5 OQL QUERIES IN THE JAVA BINDING ............................................................. 12

3.6 OQL QUERIES IN THE C++ BINDING ............................................................ 16

3.7 CONCLUSION ................................................................................................... 19

CHAPTER 4 ADD PARAMETERIZED TYPES TO JAVA ............................................. 20

4.1 WHERE CLAUSES ............................................................................................ 20

4.1.1 Parameterized Definitions ........................................................................... 20

4.1.2 Virtual Machine Extensions ....................................................................... 21

4.1.3 Bytecode Extensions .................................................................................. 22

- vi -
3. ODMG D_DICTIONARY ................................................................. 66
4. ODMG D_LIST ........................................................................... 69
5. ODMG D_SET ........................................................................... 77
REFERENCES ....................................................................................... 86
VITA AUCTORIS .................................................................................... 89
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Universal type quantification</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Bounded parametric polymorphism</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>F-bounded polymorphism</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Instantiation of generic classes</td>
<td>5</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>The top interface</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Collection interface</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Bag interface</td>
<td>7</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Simple query</td>
<td>7</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Complex query</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Query with order by clause</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Class with extent 1</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Class with extent 2</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Syntax of an OQL query</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>OQL queries and the ODMG object model</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Messages</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>OQL and ODL</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>OQL and Java</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.14</td>
<td>Java Interface Collection, Class Professor and Class Course</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.15</td>
<td>Valid OQL query failing a type-check</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3.16</td>
<td>OQL query with type casts 1</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3.17</td>
<td>Type casts of range variables</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3.18</td>
<td>Queries with type casts</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3.19</td>
<td>OQL query with type casts 2</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.20</td>
<td>Parametric interfaces</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.21</td>
<td>Parametric interfaces and inheritance</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.22</td>
<td>Instantiation</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.23</td>
<td>OQL and C++</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.24</td>
<td>Parametric collection</td>
<td>18</td>
</tr>
</tbody>
</table>
Figure 3.25  OQL query with C++ templates
Figure 4.1   Where clause interface definitions
Figure 4.2   The SortedList interface
Figure 4.3   Calling lesthan method
Figure 4.4   A parameterized class hierarchy
Figure 4.5   ODMG d_set using where clause 1
Figure 4.6   ODMG d_set using where clause 2
Figure 4.7   Polymorphism in Pizza
Figure 4.8   Heterogenous translation of polymorphism into Java
Figure 4.9   Homogenous translation of polymorphism into Java
Figure 4.10  Polymorphic arrays in Pizza
Figure 4.11  Homogenous translation of polymorphic arrays into Java
Figure 4.12  ODMG d_Set using Pizza
Figure 4.13  The Java Virtual Machine
Figure 4.14  Original class file format, extensions for parameterized classes
Figure 4.15  Java bytecodes for the insert method
Figure 4.16  ODMG d_set using load_time expansion 1
Figure 4.17  ODMG d_set using load_time expansion 2
Figure 4.18  ODMG d_set using load_time expansion 3
Figure 4.19  ODMG d_set using load_time expansion 4
Figure 4.20  Rough edges of Pizza
Figure 5.1   ODMG Collection Interface
Figure 5.2   Java d_Collection Interface
Figure 5.3   Java d_List Class definition
Figure 5.4   Java d_GeneralList Class definition
Figure 5.5   Java d_List$Models Class definition
Figure 5.6   Java d_List$ModelsSpecial Class definition
Figure 5.7   Methods definition by Class d_List$Models
Chapter 1 Introduction

1.1 Overview of This Thesis

The core of the ODMG (Object Database Management Group) standard is the ODMG object model. This model comes equipped with its associated declarative languages: ODL (Object Definition Language) and OQL (Object Query Language)[7]. Both languages are strongly typed, and are supposed to work well with each other. At a first glance, it appears they do. But a careful analysis reveals some unexpected and serious problems. The ODMG standard also includes bindings for well-known programming languages. At this point such bindings are defined for C++, Java, and Smalltalk [7]. OQL is also meant to work well with these languages. Indeed, the impedance mismatch is supposed to disappear in an object-oriented database management system [4]. One of the goals of the ODMG standard is a unified type system that works uniformly across database and programming languages. OQL is thus expected to avoid the problems of embedded SQL. However, there are still some unforeseen and serious problems in matching OQL with the type systems of the programming language bindings of the ODMG standard. Type-checking enforces the rules of the type system. Static type-checking ensures that a query or a transaction will not fail at runtime due to a type error. Runtime errors are generally more critical in database systems than in programming languages and systems. Indeed, an expensive recovery procedure may have to be invoked just because of a type error. An additional reason in support of static type-checking is that dynamic type-checking reduces runtime efficiency. This becomes particularly significant if a large number of persistent objects must be accessed and, for each one of them, type-checking performed at runtime. Furthermore, if a query fails at runtime due to a type error, all the optimization efforts carried out at compile time is lost. So efficiency and reliability make static type-checking preferable in object-oriented database systems. However, there are situations in which dynamic type-checking is unavoidable. A sophisticated type system offers static type-checking in most situations and isolates the situations in which dynamic type-checking is necessary. This is why the choice of a suitable type system is so critical for the ODMG standard.
Object-oriented languages have dynamic binding of a message (a function call) to a method (a function). This means that selection of a suitable method for executing a message is performed at runtime. Static type-checking of a message guarantees that at runtime there will be at least one suitable method for executing the message. This is a distinguishing characteristic of object-oriented type systems. Static type-checking ensures that runtime errors of the type "message not understood" will never happen.

In this paper, I consider the most important form of OQL queries: select expressions. The first negative result is that OQL queries cannot be type-checked in the type system underlying the ODMG object model and its definition language ODL. This result is hardly a surprise from our viewpoint. Indeed, it is a consequence of a confusion that exists in the type system of the ODMG object model. A surprising and really disturbing result is that OQL queries cannot be type-checked in the type system of the Java binding of the ODMG standard either. This result holds in spite of the fact that the Java type system does not suffer from inconsistencies of the type system of the ODMG object model. This negative result is a consequence of two facts. Typical classes declared in a schema come with extents. An extent has a collection type. The best we can do is to specify a generic collection type in Java [10] with Object as the element type. Object is the root class in Java. It encapsulates only the very basic features applicable to all object types. This leads to a loss of type information and makes type-checking OQL queries impossible. This problem has a solution within the framework of the Java type system, but it is not an attractive one. If explicit type casts of control variables down the inheritance relationships are required in OQL queries, type-checking OQL queries become possible. This solution is not appealing from the user viewpoint. It appears that a user is required to specify redundant type information in a query. At the same time, type checking must necessarily be carried out dynamically, which presumably is contrary to the intent of OQL and ODL. In contradistinction to the situation described above, a positive result that established formally is that type-checking OQL queries presents no problem for a type system with the basic form of parametric polymorphism (universal type quantification). A corollary to this result is that the type system of the C++ binding of the ODMG standard allows static type-checking of OQL queries. This is made
possible because C++ has limited parametric polymorphic facilities supported by
templates [21].
Type-checking OQL queries with an order by clause involves further subtleties if runtime
effects of the type "message not understood" are to be avoided. A further positive result is
that the type system of the C++ binding of the ODMG standard can handle type-checking
of OQL queries with an order by clause. Only dynamic type-checking of queries with an
order by clause is possible in the type system of the Java binding of the ODMG standard.
Queries often require ordered collections both as their result and in the supporting query
evaluation algorithms. These notions are bounded type quantification (constrained
genericity) and F-bounded polymorphism. Java OQL of the ODMG standard combines
features of two strongly and mostly statically typed languages. But this combined
sublanguage admits neither static nor the standard dynamic type-checking.

1.2 Organization of This paper
This paper is organized as follows. Chapter 1 gives a general introduction of parametric
polymorphism. Chapter 2 explains why we need add parametric polymorphism to Java
when we choose Java to bind the ODMG standard. Starting with specification of generic
collection types in ODMG type system, this chapter introduces OQL queries and ODL
classes using illustrative examples. Type-checking OQL queries in the type system
underlying the ODMG object model is considered in Section 4. Type-checking OQL
queries in the type system of the Java binding of the ODMG standard is considered in
Section 5. Type-checking OQL queries in a type system that supports parametric
polymorphism is considered in Section 6, where a positive result about static type-
checking of OQL queries is introduced.
In Chapter 3, I illustrate the existing techniques known as Pizza, Where Clauses and JVM
Load-Time Expansion. Neither is available in the ODMG standard. In Chapter 4, the
comparisons between these techniques are given.
In Chapter 5, I will give our solution on how to add parametric polymorphism in Java to
settle the problems of Java binding of the ODMG standard.
Chapter 2 Parametric Polymorphism

Parametric polymorphism allows developers of object-oriented applications to declare generic classes, which can later be instantiated in order to fulfill specific requirements [15]. Type parameterization has been used in C++, Eiffel, and other languages. The feature is by no means only relevant to object orientation. In this section, I will explain the notions of parametric polymorphism (universal type quantification), bounded parametric polymorphism, and F-bounded polymorphism [20]. The examples that I will use throughout this chapter are based on familiar generic classes representing collections and ordered collections. In Figure2.1, the generic class Collection has one formal type parameter \( T \) that stands for any type (universal type quantification). This simple form of parametric polymorphism is supported by C++ through templates.

Figure2.1 Universal type quantification
public class Collection\(<T>\>
{ ...
  public boolean exists (T aElement);
  public void addElement (T aElement);
  public void removeElement (T aElement);
 }

In a type system that has a root class Object as in Java, the situation of Figure2.1 can be interpreted as if the upper bound of \( T \) were Object. In Figure2.2, on the other hand, class OrdCollection has a formal type parameter \( T \) whose upper bound is Ord (bounded type quantification).

Figure2.2 Bounded parametric polymorphism
public interface Ord{
  public boolean lessThan (Object a0bj);
  ...
}
public class OrdCollection\(<T>\ implements Ord> extends Collection\(<T>\>{
  ...
}

Note that method lessThan in Figure2.2 takes one argument whose compile-time type is Object. Using the type Object often leads to dynamic type checking. A more sophisticated form of bounded parametric polymorphism does not have this problem. In class OrderedCollection, shown in Figure2.3, the upper bound is generically and recursively specified as Ordered\(<T>\). Such a specification of OrderedCollection can be
done in a language that supports F-bounded polymorphism [20]. The specification simply means that instances of OrderedCollection contain only elements that can be compared with elements of the same type.

Figure 2.3 F-bounded Polymorphism

```java
public interface Ordered<T> {
    ...
    public boolean lessThan (T aobject);
}
public class OrderedCollection<T implements Ordered<T>>
    extends Collection<T>{
    ...
}
```

These more sophisticated notions of parametric polymorphism are obviously justified by some very pragmatic situations. However, they come with a price. Bounded (and thus F-bounded) type quantification may lead to undecidable type checking. Generic classes have to be instantiated before they are used. Figure 2.4 shows how instantiations are carried out. OrderedCollection<Employee> is a class instantiated from a generic class OrderedCollection. In this case, Employee is an actual type parameter in the instantiation of OrderedCollection.

Figure 2.4 Instantiation of generic classes

```java
public class Employee extends Person
    implements Ordered<Employee> {
    ...
    public Employee (String aName, float asalary);
    public String getName0;
    public float getSalary0;
    public boolean lessThan (Employee aEmployee);
}
OrderedCollection<Employee> empCollec = new OrderedCollection<Employee>();
empCollec.addElement (new Employee ("Jones", 50000.0));
Employee anEmployee = empCollec.getFirstElement0; // No type cast
```

Several proposals exist for extending Java with parametric polymorphism. However, all of these proposals have a common problem: they overlook the need to make the implementation techniques for parametric polymorphism compatible with Java Core Reflection [24].
Chapter 3 Why Need Parametric Polymorphism

There are several negative results concerning about the type-checking when doing queries in the existing standard object-oriented database. Two of them are serious and really disturbing. The first one is that OQL queries cannot be type-checked in the type system underlying the ODMG object model and its definition language ODL. The second one is that the OQL queries cannot be type-checked in the type system of the Java binding of the ODMG standard.

Type-checking OQL queries presents no problem for a type system with the basic form of parametric polymorphism (universal type quantification). A corollary to this result is that the type system of the Java binding of the ODMG standard allows static type-checking of OQL queries if we add parametric polymorphism facilities to Java.

In order to explain these, first of all, I will introduce the following concepts used in the ODMG Standard.

3.1 Collection Classes

The type system of the ODMG object model is equipped with the root object type. This object type is called Object, and its interface is given below. The root object type encapsulates methods applicable to all objects, regardless of their type. All other object types extend Object by inheriting the methods of Object and introducing additional ones.

*Figure 3.1 The top interface*

```java
interface Object {
    boolean same_as(in Object anObject);
    Object copy();
    void delete();
};
```

The ODMG object model has both interfaces and classes in its 3.0 release. Unlike Java, the ODMG standard is not always careful about the distinction between interfaces and classes. In the ODMG object model the interface of the *Collection* type is specified as follows:

*Figure 3.2 Collection interface*

```java
interface Collection: Object {
    unsigned long cardinality();
    boolean is_empty();
    void insert_element(in any element);
    void remove_element(in any element);
};
```
boolean contains_element(in any element);
Iterator create_iterator();
);

The above interface is a good illustration of an awkward way of expressing parametric polymorphism in the ODMG object model. In the above specification, \textit{any} stands for any type. The position of \textit{any} in the type system of the ODMG object model is unclear. This creates major difficulties in type checking. \textit{any} is used in the ODMG object model where a type parameter would appear in a type system that supports parametric polymorphism. However, the ODMG object model does not support parametric polymorphism. In a type system with parametric polymorphism, the type parameter is replaced by a specific type in order to get a specific type from a parametric one. In the ODMG object model, \textit{any} is not a type parameter.

Other generic collection types supported by the ODMG object model and its language bindings are \textit{Set}, \textit{Bag}, \textit{List}, and \textit{Array}. These collection types are derived by inheritance from \textit{Collection} in a straightforward manner. Only the simplest one of those interfaces is given below as it appears in the ODMG object model.

\textbf{Figure3.3} Bag interface
interface Bag: Collection{
    unsigned long occurrence_of(in any element);
    Bag create_union(in Bag other_bag);
    Bag create_intersection(in Bag other_bag);
    Bag create_difference(in Bag other_bag);
};

As explained earlier, the type \textit{any} in the ODMG object model is used where a type parameter is in fact required.

\section*{3.2 OQL Queries}
A simple and self-explanatory OQL query is given below.

\textbf{Figure3.4. Simple query}
select well_paid(emp: x.name, sal: x.salary())
from employees as x
where x.salary()>50,000

The result of the above query is a \textit{bag} of objects representing well-paid employees. \textit{well\_paid} is the name of an already defined type. The expression \textit{well\_paid (emp: x.name, sal: x.salary())} constructs an object of the \textit{well\_paid} type. Objects of the \textit{well\_paid} type have two attributes named \textit{emp} and \textit{sal}. 

- 7 -
In order to get a set of objects, i.e., to eliminate duplicates from the result, the select distinct option must be used. The above query does not involve traversal of the complex structure of an employee object, and because of that it looks just like an SQL query. Let us now consider a complex query that produces objects representing well-paid employees from large departments:

\begin{verbatim}
Figure 3.5 Complex query
select well_paid (emp: x.name, sal: x.salary())
from employees as x
where x.salary() . 50,000
and x.dept.NoOfEmployees() > 100
\end{verbatim}

There are of course many other options for OQL queries that I do not consider. But it turns out that from the viewpoint of type checking, queries with order by clause involve further subtleties. An example of such a query follows:

\begin{verbatim}
Figure 3.6 Query with order by clause
select well_paid (emp: x.name, sal: x.salary())
from employees as x
where x.salary() . 50,000
and x.dept.NoOfEmployees() . 100
order by emp
\end{verbatim}

Note that the result of the above query is a list of objects, i.e., an ordered collection.

3.3 ODL Classes

The notion of an interface plays a crucial role in the ODMG object model. An interface is an abstract object type. It consists of a collection of signatures of methods applicable to instances (objects) of that type. A class is also an object type. In addition to methods applicable to its objects, a class is equipped with the specification of the (abstract) components of the object state. These components are called attributes. Unlike an interface, a class can be instantiated to create objects of that class. Instances of an object type defined by an interface can be created only if there is a class that implements that interface. Specification of an object type requires specification of signatures of methods of that type. A method signature consists of the method name, the names and the types of arguments, and the type of the result, which may be void. A collection of method signatures of an object type (a method suit) is given as a structure (record). If an object type is a class, then the structure also contains specification of attributes (their names and
types). In the ODMG standard, ODL classes do not contain procedural representation of methods. The reason is that ODL is a declarative language. Methods are implemented in procedural object-oriented languages for which the ODMG standard specifies the required bindings (C++, Small-talk, and Java). A class in ODL may have an extent. The *extent* of a class is the collection of all the created objects of that class. Object-oriented languages do not support class extents; database systems do. This support includes the abilities to cluster class extents, build access paths (indices) on top of them, etc. Typical queries refer to class extents.

Now consider a familiar example: a class *Employee* with attributes *name, id* and *dept* and methods *salary, hire* and *fire*. The fact that the type of the attribute *dept* is *Department* makes an employee object complex. The *Employee* class has an extent *employee*. This is the set of all objects of the class *Employee* maintained by the object-oriented database system.

*Figure 3.7 Class with extent 1*

class Employee (extent employees){
    attribute string name;
    attribute short id;
    attribute Department dept;
    float salary();
    void hire(in Department d);
    void fire();
};

The class *Employee* above refers to the class *Department*. Specification of the class *Department* is given below. A department object has attributes *name, id* and a method *NoOfEmployees*. The class *Department* has an extent called *departments.*

*Figure 3.8 Class with extent 2*

class Department (extent departments){
    attribute string name;
    attribute short id;
    short NoOfEmployees();
};

### 3.4 OQL Queries in the ODMG Object Model

Let's first consider type checking of OQL queries of the following general form:

*Figure 3.9 Syntax of an OQL query*

    select projection
    from $e_1$ as $x_1$, $e_2$ as $x_2$, ..., $e_n$ as $x_n$
    where $e$
In the above query, the qualification expression $e$ is required to be of type \textit{boolean}. Let's consider the most general case in which each $e_i$ for $i=1, 2, \ldots n$ is an expression of a collection type. The type of a range variable $x_i$ is obviously intended to be the same as the element type of the collection $e_i$ for $i =1, 2, \ldots n$. The result of this query is a bag. The type of \textit{projection} determines the type of elements of this resulting bag. If \textit{select distinct} option is used, the result is a set. In type checking a query expression of the above form, the types of range variables $x_i$ must be inferred first. This is done by inspecting the \textit{from} clause from left to right. The expression $e_i$ is checked to be of a collection type $i =1, 2, \ldots n$. The type of the range variable $x_i$ is then inferred to be the same as the type of elements of the collection type of $e_i$. Due to the limitations of the type systems, the type of $x_i$ determined this way is not specific in either the ODMG object model or the Java programming language. When the type of the expression $e_i$ is being inferred in this process, the types of the (collection) expressions $e_j$ are already determined, and so is the type of the range variables $x_j$. With the limited type information for the range variables $x_i$ inferred from the \textit{from} clause, type checking must determine that the type of the qualification expression $e$ is \textit{boolean}. In addition, the type of the expression \textit{projection} must be inferred. The overall type of a query expression of the above form is a suitable collection type. If the type of \textit{projection} is $C$, then $C$ should obviously be the type of elements of the collection representing the query result. In determining the type of this collection, the limitations of the type systems of the ODMG object model and the Java programming language again come into play. In both type systems the type of elements of the resulting collection is not specific. Thus in the absence of a \textit{select distinct} option, the type of the above query expression is \textit{Bag}, otherwise it is \textit{Set}. Neither type specifies the element type.

Now I will illustrate the first negative result.

Let's reconsider Figure 3.10, which may actually be used as a counterexample for the illustration that follows.

\textbf{Figure 3.10 OQL queries and the ODMG object model}

\texttt{select well_paid (emp: x.name, sal: x.salary())}

\texttt{from employees as x}

\texttt{where x.salary()>50,000}
In the type system of the ODMG object model, the type of the extent employees is Collection. In this type system it is not possible to derive a type Employee_Collection from the type Collection by changing the inherited method signatures. Such changes are not allowed for reasons of type safety. The same limitation applies to the Java and the C++ type systems. Since the element type of Collection is in the ODMG object model any, the type of the range variable x is inferred to be any. The methods of the type any are not defined in the ODMG standard. This is why it is not possible to deduce whether x. name and x. salary () is type correct or not.

(OQL queries in the ODMG object model). OQL queries cannot be type-checked in the ODMG object model.

In order to type-check a query

```
select projection
from e_1 as x_1, e_2 as x_2... e_n as x_n
where e
```

The types of range variables x_i must be determined. Indeed, these range variables in general appear both in the projection expression and in the qualification expression e. In order to determine the type of a range variable x_n, the type of the expression e_i must be determined. The best we can do is to determine that the type of e_i is Collection. According to the ODMG object model, we can then deduce that the type of x_i is any. With this type information about range variables x_i, we have to type-check the expressions projection and e. The required proof rule has the following form:

Consider a message x_i. M (a_i1, a_i2...a_in) either in the projection or in the qualification of an OQL query. Take a look at the following formal specification:

**Figure 3.11 Messages**

\[ \phi \text{ schema, interface } CR_C \in \phi, \]
\[ \phi \vdash R_C \text{ extends } \{C_m(C_1 x_1, C_2 x_2, ..., C_n x_n)\}; \]
\[ \phi \vdash o : C_o, \quad \phi \vdash C_0 \leq C, \]
\[ \phi \vdash a_i : C_{a_i}, \quad \phi \vdash C_{a_i} \leq C_i, \quad 1 \leq i \leq n, \]
\[ \phi \vdash \text{body}_m <x_1/a_1, x_2/a_2, ... x_n/a_n> : C_m \]

\[ \phi \vdash o.m(a_1, a_2 ... a_n) : C_m \]

- 11 -
for type-checking messages, we have to prove:

1. \( \varphi \vdash R_{\text{any}} \text{ extends } \{ C_{i1} \ y_{i1}, C_{i2} \ y_{i2}, \ldots C_{in} \ y_{in} \} \)

2. If \( \varphi \vdash a_{ij} : C_{aij} \) then \( \varphi \vdash C_{aij} \leq C_{ij} \).

But in the ODMG object model, it is not possible to deduce (1) above. Indeed, the model does not specify method signatures in \( R_{\text{any}} \). If we interpret the ODMG object model in such a way that (1) always holds, then in the message \( x_i.M(a_{i1}, a_{i2}, \ldots, a_{in}) \), the type of

\[
\varphi \text{ schema, } \varphi \vdash e_1 : \text{Collection}, \\
\varphi \cup U_{i=1}^{k} \{ \text{any } x_i \} \vdash e_{k+1} : \text{Collection, } 1 \leq k < n, \\
\varphi \cup U_{i=1}^{n} \{ \text{any } x_i \} \vdash e : \text{boolean}, \\
\varphi \cup U_{i=1}^{n} \{ \text{any } x_i \} \vdash \text{projection : } C,
\]

\( \varphi \vdash \text{select projection from } e_1 \text{ as } x_1, \ldots, e_n \text{ as } x_n \) where \( e : \text{Bag} \)

the receiver \( x_i \) is always correct. This obviously leads to runtime type errors of the type "message not understood," in spite of a successful static type check. Figure 3.12 shows that it is not possible to define the type any in a way that allows type-checking OQL queries.

### 3.5 OQL Queries in the Java Binding

The second negative result is that OQL queries cannot be type-checked in the type system of the Java binding of the ODMG standard.

(OQL queries in the Java binding). Static type-checking OQL queries are not possible in the type system of the Java binding of the ODMG standard.

The main difference, in comparison to the previous illustration, is that in the Java type system we can assert that the type of a range variable \( x_i \) is Object. An attempt to formulate a rule for type-checking OQL queries in the type system of the Java binding of the ODMG standard leads to the following:
Figure 3.13 OQL and Java

\( \varphi \text{ schema, } \varphi \models e_1 : \text{Collection,} \)
\( \varphi \cup U^k_{i=1} \{ \text{Object } x_i \} \models e_{k+1} : \text{Collection, } 1 \leq k < n, \)
\( \varphi \cup U^n_{i=1} \{ \text{Object } x_i \} \models e : \text{boolean,} \)
\( \varphi \cup U^n_{i=1} \{ \text{Object } x_i \} \models \text{projection : C,} \)

\( \varphi \models \text{select projection from } e_1 \text{ as } x_1, \ldots, e_n \text{ as } x_n \)
\( \text{where } e : \text{Bag} \)

Now consider a message \( x_i.M(ai_1, ai_2, \ldots ai_n) \) either in the projection or in the qualification of an OQL query where \( m \neq \text{same_as, } m \neq \text{copy and } m \neq \text{delete} \). The type-check fails, since the type system only allows the conclusion that the type of \( x_i \) is Object.

In other words, all queries referring to classes other than Object fail a static type check, irrespectively of whether they are type correct or not.

The Java collection interface is given below. Note that in order to provide a generic specification of the type Collection in Java, the elements of a generic collection must be of type Object:

Figure 3.14 Java Interface Collection, Class Professor and Class Course definition

interface Collection {
    int size();
    boolean isEmpty();
    void add(in Object element);
    boolean remove(in Object element);
    boolean contains(in Object element);
    Iterator iterator();
    ...
}

Consider the following Java collections:

class Professor {
    ...
    public float salary(){...}
    public Collection courses(){...}
    ...
}
class Course {
    ...
    public int enrollment(){...}
    ...
}

Collection professors;
Collection courses;

Note that it does not help if we derive classes EmployeeCollection and PersonCollection from the generic class Collection. The signatures of the methods inherited from
Collection would have to be the same in EmployeeCollection and in PersonCollection. This is a property of the Java type system, but also of the ODMG object model and C++. An example of a valid OQL query on the above Java collections fails type-checking follows.

Figure 3.15 Valid OQL query failing a type-check
select x
from professors as x, x.courses() as y
where x.salary() > 70,000
and y.enrollment() > 50

The messages x.salary() fails the type-checking because in the Java type system x is of type Object. The message x.courses() also fails the type-checking. Even if the type of x.courses() could be determined at this point, type-checking the message y.enrollment() would still not succeed, because the type of x.courses() would be Collection, and the type of y would be Object. From our viewpoint, the above result is unexpected and very disturbing. Perhaps it is also obvious, but the fact is that it has not been observed so far.

This result shows that Java and OQL cannot work well together in a unified type system. Such a unified type system is precisely one of the goals of the ODMG standard. This negative result also raises a fundamental question: Is Java, as it is, well suited as a database programming language? Before trying to answer this question, let us consider a possible fix, as indeed there is one. OQL could be changed to require explicit type casts of range variables down the inheritance relationships. The general form of such a query is now:

Figure 3.16 OQL query with type casts
select projection
from e1 as (C1)x1, e2 as (C2)x2, ..., en as (Cn)x_n
where e

There are two problems with the above fix. The first one is that the user is required to specify apparently redundant type information for range variables. The second negative of the above solution is that type casts require dynamic type checking. This is surely not the intent of the authors of the ODMG standard. In order to see how awkward this situation is, one has to keep in mind that optimization is intended to be carried out at compile time and still a query fails at runtime due to a type error!
(Type casts). If explicit type casts are used for range variables, type-checking OQL queries in the type system of the Java binding of the ODMG standard becomes possible, but only at the expense of dynamic checks.

Although the type of $e_i$ is still $Collection$, explicit type casts in the bindings $e_i$ as $(C_i)$, $x_i$ of a query "select projection from $e_1$ as $(C_1)x_1$, $e_2$ as $(C_2)x_2$, ..., $e_n$ as $(C_n)x_n$ where $e$" now allows the type system to assign a specific type $C_i$ to the range variable $x_i$. This step is expressed by the following deduction:

**Figure 3.17 Types casts of range variables**

\[ \varphi \text{ schema, } \varphi \vdash e : \text{Collection, } \varphi \vdash C \leq \text{Object} \]

\[ \varphi \cup \{ e \text{ as } (C)x \} \vdash x : C \]

The main difference from the above illustration is that the types of range variable are now specific, rather than just being $Object$. This fact is expressed in the following rule.

**Figure 3.18 Queries with type casts**

\[ \varphi \text{ schema, } \varphi \vdash e_1 : \text{Collection,} \]

\[ \varphi \cup \bigcup_{i=1}^{k} \{ C_i x_i \} \vdash e_{k+1} : \text{Collection, } 1 \leq k < n, \]

\[ \varphi \vdash C_i \leq \text{Object}, \quad 1 \leq i \leq n, \]

\[ \varphi \cup \bigcup_{i=1}^{n} \{ C_i x_i \} \vdash e : \text{boolean,} \]

\[ \varphi \cup \bigcup_{i=1}^{n} \{ C_i x_i \} \vdash \text{projection : C}, \]

\[ \varphi \vdash \text{select projection from } e_1 \text{ as } (C_1) x_1, \ldots, e_n \text{ as } (C_n) x_n \]

where $e : \text{Bag}$.

Consider a message $x_i.M(ai_1, ai_2, \ldots, ai_n)$. With $x_i:C_i$, type-checking requires the following deductions:

1. $\varphi \vdash \text{R}_c \text{ extends } \{ C_{i m} (C_{i 1} y_{i 1}, C_{i 2} y_{i 2}, \ldots, C_{i n} y_{i n}) \}, \quad \varphi \vdash C_i \leq C$

2. If $\varphi \vdash a_{ij} : C_{ij}$ then $\varphi \vdash C_{aij} \leq C_{ij}$.

If the above deductions are possible, the query type checks. If not, it fails, as it should. But if the type checking succeeds, a dynamic type-check is still generated because of the type casts. The type of $e_i$ is inferred to be $Collection$. Without a type cast, the type of $x_i$ is inferred to be $Object$. The type checker has no basis for inferring more specific type information. The type cast is thus an assertion that the type of $x_i$ is $C_i$. The type checker
cannot verify this assertion at compile time. This is why a runtime test (dynamic check) is
generated. The query type-checks only because of this dynamic check. The type casts of
the above form, and the associated dynamic type-checks, are very typical for the Java
programming language. In many situations the reason is precisely the fact that Java lacks
parametric polymorphism.

Suppose we are given the following two Java collections:

**Figure 3.19 OQL query with type casts 2**
Collection professors;
Collection courses;
Type checking of the query given below succeeds because of the usage of explicit type casts.

```plaintext
select x
from professors as (Professor)x,
x.courses() as (Course)y
where x.salary() > 70,000
and y.enrollment() > 50
```

But the run-time checks must be generated.

### 3.6 OQL Queries in the C++ Binding

The type system of the C++ binding allows parametric (generic) classes [21]. In order to
introduce formally what this means for type-checking OQL queries, we have to extend
the formal system developed so far with rules for parametric classes. A rule for adding a
parametric interface to a schema is given below:

**Figure 3.20 Parametric interfaces**

\[ \varphi \text{ schema}, C \not\in \text{dom}(\varphi), \]
\[ \varphi, T \leq \text{Object}, C<T> \leq \text{Object} \vdash R \text{ extends RObject} \]

\[ \varphi \cup \{ \text{interface C<T> R} \} \text{ schema} \]

There are two subtleties in checking whether the method suits \( R \) of a parametric class
\( C<T> \) is well formed (\( R \text{ extends RObject} \)). The first one is that \( R \) contains occurrences of
the type parameter \( T \), illustrated by the parametric Collection \( <T> \) class. Because of this,
we must assume that \( T \) stands for a valid class (\( T<\text{Object} \)). Likewise, occurrences of
\( C<T> \) may appear in \( R \), as in the Bag \( <T> \) class. We must assume that \( C<T> \) stands for a
valid class, i.e \( C<T> <\text{Object} \). The rule is recursive, just as the rule for type-checking
interfaces that do not have type parameters.

- 16 -
Adding a parametric interface $C_2$ to a schema by deriving $C_2$ from another parametric interface $C_1$ already available as follows:

$$
\varphi \text{ schema, } \text{interface } C_1 \langle T \rangle \quad R_1 \in \varphi,
C_2 \notin \text{dom}(\varphi), \quad \varphi, \space C_2 \langle T \rangle \leq \text{Object } \mid - \quad R_2 \text{ extends } R_1
$$

$$
\varphi \cup \{\text{interface } C_2 \langle T \rangle : C_1 \langle T \rangle \text{ R}_2 \} \text{ schema}
$$

Instantiating a parametric interface $C \langle T \rangle$ with a specific object type $A$ is governed by the following rule:

$$
\varphi \text{ schema, } \varphi \mid - A \leq \text{Object, } \text{interface } C \langle T \rangle \quad R \in \varphi
$$

$$
\text{interface } C \langle A \rangle \quad R \langle T/A \rangle \in \varphi
$$

In the above rule, $R \langle T/A \rangle$ stands for the result of substituting $A$ for $T$ in $R$. The above rule states that if we have an interface $C \langle T \rangle$ in the schema, then effectively we also have an interface $C \langle A \rangle$ in the schema for each existing interface $A$ in the schema. There are some obvious additional requirements for avoiding name clashes that may be caused by this substitution. We are now equipped to formally illustrate the following positive result:

*(OQL queries and parametric polymorphism). A type system that supports parametric polymorphism restricted to the universal type quantification allows static type-checking of OQL queries.*

This positive result is a consequence of the fact that this type system allows compile-time specification of the type of elements of a collection type. This then allows deduction of the specific types of range variables in an OQL query from the bindings of the form $e_i$ as $x_i$. The type of the collection expression $e_i$ is now $\text{Collection } \langle C \rangle$. The type of elements of this collection type is now a specific type $C_i$, where $C_i < \text{Object}$. Because of this, the type of the range variable $x_i$ is a specific type $C_i$. A message $x_i.M(a_1, a_2, \ldots, a_n)$ will now type-check if $C_i$ is equipped with a method $m$ with the appropriate signature, otherwise it will fail. Let's look at the following expression:
Figure 3.23 OQL and C++

\[ \phi \text{ schema, } \phi \models e_1 : \text{Collection}\langle C_1 \rangle, \]
\[ \phi \cup U^k_{i=1}\{C_i \times_i \} \models e_{k+1} : \text{Collection}\langle C_{k+1} \rangle, \ 1 \leq k < n, \]
\[ \phi \cup U^n_{i=1}\{C_i \times_i \} \models e : \text{boolean}, \]
\[ \phi \cup U^n_{i=1}\{C_i \times_i \} \models \text{projection : } C, \]

\[ \phi \models \text{select projection from } e_1 \text{ as } x_1, \ldots, e_n \text{ as } x_n \]
\[ \text{where } e : \text{Bag} \langle C \rangle \]

Note that the type of the result of a query is now also precisely determined. If select distinct is used, the type of the result is Set \langle C \rangle.

With proper usage of parametric polymorphism, the collections from above Example now have the following form:

**Figure 3.24 Parametric collection**

interface Collection \langle T \rangle. { 
    unsigned long cardinality(); 
    boolean is_empty(); 
    void insert_element(T element); 
    void remove_element(T element); 
    boolean contains_element(T element); 
    Iterator \langle T \rangle create_iterator(); 
};

(Parametric Bag interface)
interface Bag \langle T \rangle: Collection \langle T \rangle { 
    unsigned long occurrence_of(T element); 
    Bag \langle T \rangle create_union(Bag \langle T \rangle other_bag); 
    Bag \langle T \rangle create_intersection(Bag \langle T \rangle other_bag); 
    Bag \langle T \rangle create_difference(Bag \langle T \rangle other_bag); 
};

**Figure 3.25 OQL query with C++ templates**

Collection \langle Professor \rangle : professors;
Collection \langle Course \rangle : courses;

The query from the above example now type-checks.

select x
from professors as x, x.courses() as y
where x.salary() \geq 70,000
and y.enrollment() \geq 50

The type of x is now inferred to be Professor, and the type of y is inferred to be Course.

Type checking succeeds. It is performed entirely at compile time.
3.7 Conclusion

While the current Java language does not provide any form of type parameterization, it does include a supertype `Object` of all object types. This allows some classes that could be written in other languages using parametric polymorphism to be expressed using `Object` in place of a parameter type. However, there are several disadvantages of this substitute for type parameterization. First, the programmer must insert type casts to change an expression with static type `Object` to a more specific type. This can be an annoyance and, since Java type casts are checked at run time, it also leads to some decrease in execution speed. The use of superclass `Object` becomes more cumbersome when binary operations (such as an ordering) are needed on objects of the parameter type, since typing considerations become more complex. Finally, use of a fixed type `Object` does not allow us to capture certain generic concepts that could be expressed as parameterized classes or interfaces. For all of these reasons, I believe that adding type parameterization to the Java language will be unavoidable if we choose Java binding ODMG standard.
Chapter 4 Add Parameterized Types to Java

If we want to use Java to bind ODMG object models, we should extend Java in one area where more power is needed: support for parametric polymorphism, which allows the definition and implementation of genetic abstractions.

Up to now, there are three ways that can be used to make Java support parametric polymorphism.

4.1 Where Clauses

4.1.1 Parameterized Definitions

In the extended Java specification, interface and class definitions can be parameterized, allowing them to define a group of related types that have similar behavior but which differ in the types of objects they manipulate [11]. All parameters are types; the definition indicates the number of parameters, and provides formal names for them. For example, the interface

    interface SortedList [T] ...

might define SortedList type, which differ in the type of element stored in the list (e.g., "SortedList[int]" stores ints, while "SortedList[String]" stores strings). As a second example,

    interface Map [Key, Value] ...

defines Map type, such as "Map[String, SortedList[int]]", that map keys to values. In general, a parameterized definition places certain requirements on its parameters. For example, a "SortedList" must be able to sort its elements; this means that the actual element type must provide an ordering on its elements. Similarly a "Map" must be able to compare keys to see if they are equal. The parameterized definition states such requirements explicitly by giving where clauses, which state the signatures of methods and constructors that objects of the actual parameter type must support. Figure 4.1 shows these two interfaces with their where clauses.

*Figure 4.1 Where clause Interface definitions*

interface SortedList [T]
    where T { boolean lt (T t); } ... { }
interface Map [Key, Value]
    where Key { boolean equals (Key k); } ... { }

In the definition of "SortedList", the where clause indicates that a legal actual parameter
must have a method named "lt" (less than) that takes a "T" argument and returns a "boolean". The second definition states that a legal actual parameter for "Key" must have
a method named "equals" that takes a "Key" as an argument and returns a "boolean". Note that "Map" does not impose any constraints on the "Value" type, and therefore any
type can be used for that parameter.

The body of a parameterized definition uses the type parameters to stand for the actual
types that will be provided when the definitions are used. For example, Figure 4.2 gives
the SortedList interface. Note the use of the parameter "T" to stand for the element type
in the headers of the methods. Thus the "insert" method takes in an argument of type "T",
and the "first" method returns a result of type "T".

Figure 4.2 The SortedList interface

interface SortedList[T]
    where T {boolean lt (T t);}
    { //overview: A "SortedList" is a mutable, ordered sequence.
        //Ordering is determined by the "lt" method of the element type.
        void insert (T x);
        //modifies: this
        //effects: adds x to this
        T first () throws empty;
        //effects: if this is empty, throws empty,
        //else returns the smallest element of this
        void rest () throws empty;
        //modifies: this
        //effects: if this is empty, throws empty,
        //else removes the smallest element of this
        boolean empty ( );
        //effects: if this is empty, returns true,
        //else returns false.
    }

4.1.2 Virtual Machine Extensions

In this section, I discuss the implementation issues that arise in adding parameterized
types to Java and focus on what is new: extensions to the bytecodes of the Java Virtual
Machine (JVM) that support parametric polymorphism, and the effect of these extensions
on both the bytecode verifier and interpreter.
The Java compiler generates "class" files, containing code in a bytecode format, along with other information needed to interpret and verify the code [12]. The format of a "class" file is described by the JVM specification. The extended virtual machine supports not only the extended Java described above but also could be used as target architecture for other languages with subtyping or parametric polymorphism.

The extended bytecode interpreter has been implemented, showing that parametric polymorphism can be added to Java while preserving verification and speeding up some code. Code that does not use parametric polymorphism suffers little performance penalty.

4.1.3 Bytecode Extensions

The most visible change to the virtual machine is the addition of two new opcodes (named invokewhere and invokevirtualwhere) that support invocation of the methods that correspond to the where clauses. They provide invocation of normal and static methods respectively. Constructors are treated as normal methods. For example, the expression “where T { boolean lT (T t);}” in Figure 4.2 is implemented using invokewhere, as shown in Figure 4.3

*Figure 4.3 Calling lessthan method*

```java
aload_2
getfield <Field SortedList[T] #0;>
aload_3
invokewhere <where lT (#0;) Z>  //call lessthan method
```

Other minor changes were made to the format of a "class" file, which supply the bytecode verifier with enough added information to directly verify parameterized code. The encoding of type signatures used to capture instantiation and formal parameter types, and added information to describe the parameters and where clauses of the class. Figure 4.3 shows one example of the extended type signatures: the signature for "less than" includes a "#0", which represents the first formal parameter type (T t.) of the current class. (The Z indicates that the return type of "lt" is Boolean.)

4.1.4 Verifier Extensions

The JVM bytecodes are an instruction set that can be statically type checked. For example, bytecodes that invoke methods explicitly declare the expected argument and
result types. The state in the execution model is stored in a set of local variables and on a stack. The types of each storage location can be determined by straightforward dataflow analysis that infers types for each stack entry and each local variable slot, at each point in the program.

It is important that extensions for parameterized types do not remove the ability of static type-checking bytecodes. The standard Java bytecode verifier works by verifying one class at a time. A call to a method of another class is checked using a declaration of the signature of that method, which is inserted in the “.class” file by the compiler. When the other class is loaded dynamically, these declarations are checked to ensure that they match the actual class signature.

The extensions to the JVM preserve this efficient model of verification. The code of a parameterized or non-parameterized class is verified only once, in isolation from other classes, thus verifying it for all legal instantiations of the code. An instantiation of a class or interface can be checked for legality by examining only signature information in the “.class” file for the parameterized class or interface; examining the bytecodes in the “.class” file is unnecessary.

Both the compiler and the verifier perform similar type checking, treating formal parameter types as ordinary types with a limited set of allowed operations. The important difference between the compiler and the verifier is that compiler variables have declared types, but the verifier must infer their types. The verifier must assign types to stack locations and local variable slots, which are specific enough that an instruction can be type checked (e.g., if it invokes a method, the object must have that method). The assigned types must also be general enough that they include all possible results of the preceding instructions. For each instruction, the verifier records a type with this property, if possible. It uses standard iterative dataflow analysis techniques either to assign types to all stack locations for all instructions, or to detect that the program does not type-check.

Because the bytecodes include branch instructions, different instructions may precede the execution of a particular instruction X. For type safety, the possible types of values placed on the stack by the preceding instructions must all be subtypes of the types expected by X. The core of the verifier performs this operation; it is a procedure to merge a set of types, producing the most specific supertype, or least upper bound in the type
hierarchy. The dataflow analysis propagates this common supertype through X and sends its results to the succeeding instruction(s). The analysis terminates when the types of all stack locations and local variable slots are stably assigned.

\[ \text{Diagram: Object at top, extending downward to A[T], B[U], C[K,V], D.} \]

\[ \text{A[T] extends Object} \]
\[ \text{B[U] extends A[U]} \]
\[ \text{C[K,V] extends A[K]} \]
\[ \text{D extends A[B[int]]} \]

*Figure 4.4 A parameterized class hierarchy*

The primary change to the verifier for parameterized types is a modification to this merge procedure. To find the lowest common class in the hierarchy, we walk up the hierarchy from all the classes to be merged [9, 14]. Each time that a link is traversed to a superclass, we apply the parameter substitution that is described by the extension clause of the class definition. When a common class is reached in the hierarchy, the actual parameters of the common class must be equal.

Consider the class and interface hierarchy shown in Figure 4.4, with the corresponding extends clauses. The union of B[X] and C[X,Y] can be conservatively approximated by successively moving up the tree to find a common node while substituting parameters.

The result is as follows:

\[ (B[X] \cup C[X,Y]) = (A[X] \cup A[X]) = A[X] \]

So these two types are merged to produce A[X]. Similarly, for B[X] and C[Y,X] we have

\[ (B[X] \cup C[Y,X]) = (A[X] \cup A[Y]) = \text{Object} \]

In this case, the merge result is Object, since the parameters did not match for A. Note that unlike in the non-parameterized verifier, the lowest common superclass node is not always sufficient for the merge result, since it may be instantiated differently by the
merged types. Finally, consider merging $B[B\text{[int]]}$ and $D$, which demonstrates that parameterized and non-parameterized classes can be merged:

$$ (B[B\text{[int]] \text{ union } D) = A[B\text{[int]]} $$

### 4.1.5 ODMG Examples

I will give another example to show how to use where clauses to help Java bind the ODMG standard.

Here we define a `d_Set` class. In Java it is possible to define a new type such as a set of integers. However it is not possible to capture a set of abstraction where the elements of a particular set are homogeneous and the element type can differ from one to another. If using "where clause" to extend Java with parametric polymorphism, we can do that.

**Figure 4.5 ODMG d_set using where clause 1**

```java
class d_Set<T>  
    where T{extends otherclass}  
        implements Collection{  
            d_Set[T] Value;  
            d_Set[T] d_create_union(d_Set[T] otherSet){};  
            d_Set[T] d_create_intersection(d_Set[T] otherSet){};  
            d_Set[T] d_create_difference(d_Set[T] otherSet){};  
            boolean d_is_superset_of(d_Set[T] otherSet){};  
            boolean d_is_proper_superset_of(d_Set[T] otherSet){};  
            boolean d_is_subset_of(d_Set[T] otherSet){};  
            boolean d_is_proper_subset_of(d_Set[T] otherSet){};  
            boolean d_is_empty(){};  
            boolean d_is_ordered(){};  
            boolean d_allows_duplicates(){};  
            boolean d_contains_element(Object element){};  
            void d_insert_element(Object element){};  
            void d_remove_element(Object element) throws ElementNotFound{};
        }
```

Actually we can define any type that we need, for example if we need to define a set of Employees we can do like this way:

**Figure 4.6 ODMG d_set using where clause 2**

```java
class d_Set[T]  
    where {T extends Employee}  
        implements Collection{  
            d_Set[T] Value;  
            d_Set[T] d_create_union(d_Set[T] otherSet){};  
            d_Set[T] d_create_intersection(d_Set[T] otherSet){};  
            d_Set[T] d_create_difference(d_Set[T] otherSet){};  
            boolean d_is_superset_of(d_Set[T] otherSet){};  
            boolean d_is_proper_superset_of(d_Set[T] otherSet){};  
            boolean d_is_subset_of(d_Set[T] otherSet){};  
            boolean d_is_proper_subset_of(d_Set[T] otherSet){};  
            boolean d_is_empty(){};
```

- 25 -
boolean d_is_ordered(){
boolean d_allows_duplicates(){
boolean d_containsElement(Object element){
void d_insert_element(Object element){
void d_remove_element(Object element) throws ElementNotFound{)

4.2 PIZZA into Java

4.2.1 Introduction

Pizza [19] is a strict superset of Java that extends Java with parametric polymorphism, high-order functions and algebraic data types. It is defined by translation into Java and compiles into the Java Virtual Machine. Pizza fits smoothly to Java, with some rough edges. There are two main methods for translation: heterogeneous and homogenous. Heterogeneous translation will produce a specialized copy of code with a universal representation. Typically, the heterogeneous translation yields code that runs faster, while homogenous translation yields code that is more compacts. In this paper, I just focus on the Pizza's extending Java with parametric polymorphism.

4.2.2 Parametric Polymorphism Extension

A set of integers is much the same as a set of characters; sorting strings is much the same as sorting floats. Polymorphism provides a general approach to describe data or algorithms where the structure is independent of the type of element manipulated. As a trivial example, we consider an algorithm to swap a pair of elements in Figure 4.7.

The class Pair takes a type parameter “elem”. A pair has two fields x and y, both of type “elem”. The constructor Pair takes two elements and initialises the fields. The method “swap” interchanges the field contents, using a local variable “t”. The test code at the end creates a pair of integers and a pair of strings and prints to the standard output. We consider two ways in which Java may simulate polymorphism. The first method is to macro expand a new version of the Pair class for each type at which it is instantiated. It was called heterogeneous translation, and it is shown in Figure 4.8.

The appearance of parameterised classes Pair<String> and Pair<int> causes the creation of the expanded classes Pair<String> and Pair<int> within which each occurrence of the type variable “elem” is replaced by the types “String” and “int” respectively.
The second method is to replace the type variable “elem” by the class Object, the top of the class hierarchy. We call this the homogenous translation and it is shown in Figure 4.9. The key to this translation is that a value of any type may be converted into type Object and later recovered. Every type in Java is either a reference type or one of the eight base types, such as “int”. Each base type has a corresponding reference type, such as “Integer”, the relevant fragment of which appears in Figure 4.9.

If v is a variable of reference type, say String, then it is converted into an object o by widening (Object) v, and converted back by narrowing (String) o. If v is a value of base type, say int, then it is converted to an object o by (Object)(new Integer (v)), and converted back by ((Integer) o).intValue(). (In Java, widening may be implicit, the cast (Object) is explicitly for clarity.)

Java programmers can and do use idioms similar to the heterogenous and homogenous translations given here. Given the code duplication of the first, and the lengthy conversions of the second, the advantages of direct support for polymorphism are clear.

Figure 4.7 Polymorphism in Pizza

class Pair<elem> {
    elem x; elem y;
    Pair (elem x, elem y) {this.x = x; this.y = y;}
    void swap () {elem t = x; x = y; y = t;}
}
Pair<String> p = new Pair("world!", "Hello, ");
p.swap();
System.out.println(p.x + p.y);
Pair<int> q = new Pair(22, 64);
qu.swap();
System.out.println(q.x -q.y);

Figure 4.8 Heterogenous translation of polymorphism into Java

class Pair String {
    String x; String y;
    Pair String (String x, String y) {this.x = x; this.y = y;}
    void swap () {String t = x; x = y; y = t;}
}
class Pair_int {
    int x; int y;
    Pair_int (int x, int y) {this.x = x; this.y = y;}
    void swap () {int t = x; x = y; y = t;}
}
Pair_String p = new Pair_String("world!", "Hello, ");
p.swap(); System.out.println(p.x + p.y);
Pair int q = new Pair_int(22, 64);
qu.swap(); System.out.println(q.x-q.y);
A homogeneous translation produces generic code, shared by all instantiated classes. In the generic class, every formal type parameter is replaced by the corresponding upper bound. A heterogeneous translation produces specific code for each instantiated class.

The fact that a homogeneous translation uses the bound type in place of the actual type parameter does not affect the correct performance of the Java Virtual Machine. This is because Java is based on single dispatch. In addition, this representation does not affect the correct performance of static type checking.

Homogeneous translations are clearly more space-efficient than heterogeneous translations, but they suffer from serious limitations. The main problem with homogeneous translations is that the run-time information about classes and parameter types is incorrect, and in some cases unavailable. This is particularly relevant in languages like Java that support typed reflection.

Heterogeneous translations, on the other hand, provide correct run-time information about instantiated classes, but information about generic classes and formal parameter types is unavailable. Moreover, we do not think it is acceptable to allow the kind of unnecessary redundancy of code produced by heterogeneous translations. Furthermore, there are some situations that neither heterogeneous nor homogeneous translations can handle very well.
4.2.3 Rough Edges

There are surprisingly few places where it cannot achieve a good fit of Pizza to Java. Some of these are listed here: casting, visibility, and arrays.

Casting. Java ensures safe execution by inserting a run-time test at every narrowing from a superclass to a subclass. Pizza has a more sophisticated type system that renders some such tests redundant. Translating Pizza to Java (or to the Java Virtual Machine) necessarily incurs this modest extra cost. Java also promotes safety by limiting the casting operations between base types. By and large this is desirable, but it is a hindrance when implementing parametric polymorphism. For instance, instantiations of a polymorphic class at types int and float must have separate implementations in the heterogenous translation, even though the word level operations are identical. These modest costs could be avoided only by altering the Java Virtual Machine or by compiling Pizza directly to some other portable low-level code, such as Microsoft's OmniCode or Lucent's Dis.

Visibility. Java provides four visibility levels: private, visible only within the class; default, visible only within the package containing the class; protected, visible only within the package containing the class and within subclasses of the class; public, visible everywhere. Classes can only have default or public visibility; fields and methods of a class may have any of the four levels. A function abstraction in Pizza defines a new class in Java (to represent the closure) and a new method in the original class (to represent the abstraction body). The constructor for the closure class should be invoked only in the original class, and the body method should be invoked only within the closure class. Java provides no way to enforce this style of visibility. Instead, the closure class and body method must be visible at least to the entire package containing the original class. One cannot guarantee appropriate access to closures, unless one relies on dynamically allocated keys and run-time checks. For similar reasons, all fields of an algebraic type must have default or public visibility, even when private or protected visibility may be desirable.
Arrays. There is a rather poor fit between polymorphic arrays in Pizza and their
translation into Java. For instance:

Figure 4.10 Polymorphic arrays in Pizza

```java
class Rotate{
    static <elem> void rotate(elem[ ] x) {
        elem t = x[0];
        for (int i = 0; i < x.length-1; i++) f x[i] = x[i+1];
    x[x.length] = t;
}
}
String[ ] u = {"world!", "Hello"};
Rotate.rotate(u);
System.out.println(u[0]+" "+u[1]);
int[ ] v = {1,2,3,4}
Rotate.rotate(v);
```

Figure 4.11 Homogenous translation of polymorphic arrays into Java

```java
class Rotate {
    static void rotate(Array x){
        Object t = x.get(0);
        for(int i = 0;i<x.length()-1; i++) {x.set(i,x.get(i+1));
    x.set(x.length(),t);
}
}
String[ ] u = {"world!", "Hello"};
Rotate.rotate(new Array Object(u));
System.out.println(u[0]+" "+u[1]);
int[ ] v = {1,2,3,4}
Rotate.rotate(new Array int(v));
```

The homogenous translation for array creation is problematic. Consider the phrase “new
elem[size]”, where elem is a type variable and size is an integer. At first glance, the
translation “new Object[size]” might seem sensible, but a moment of thought shows this
fails for base types, since they are allocated differently. An additional moment of thought
shows that it also fails for reference types. For instance, if elem is String, then the result
is an array of objects that all happen to be strings, and there is no way in Java to cast this
to an array of strings.

4.2.4 ODMG Examples

In the following, I will give another example from the ODMG standard.

We can define d_Collection<T> using Pizza.
Figure 4.12 ODMG d_Set using Pizza

```java
interface d_Collection<T> {
    boolean d_is_empty();
    boolean d_is_ordered();
    boolean d_allows_duplicates();
    boolean d_containsElement(Object element);
    void d_insert_element(Object element);
    void d_remove_element(Object element);
}

class d_Set<T> implements Collection{
    d_Set<T> Value;
    d_Set<T> d_creat_union(d_Set<T> otherSet){};
    d_Set<T> d_creat_intersection(d_Set<T> otherSet){};
    d_Set<T> d_creat_difference(d_Set<T> otherSet){};
    boolean d_is_superset_of(d_Set<T> otherSet){};
    boolean d_is_proper_superset_of(d_Set<T> otherSet){};
    boolean d_is_subset_of(d_Set<T> otherSet){};
    boolean d_is_proper_subset_of(d_Set<T> otherSet){};
    boolean d_is_empty();
    boolean d_is_ordered();
    boolean d_allows_duplicates();
    boolean d_containsElement(Object element){};
    void d_insert_element(Object element){};
    void d_remove_element(Object element) throws ElementNotFound{};
}
```

Pizza can translate its code into Java code and compiles into the Java Virtual Machine.

So we can use the following definition:

```java
d_Set<Example> employees=new d_Set<Example>
```

### 4.3 JVM Load Time Expansion

#### 4.3.1 Load_Time Expansion

The implementation technique adds a preprocess phase to the Java Virtual Machine loader. The preprocessor instantiates parameterized classes at class load time to achieve a balance between pre-run-time expansion (for expressive power, and run-time efficiency), and minimization of compiled (or transmitted) code size.
Figure 4.13 shows a schematic diagram of the Java Virtual Machine [23]. When a running program refers to a class that has not been loaded, compiled bytecode passes through three largely independent components of the virtual machine. The loader obtains the bytecode for the class from either a local file or a remote site. The verifier validates the bytecode by checking that operation codes are valid, branches are to legitimate locations, methods have structurally correct signatures, and that every instruction obeys the Java type discipline. Finally, the linker initializes static fields of the new class and may load related classes.

It is possible for Java programs to modify the behavior of the loader on certain classes by extending the Java ClassLoader class. For example, while the default loading mechanism searches the local file system for classes, an alternate ClassLoader could obtain classes over the network.

The general strategy is to compile a parameterized class into an extended form of the class file. In addition to all the information usually found in a class file, the extended file includes information about parameters and constraints. When the virtual machine attempts to load an instantiation of a parameterized class or interface, a loader preprocess phase transforms the parameterized class file into the desired instantiation and then declares it as if it were a normal class. In other words, one “template” class file is used to
generate regular, non-parameterized class objects for each instantiation of a parameterized class.

The design of the loader provides a great deal of flexibility in the way the binary representation of a class is located. We can take advantage of this flexibility by augmenting the existing Java loader to create instantiations of parameterized classes. The steps involved in this process are listed below. Note that non-parameterized classes continue to be loaded in the usual manner. The steps to load an instantiation are:

- Convert the name of the class into a parameterized class name and argument names.
- Translate the argument names to class and interface types. This may involve loading other class files.
- Find the file containing the binary representation of the parameterized class and read it into memory.
- Check the parameter constraints against the arguments. If a constraint is violated, raise an exception.
- If the instantiation is valid, generate the binary representation of the instantiation from the parameterized class file and arguments; proceed in the usual manner to create a Class object for it. The bytecode verification process will ensure that the few, remaining type checks are performed.

Since the Java Virtual Machine specification does not allow class names of the syntactic form such as Hashtable<Name, Integer>, the compiler and ClassLoader must rewrite parameterized class names into a valid form.
A class file is a binary representation for a compiled class that can be loaded by any Java Virtual Machine [23]. The major pieces of the class file are a header, starting with a distinguishing magic number, a constant pool, and a description of the fields and methods in the class.

The constant pool contains strings representing all names mentioned in the class file, including class names, field and method names, type names, and so on. The index of an entry in the constant pool is used wherever that constant is needed in the class file. The Java Virtual Machine uses these strings for many operations including type checking, dynamic linking, and run-time resolution of field and method names.

To support load-time instantiation of parameterized class files, an extended class file format is needed, as shown in Figure 4.14. The file begins with a unique identifying number different from the standard class file magic number. Following this are the argument constraints, whose form is also shown.

The following introduce how to use the method defined above to implement adding type parameterization to the Java language.
Hashtable<$1,$2>$ insert method:
Method void insert($1,$2)
aload-
invokevirtual #6 <Method $1.hashCode()$>
istore_5
aload_0
iload_5
aload_1
aload_2
invokevirtual #7 <Method Hashtable<$1,$2$>.insertAt(IL$1;L$2;$)V>
return

Hashtable<Name, Integer> insert method:
Method void insert(Name, Integer)
aload_1
invokevirtual #6 <Method Name.hashCode()$>
istore_3
aload_0
iload_3
aload_1
aload_2
invokevirtual #7 <Method Hashtable<Name, Integer> .insertAt(ILName;LInteger;)V>
return

Figure 4.15 Java bytecodes for the insert method as it appears in the Hashtable parameterized class file, and the same method instantiated as Hashtable<Name, Integer>. insert.

4.3.2 ODMG Examples

The following is an ODMG example using the above method.

Figure 4.16 ODMG d_set using load_time expansion 1
interface d_Collection {
    boolean d_is_empty();
    boolean d_is_ordered();
    boolean d_allows_duplicates();
    boolean d_containsElement(Object element);
    void d_insert_element(Object element);
    void d_remove_element(Object element);
}

interface d_Set extends d_Collection{
    d_Set d_create_union(d_Set otherSet);
    d_Set d_create_intersection(d_Set otherSet);
    d_Set d_create_difference(d_Set otherSet);
    boolean d_is_superset_of(d_Set otherSet);
    boolean d_is_proper_superset_of(d_Set otherSet);
    boolean d_is_subset_of(d_Set otherSet);
    boolean d_is_proper_subset_of(d_Set otherSet);
    boolean d_is_empty();
    boolean d_is_ordered();
    boolean d_allows_duplicates();
    boolean d_containsElement(Object element);
    void d_insert_element(Object element);
    void d_remove_element(Object element);
}
Compiled from d_Set.java:

*Figure 4.17 ODMG d_set using load time expansion 2*

```java
interface d_Set extends d_Collection{
    /* ACC_SUPER bit NOT set */
    public abstract boolean d_allows_duplicates();
    public abstract boolean d_containsElement(java.lang.Object);
    public abstract d_Set d_create_difference(d_Set);
    public abstract d_Set d_create_intersection(d_Set);
    public abstract d_Set d_create_union(d_Set);
    public abstract void d_insert_element(java.lang.Object);
    public abstract boolean d_is_empty();
    public abstract boolean d_is_ordered();
    public abstract boolean d_is_proper_subset_of(d_Set);
    public abstract boolean d_is_proper_superset_of(d_Set);
    public abstract boolean d_is_subset_of(d_Set);
    public abstract void d_remove_element(java.lang.Object);
}
```

If we can design loader like this way:

*Figure 4.18 ODMG d_set using load time expansion 3*

```java
interface d_Set<$1> extends d_Collection{
    /* ACC_SUPER bit NOT set */
    public abstract boolean d_allows_duplicates();
    public abstract boolean d_containsElement(java.lang.Object);
    public abstract d_Set<$1> d_create_difference(d_Set<$1>);
    public abstract d_Set<$1> d_create_intersection(d_Set<$1>);
    public abstract d_Set<$1> d_create_union(d_Set<$1>);
    public abstract void d_insert_element(java.lang.Object);
    public abstract boolean d_is_empty();
    public abstract boolean d_is_ordered();
    public abstract boolean d_is_proper_subset_of(d_Set<$1>);
    public abstract boolean d_is_proper_superset_of(d_Set<$1>);
    public abstract boolean d_is_subset_of(d_Set<$1>);
    public abstract void d_remove_element(java.lang.Object);
}
```

We can easily define:

*Figure 4.19 ODMG d_set using load time expansion 4*

```java
interface d_Set<Employee> extends d_Collection{
    /* ACC_SUPER bit NOT set */
    public abstract boolean d_allows_duplicates();
    public abstract boolean d_containsElement(java.lang.Object);
    public abstract d_Set<Employee> d_create_difference(d_Set<Employee>);
    public abstract d_Set<Employee> d_create_intersection(d_Set<Employee>);
    public abstract d_Set<Employee> d_create_union(d_Set<Employee>);
    public abstract void d_insert_element(java.lang.Object);
    public abstract boolean d_is_empty();
    public abstract boolean d_is_ordered();
    public abstract boolean d_is_proper_subset_of(d_Set<Employee>);
    public abstract boolean d_is_proper_superset_of(d_Set<Employee>);
}
```
public abstract boolean d_is_subset_of(d_Set<Employee>);
public abstract boolean d_is_superset_of(d_Set<Employee>);
public abstract void d_remove_element(java.lang.Object);
}

4.4 Comparability

All of the above three methods can be used for adding parameterized types or generic classes to Java. Although it may appear that the language extensions from these proposals are equally expressive and that the different implementation strategies are more or less equal in power, fundamental differences do exist between them. This part discusses the designs, touching on the distinguishing features, as well as potential advantages and disadvantages of each.

Where Clause. On the positive side, where clauses allow the programmer to state that, a class conforms to a certain constraint without explicitly declaring the relationship when the class is defined. The main implementation described in [3] for where clauses changes the bytecodes and virtual machine significantly in order to allow shared code, but separate constant pool information, among instantiations of a parameterized class. This makes the implementation more costly and the virtual machine more complicated. Any change to the bytecodes and bytecode verifier will require all safety and security aspects of the system to be reevaluated. Clearly, this could be an undesirable and daunting task.

Pizza. Pizza is another parametric class extension using a constraint mechanism based on F-bounded quantification and extends and implements type relations. However, the type parameters in Pizza are prevented from appearing in certain places where class names may appear. For example, two errors occur in the following program when compiled in Pizza.

Figure 4.20 Rough edges of Pizza

class List<A>{
    static A a; //not allowed in Pizza
    void newA() {
        A a;
        a=new A(); //not allowed in Pizza
    }
}
Another restriction is that a class cannot inherit from a parameter, making mix-ins impossible to express. There are some other surprisingly places where Pizza can’t fit Java very well. These include casting, visibility, dynamic loading, and array.

JVM Load-Time Expansion. This way describes a parameterized type mechanism for the Java programming language and an implementation based on inserting a preprocess step into the load phase of the Java Virtual Machine [13, 14]. In comparison with the other two competing proposals described above, this implementation is relatively simple and supports more flexible language extensions than others. By delaying instantiation until load time, it is able to achieve an appropriate balance between language expressiveness, run-time efficiency and compiled code size. Moreover, use of the standard verifier and bytecode interpreter means that this extension does not compromise security properties. However, the changes to the Java virtual machine are restricted to the loader. It also makes the implementation more costly and the virtual machine more complicated.
Chapter 5 Java Binding of the ODMG Standard

5.1 Introduction

From the above introduction, we can easily find that in order to add parametric polymorphism to Java, the changing of Java Virtual Machine or the Java bytecode is unavoidable. Can we find a way making Java binding ODMG standard as great as C++? Can we use Java bind the ODMG models without touching anything inside Java and still keep the static type checking? Can we add parametric polymorphism to Java and define template classes in Java as it is used in C++?

The solution will be given in this chapter to answer all these questions. By the end of this chapter, I will attach the implementation programs, which can be used to prove the solution that we have established is correct.

As an object-oriented language, the major motivating factor in the invention of Java approach is to salvage some of the flaws encountered in the procedural approach. Java treats data as a critical element in the program development and does not allow it to freely flow around the system. It ties data more closely to the functions that operate on it and protects it from accidental modification from outside functions. Java allows us to decompose a problem into a number of entities called Java objects and then builds data and functions around these entities. The data of an object can be accessed only by the functions associated with the object. However functions of one object can access the functions of other objects. Some of the striking features of Java programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as Java objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
Java objects are considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

5.2 Techniques

In our ODMG java binding, we use the following Java techniques.

Objects. Objects are the basic run-time entities in Java system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely the real world objects. When a program is executed, the objects interact by sending messages to one another. For example, if "customer" and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted and type of response returned by the objects.

Classes. The entire set of data and code of an object can be made a user-defined data type with the help of a class. In fact, objects are variables of type class. Once a class has been defined, we can create any number of objects belonging to the class. Each object is associated with the data type class with which they are created. A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language.

Data Abstraction and Encapsulation. The wrapping up of data and functions into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those functions, which are wrapped in the class, can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access
by the program is called data hiding. Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction to define a list of abstract attributes such as size, weight, cost, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. Since the classes use the concept of data abstraction, they are known as Abstract Data Types.

Inheritance. Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example, the bird robin is a part of the class flying bird, which is again a part of the class bird. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived. In Java, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side effects into the rest of the classes.

Polymorphism. Polymorphism is another important Java concept. Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behaviors in different instances. The behavior depends upon the data types used in the operation. For example consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. This is something similar to a particular word having several different means depending upon the context. Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though the specific actions associated with each operation may different.

Dynamic Binding. Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a
given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Message Communication. A Java program consists of a set of objects that communicate with each other. Objects communicate with one another by sending and receiving information much like the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts. A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and the information sent.

5.3 Implementation

In our ODMG java binding, the core is using the above advantages of Java language to add parametric polymorphism to Java to make Java have the same “template class” feature as it is used in C++ language.

Parametric polymorphism is implemented by C++ through its template class. It adds parameter type constraints into the definition of classes to make it stricter and the static type checking is still reliable. The type mismatch errors will not happen in run time.

For example, the java compiler do not support such kind of expressions like List<Professor> or List<Student> however C ++ like it. List<Professor> means that in this List object, only “Professor” objects are valid. Any other objects whose types are not “Professor” will not be allowed to enter into it. It is a kind of restriction in the definition of class List. Please look at the following ODMG Collection interface definition:

Figure 5.1 ODMG Collection Interface

```java
interface Collection{
    boolean is_empty();
    boolean is_ordered();
    boolean allows_duplicates();
    boolean containsElement(Object element);
    void insert_element(Object element);
    void remove_element(Object element) throws ElementNotFound;
    Iterator create_iterator(boolean stable);
    BidirectionalIterator create_bidirectional_iterator(boolean stable) throws InvalidCollectionType;
    Object select_element(String OQL_predicate);
}
```
Iterator select(String OQL_predicate);
boolean query(String OQL_predicate, d_Collection result);
boolean exists_element(String OQL.predicate);
}

If we take a look at the methods in interface Collection, for example, “insert_element
(Object element)”, we can find that there is no restriction for the parameter “element”. In
other word, any object types can insert into it. If we defined the Collection interface like
Collection<Professor>, the parameter type must be “Professor”. Otherwise, the insert
method will not be invoked. Type mismatch error will show up. There are two issues up
to now. The first one is how to implement Collection <Professor> in Java under the
condition that Java does not support “<>” expression. The second one is how to add
parametric constrain to methods defined by collection types, for example, method
insert_element (Object element) in Collection <Professor>.

The ODMG Collection interface can be easily translated into java interface like:

\textbf{Figure 5.2 Java d\_Collection Interface}

\begin{verbatim}
public interface d_Collection {
    public boolean d_is_empty();
    public boolean d_is_ordered();
    public boolean dAllows_duplicates();
    public boolean dContainsElement(Object element);
    public void dInsert_element(Object element);
    public void dRemove_element(Object element) throws ElementNotFoundException;
    public Iterator dCreate_iterator(boolean stable);
    public BidirectionalIterator dCreate_bidirectional_iterator(boolean stable) throws
        InvalidCollectionType;
    public Object dSelect_element(String OQL_predicate);
    public Iterator dSelect(String OQL_predicate);
    public boolean dQuery(String OQL_predicate, d_Collection result);
    public boolean dExists_element(String OQL_predicate);
}
\end{verbatim}

In d_Collection interface, the common methods used by Array, Bag, Dictionary, List and
Set are included and it will be used throughout the whole ODMG java binding.

ODMG collections such as List, Bag, Set, Array and Dictionary have their own methods.
We use abstract classe to specify these distinct methods separately. For example, we can
define the ODMG List collection like:

\textbf{Figure 5.3 Java d\_List Class definition}

\begin{verbatim}
abstract class d_List implements d_Collection{
    protected void expansion(int number){};
    protected void dRemove_element_at(int index) throws InvalidIndex{};
}
\end{verbatim}
protected void d_insert_element_after( int index, Object element) throws InvalidIndex{};
protected void d_insert_element_before( int index, Object element) throws InvalidIndex{};
protected void d_insert_element_first(Object element) throws InvalidIndex{};
protected void d_insert_element_last(Object element) throws InvalidIndex{};
protected void d_remove_first_element() throws ElementNotFound{};
protected void d_remove_last_element() throws ElementNotFound{};
protected Object d_retrieve_first_element() throws ElementNotFound {return null;};
protected Object d_retrieve_last_element() throws ElementNotFound {return null;};
protected Object d_retrieve_element_at(int index) throws InvalidIndex{return null;};
protected void d_replace_element_at(Object element, int index) throws InvalidIndex{};
protected void d_append(d_List otherList){};
protected d_GeneralList concat(d_List other_List){return null;};
public void d_insert_element(Object element){};

Since interface cannot implement other interface. In order to implement interface, we
must define a class. In the above abstract class definition, the key word abstract is very
important. It means that we don't need to implement the entire methods defined in the
interface at this moment.

Then, we will define a general class to extend the above abstract class and implement all
of the methods defined by the interface and the abstract class. For example, the following
d_GeneralList class extends d_List and implements all of the methods in d_Collection
interface and d_List abstract class.

Figure 5.4 Java d_GeneralList Class definition
public class d_GeneralList extends d_List
{
    public Object data[];
    public d_GeneralList()
    {
        data= new Object[5];
    }
    public int count1()
    {
        int count1=data.length;
        for (int i=0;i<data.length;i++)
        {
            if (data[i]==null)
            {
                count1--;
            }
        }
        return count1;
    }
    public int count2(d_GeneralList otherList)
    {
        int count2=otherList.data.length;
        for (int i=0;i<otherList.data.length;i++)
        {
            if (otherList.data[i]==null)
        }
    }
    
    
    }  
    return count2;  
}

protected void expansion()
{
    Object data1[] = new Object[data.length+20];
    for (int i=0; i<count1; i++)
    {
        data1[i]=data[i];
    }
    data=data1;
}

/*implement ODMG list methods */
public void d_remove_element_at(int index) throws InvalidIndex
{
    if (index==0)
    {
        for (int i=index; i<data.length-index-1; i++)
        {
            data[i]=data[i+1];
            data[i+1]=null;
        }
    }
    else
    {
        for (int i=index; i<data.length-1; i++)
        {
            data[i]=data[i+1];
            data[i+1]=null;
        }
    }
}

protected void d_insert_element_after(int index, Object element) throws InvalidIndex
{
    int count1=count1();
    if (count1==data.length )
    {
        expansion();
    }
    if (count1<data.length )
    {
        for (int i=count1; i>index; i--)
        {
            data[i]=data[i-1];
        }
        data[index+1]=element;
    }
}

protected void d_insert_element_before(int index, Object element) throws InvalidIndex
{
    int count1=count1();
    if (count1==data.length )
    {
        expansion();
    }
```java
}  
if (count1<data.length)  
{  
  for (int i=count1; i>=index;i--)  
  {  
    data[i]=data[i-1];  
  }  
  data[index]=element;  
}  

protected void d_insert_element_first(Object element) throws InvalidIndex  
{  
  int count1=count1();  
  if (count1==data.length)  
  {  
    expansion();  
  }  
  if (count1<data.length)  
  {  
    for (int i=count1; i>0;i--)  
    {  
      data[i]=data[i-1];  
    }  
    data[0]=element;  
  }  
}  

protected void d_insert_element_last(Object element) throws InvalidIndex  
{  
  int count1=count1();  
  if (count1==data.length)  
  { 
    expansion(); 
  }  
  if (count1<data.length)  
  { 
    data[count1]=element; 
  }  
}  

protected void d_remove_first_element() throws ElementNotFound  
{  
  int count1=count1();  
  if (count1==data.length)  
  {  
    expansion();  
  }  
  data[count1]=null;  
  for (int i=0; i<count1;i++)  
  { 
    data[i]=data[i+1];  
  }  
}  

protected void d_remove_last_element() throws ElementNotFound  
{  
  int count1=count1();  
  data[count1-1]=null;  
}  

protected void d_remove_element_at(int index) throws InvalidIndex  
{  
  int count1=count1();  
  if (count1==data.length)  
  {  
    expansion();  
  }  
  if (count1<data.length)  
  {  
    for (int i=index; i<count1-1;i++)  
    {  
      data[i]=data[i+1];  
    }  
    data[index]=null;  
  }  
}  
```

{  
  expansion();
}
data[count1]=null;
for (int i=index; i<count1;i++)
{
  data[i]=data[i+1];
}

protected Object d_retrieve_first_element() throws ElementNotFound{
  return data[0];
}
protected Object d_retrieve_last_element() throws ElementNotFound{
  return data[count1-1];
}
protected Object d_retrieve_element_at(int index) throws InvalidIndex{
  return data[index];
}
protected void d_replace_element_at(Object element, int index) throws InvalidIndex{
  data[index]=element;
}
protected d_GeneralList d_concat(d_GeneralList otherList) throws InvalidIndex{
  d_append(otherList);
  return (this);
}
protected void d_append(d_GeneralList otherList) throws InvalidIndex{
  expansion();
  for (int i=0; i<count2(otherList);i++)
  {
    d_insert_element_last(otherList.data[i]);
  }
}

/*Implement ODMG Collection methods*/

public boolean d_exists_element(String OQL_predicate){
  return true;
}
public boolean d_query(String OQL_predicate, d_Collection result){
  return true;
}
public Object d_select_element(String OQL_predicate){
  return null;
}
public Iterator d_select(String OQL_predicate){
  return null;
}
public Iterator d_create_iterator(boolean stable){
  return null;
}
public BidirectionalIterator d_create_bidirectional_iterator(boolean stable) throws InvalidCollectionType{
  return null;
}
public boolean d_is_empty(){
if (count1==0)
{
    return true;
}
else return false;
}
public boolean d_is_ordered()
{
    return true;
}
public boolean d_allows_duplicates()
{
    return true;
}
public boolean d_containsElement(Object element)
{
    int flag=-1;
    for (int i=0; i<count1; i++)
    {
        if (data[i]==element)
        {
            flag++;
        }
    }
    if (flag>=0)
    {
        return true;
    }
    else return false;
}
public void d_insert_element(Object element)
{
    int count1=count1;
    if (count1==data.length)
    {
        Object data1[]=new Object[data.length+1];
        for (int i=0; i<count1; i++)
        {
            data1[i]=data[i];
        }
        data=data1;
    }
    data[count1]=element;
}
public void d_insert_element(Model element)
{
    int m,n;
    int count1=count1;
    boolean flag=false;
    if (count1==data.length)
    {
        expansion();
        for (int i=0; i<count1; i++)
        {
            if (((Model)data[i]).gt(element))
            {
                flag=true;
                for (n=count1; n>i; n--)
                {
                    data[n]=data[n-1];
                }
            }
        }
    }
}

- 48 -
data[n]=element;
for (m=0; m<i; m++){
    data[m]=data[m];
}
}

if (flag==false)
{
    for (m=0; m<count1; m++)
    {
        data[m]=data[m];
    }
    data[m]=element;
}
else
{
    for (int i=0; i<count1;i++)
    {
        if (((Model)data[i]).gt(element))
        {
            flag=true;
            for (n=count1; n>i; n--)
            {
                data[n]=data[n-1];
            }
            data[n]=element;
            for (m=0; m<i; m++)
            {
                data[m]=data[m];
            }
        }
        if (flag==false)
        {
            data[count1]=element;
        }
    }
}

public void d_remove_element(Object element) throws ElementNotFound {
    int index=0;
    int count=count1();
    for (int i=0; i<count; i++)
    {
        if (data[i]==element)
        {
            index=i;
            break;
        }
    }
    if (index==0)
    {
        for (int i=index; i<data.length-index-2;i++)
        {
            // 49
        }
    }
data[i]=data[i+1];
}

if (index!=0)
{
    for (int i=index; i<=data.length-index-1;i++)
    {
        data[i]=data[i+1];
    }
}

In the following part, a collection model class will be defined. This class extends the above class and it is very simple. It becomes simple because we don’t need to do any implementation here. The following example defines a `d_List$Models` class which extends class `d_GeneralList`. The technology that we used inside is Java “inheritance” and “Message Communication”.

**Figure 5.5 Java d_List$Models Class definition**

```java
public class d_List$Models extends d_GeneralList{
    public void remove_element_at(int index) throws InvalidIndex{
        d_remove_element_at(index);
    }

    public void insert_element_after(int index, Model element) throws InvalidIndex{
        super.d_insert_element_after(index, element);
    }

    public void insert_element_before(int index, Model element) throws InvalidIndex{
        super.d_insert_element_before(index, element);
    }

    public void insert_element_first(Model element) throws InvalidIndex{
        super.d_insert_element_first(element);
    }

    public void insert_element_last(Model element) throws InvalidIndex{
        super.d_insert_element_last(element);
    }

    public void insert_element(Model element){
        super.d_insert_element(element);
    }

    public void remove_first_element() throws ElementNotFound{
        d_remove_first_element();
    }

    public void remove_last_element() throws ElementNotFound{
        d_remove_last_element();
    }

    public Model retrieve_first_element() throws ElementNotFound{
        return (Model)super.d_retrieve_first_element();
    }

    public Model retrieve_last_element() throws ElementNotFound{
        return (Model)super.d_retrieve_last_element();
    }

    public Model retrieve_element_at(int index) throws InvalidIndex{
        return (Model)super.d_retrieve_element_at(index);
    }
}
```
return (Model)super.d_retrieve_element_at(index);
}
public void replace_element_at(Model element, int index) throws InvalidIndex{
    d_replace_element_at(element, index);
}
public d_GeneralList concat(d_GeneralList otherList) throws InvalidIndex{
    return (d_concat(otherList));
}
public void append(d_GeneralList otherList) throws InvalidIndex{
    d_append(otherList);
}
}

Using the "Data Abstraction" and "Data Encapsulation" features of Java, we can use another way to define the above model collection. See the following program.

Figure 5.6 Java dListModel$ModelsSpecial Class definition
public class dListModel$ModelsSpecial {
    d_GeneralList SpecialList = new d_GeneralList();
    public void remove_element_at(int index) throws InvalidIndex{
        SpecialList.d_remove_element_at(index);
    }
    public void insert_element_after(int index, Model element) throws InvalidIndex{
        SpecialList.d_insert_element_after(index, element);
    }
    public void insert_element_before(int index, Model element) throws InvalidIndex{
        SpecialList.d_insert_element_before(index, element);
    }
    public void insert_element_first(Model element) throws InvalidIndex{
        SpecialList.d_insert_element_first(element);
    }
    public void insert_element_last(Model element) throws InvalidIndex{
        SpecialList.d_insert_element_last(element);
    }
    public void insert_element(Model element){
        SpecialList.d_insert_element(element);
    }
    public void remove_first_element() throws ElementNotFound{
        SpecialList.d_remove_first_element();
    }
    public void remove_last_element() throws ElementNotFound{
        SpecialList.d_remove_last_element();
    }
    public Model retrieve_first_element() throws ElementNotFound{
        return (Model)SpecialList.d_retrieve_first_element();
    }
}
public Model retrieve_last_element() throws ElementNotFound
{
    return (Model)SpecialList.d_retieve_last_element();
}
public Model retrieve_element_at(int index) throws InvalidIndex
{
    return (Model)SpecialList.d_retieve_element_at(index);
}
public void replace_element_at(Model element, int index) throws InvalidIndex
{
    SpecialList.d_replace_element_at(element, index);
}
public d_GeneralList concat(d_GeneralList otherList) throws InvalidIndex
{
    return (SpecialList.d_concat(otherList));
}
public void append(d_GeneralList otherList) throws InvalidIndex
{
    SpecialList.d_append(otherList);
}

These two collection model classes are exactly what we want. The difference between these two definitions is that the second one doesn’t need to extend d_GeneralList, instead of creating a new instance of d_GeneralList.

Now, let’s consider how to use Java to implement C++ Templates such as List<Professor>. It becomes very simple if using the above programs. The only thing that you need is to replace the word “Model” to whatever object names you want, for instance, Professor, Student, Course, etc. It can easily generate collection d_List$Professors, collection d_List$Courses, etc. The following program defines an instance of class “d_List$Professors” and an instance of class “d_List$Courses”:

d_List$Professors professors= new d_List$Professors();
d_List$Courses courses= new d_List$Courses();

If we create an instance of d_List$Professors or d_List$Courses such as “professors” or “courses”, it indicates that only instances of object “Professor” or object “Course” have the right to come into or get out of the relative list. From the following program, you can find such kind of constrain.

Figure 5.7 Methods definition by Class d_List$Models
public void insert_element_before( int index, Model element) throws InvalidIndex
{
    super.d_insert_element_before(index, element);
}
public void insert_element_first(Model element) throws InvalidIndex
{
    super.d_insert_element_first(element);
}

The parameter type used in these two methods is “Model” instead of “Object” or “Any”. Only under the condition that the parameter type is correct, the method body can be invoked, otherwise, type mismatch error will show up.

According to the ODMG requirements, a List object is an ordered collection of elements. All elements in the list should be ordered. In the “insert_element” method body above, you will find it sends a message to its supper class to invoke method “insert_element” implemented there. The condition “if (((Model) data [i]).gt(element))” (“gt” means greater than) should be checked each time. Every element in the list must supply "boolean gt(Object)" method so that the element in the list can be reordered after the "d_insert_element" or d_delete_element” operation. This is another constrain for the parameters.

5.4 C++ Templates to Java Translator

Since ODMG has successfully used C++ binding their standards. In Java binding, we can reuse most of the C++ templates after doing some simple modifications.

In C++ binding, template classes such as List<Professor>, List<Model>, etc are widely used. Using our programs, we can easily translate them into Java. You just need to replace the symbol “<” to “$” and delete symbol “>” then copy the whole C++ template class to Java enivirement. You then generate the Java template classes without error. A valid Java template class name will be “List$Professor” or “List$Mode”, etc.
Chapter 6 Conclusion and Future Work

6.1 Conclusion

In this paper, we consider the following two key negative results:
—OQL queries cannot be properly type-checked in the type system of the ODMG object model and its object definition language (ODL).
—OQL queries cannot be properly type-checked, even in the type system of the Java binding of the ODMG standard.

Reasonable pragmatic solutions for these problems that do not require unattractive linguistic fix-ups and dynamic type-checking are as follows:
—The ODMG object model must be modified to provide proper support for parametric polymorphism.
—Java cannot be a viable database programming language (for the ODMG standard in particular), unless extended with parametric polymorphism.

The above solutions are in the spirit of several existing proposals for adding type parameterization to Java.

6.2 Innovation

We have developed an implementation technique showing that the current Java technology actually makes possible runtime linguistic reflection and suitable for Java OQL. The following is the achievements carried out from our work:
—The type system of the Java binding properly handles static type-checking of OQL queries.
—The type system of the ODMG object model are compatible with the type system of the Java programming language extended with parametric polymorphism.
—Our solution supports runtime linguistic reflection. This makes type-safe implementation with dynamic compilation of Java OQL possible.
—Our result makes the embedding a mostly statically-typed object-oriented query language (OQL) into a mostly statically-typed object-oriented programming language (Java) possible and makes this binding a statically-typed language.
The impedance mismatch does disappear in the technology proposed by the ODMG standard.

6.3 Future Work

One future work that should be carried out is to set up a series of components, which will be widely used among the commercial object-oriented database management systems, for example O2 [5].

O2 is a system that in fact has linguistic reflective capabilities of the sort required by Java OQL and has OQL [5]. But the type system of O2 is different from either the Java type system or any of the type systems of the ODMG standard [2].

Up to now, both the ODMG C++ binding, and our Java binding are all compliant with the ODMG standard. We should continue our work to make our contribution suitable among the whole ODBMS.
1. ODMG d\_Array

```java
public interface d\_Collection{
    public boolean d\_is\_empty();
    public boolean d\_is\_ordered();
    public boolean d\_allows\_duplicates();
    public boolean d\_contains\_element(Object element);
    public void d\_insert\_element(Object element);
    public void d\_remove\_element(Object element) throws ElementNotFoundException;
    public Iterator d\_create\_iterator(boolean stable);
    public Bidirectional\_Iterator d\_create\_bidirectional\_iterator(boolean stable) throws InvalidCollectionType;
    public Object d\_select\_element(String OQL\_predicate);
    public Iterator d\_select(String OQL\_predicate);
    public boolean d\_query(String OQL\_predicate, d\_Collection result);
    public boolean d\_exists\_element(String OQL\_predicate);
}

public interface d\_Array extends d\_Collection{
    public void expansion(int number);
    public void d\_replace\_element\_at(int index, Object element) throws InvalidIndex;
    public void d\_remove\_element\_at(int index) throws InvalidIndex;
    public Object d\_retrieve\_element\_at(int index) throws InvalidIndex;
    public void d\_resize(int new\_size) throws InvalidSize;
}

public class d\_GeneralArray implements d\_Array {
    public Object data[];
    public d\_GeneralArray() {
        data= new Object[4];
    }

    public int count1() {
        int count1=data.length;
        for (int i=0; i<count1; i++) {
            if (data[i]==null) {
                count1--;
            }
        }
        return count1;
    }

    public int count2(d\_GeneralArray otherList) {
        int count2=otherList.data.length;
        for (int i=0; i<count2; i++) {
            if (otherList.data[i]==null) {
                count2--;
            }
        }
        return count2;
    }

    public void expansion(int number) {
        Object data1[]=new Object[number];
        int count1=count1();
        for (int i=0; i<count1; i++) {
```
    data1[i]=data[i];
    }
    data=data1;
}

public void d_replace_element_at(int index, Object element) throws InvalidIndex
{
    data[index]=element;
}

public void d_remove_element_at(int index) throws InvalidIndex
{
    if (index==0)
    {
        for (int i=index; i<=data.length-index-2;i++)
        {
            data[i]=data[i+1];
        }
    }
    if (index!=0)
    {
        for (int i=index; i<=data.length-index-1;i++)
        {
            data[i]=data[i+1];
        }
    }
}

public Object d_retrieve_element_at(int index) throws InvalidIndex
{
    return data[index];
}

public void d_resize(int new_size) throws InvalidSize
{
    expansion(new_size);
}

public boolean d_exists_element(String OQL_predicate)
{
    return true;
}

public boolean d_query(String OQL_predicate, d_Collection result)
{
    return true;
}

public Object d_select_element(String OQL_predicate)
{
    return null;
}

public Iterator d_select(String OQL_predicate)
{
    return null;
}

public Iterator d_create_iterator(boolean stable)
{
    return null;
}

public BidirectionalIterator d_create_bidirectional_iterator(boolean stable) throws InvalidCollectionType
{
    return null;
}
public boolean d_is_empty()
{
    if (count1==0)
    {
        return true;
    }
    else return false;
}
public boolean d_is_ordered()
{
    return true;
}
public boolean d_alloweduplicates()
{
    return true;
}
public boolean d_containsElement(Object element)
{
    int flag=-1;
    for (int i=0;i<count1;i++)
    {
        if (data[i]==element)
        {
            flag++;    
        }
    }
    if (flag>=0)
    {
        return true;
    }
    else return false;
}
public void d_insert_element(Object element)
{
    int count1=count1();
    if (count1==data.length )
    {
        expansion(data.length +1);
    }
    if (count1<data.length )
    {
        data[count1]=element;
    }
}
public void d_remove_element(Object element) throws ElementNotFound
{
    int lock=0;
    int count1=count1();
    for (int i=0; i<count1 ; i++)
    {
        if(data[i]===element)
        {
            lock=i;
            data[i]=null;
            return true;
        }
    }
}
break;
}
}

try
{
    d_remove_element_at(lock);
} catch(InvalidIndex e1 )
{
    //do something
}

}

public class d_Array_Models extends d_GeneralArray{
    public void replace_element_at(int index, Model element) throws InvalidIndex
    {
        super.d_replace_element_at(index, element);
    }
    public void remove_element_at(int index) throws InvalidIndex
    {
        d_remove_element_at(index);
    }
    public Model retrieve_element_at(int index) throws InvalidIndex
    {
        return ((Model)(super.d_retrieve_element_at(index)));
    }
    public void resize(int new_size) throws InvalidSize
    {
        d_resize(new_size);
    }
    public boolean is_empty() throws InvalidSize
    {
        return d_is_empty();
    }
    public boolean containsElement(Model element)
    {
        return (super.d_containElement(element));
    }
    public void remove_element(Model element) throws ElementNotFound
    {
        super.d_remove_element(element);
    }
    public void insert_element(Model element)
    {
        super.d_insert_element(element);
    }
}
2. ODMG d_Bag

```java
public interface d_Collection{
    public boolean d_is_empty();
    public boolean d_is_ordered();
    public boolean d_allows_duplicates();
    public boolean d_containsElement(Object element);
    public void d_insert_element(Object element);
    public void d_remove_element(Object element) throws ElementNotFound;
    public Iterator d_create_iterator(boolean stable);
    public BidirectionalIterator d_create_bidirectional_iterator(boolean stable) throws InvalidCollectionTypeError;
    public Object d_select_element(String OQL_predicate);
    public Iterator d_select(String OQL_predicate);
    public boolean d_query(String OQL_predicate, d_Collection result);
    public boolean d_exists_element(String OQL_predicate);
}

abstract class d_Bag implements d_Collection{
    public Object[] d_create_union(d_Bag otherBag){ return null; };
    public Object[] d_create_intersection(d_Bag otherBag){ return null; };
    public Object[] d_create_difference(d_Bag otherBag){ return null; };
    public long d_occurrences_of(Object element){ return 0; };
    public void expansion(int number){}
    public void d_remove_element_at(int index) throws InvalidIndex{);
}

public class d_GeneralBag extends d_Bag {
    public Object data[];
    int k=0;
    public d_GeneralBag(){
        data= new Object[10];
    }
    public int count1(){
        int count1=data.length;
        for (int i=0;i<data.length;i++){
            if (data[i]==null){
                count1--;
            }
        }
        return count1;
    }
    public int count2(d_GeneralBag otherBag){
        int count2=otherBag.data.length;
        for (int i=0;i<otherBag.data.length;i++){
            if (otherBag.data[i]==null){
                count2--;
            }
        }
        return count2;
    }
    public void expansion(){
        Object data1[]=new Object[data.length+20];
        for (int i=0; i<count1();i++){
            data1[i]=data[i];
        }
        data=data1;
```
public void d_remove_element_at(int index) throws InvalidIndex
{
    if (index==0)
    {
        for (int i=index; i<=data.length-index-2; i++)
        {
            data[i]=data[i+1];
        }
    }
    if (index!=0)
    {
        for (int i=index; i<=data.length-index-1; i++)
        {
            data[i]=data[i+1];
        }
    }
}

public Object[] d_create_union(d_GeneralBag otherBag)
{
    int count1=count1();
    int count2=count2(otherBag);
    Object data1[]=new Object[count1+count2];
    for (int i=0; i<count1; i++)
    {
        data1[i]=data[i];
    }
    data=data1;
    for (int i=0; i<count2; i++)
    {
        data[count1+i]=otherBag.data[i];
    }
    return (this.data);
}

public Object[] d_create_intersection(d_GeneralBag otherBag)
{
    int count1=count1();
    int count2=count2(otherBag);
    int k=0;
    for (int i=0; i<count1; i++)
    {
        for (int j=0; j<count2; j++)
        {
            if (data[i]==otherBag.data[j])
            {
                k++;
            }
        }
    }
    Object temp[]=new Object[k];
    for (int i=0; i<count1; i++)
    {
        for (int j=0; j<count2; j++)
        {
            if (data[i]==otherBag.data[j])
            {
                temp[i]=data[i];
            }
        }
    }
public Object[] d_create_difference(d_GeneralBag A, d_GeneralBag otherBag) throws ElementNotFoundException {
    int count1 = count2(A);
    int count2 = count2(otherBag);
    int m = count1;
    int n = count2;
    for (int i = 0; i < count1; i++)
    {
        for (int j = 0; j < count2; j++)
        {
            if (A.data[i] == otherBag.data[j])
            {
                m--;
                A.d_remove_element(A.data[i]);
                n--;
                otherBag.d_remove_element(otherBag.data[i]);
            }
        }
    }
    for (int i = 0; i < k; i++)
    {
        for (int j = 0; j < count2; j++)
        {
            if (A.data[i] == otherBag.data[j])
            {
                count2--;
                otherBag.d_remove_element(otherBag.data[i]);
            }
        }
    }
    Object temp[] = new Object[m+n];
    temp = A.d_create_union(otherBag);
    /*
    for (int i = 0; i < m; i++)
    {
        temp[i] = A.data[i];
    }
    for (int i = count1+1; i < m+n; i++)
    {
        temp[i] = otherBag.data[i];
    }
    return (temp);
    */
    public long d_occurrences_of(Object element) {
        int flag = 0;
        for (int i = 0; i < count1; i++)
        {
            if (data[i] == element)
            {
                flag++;
            }
return flag;
}

public boolean d_exists_element(String OQL_predicate)
{
    return true;
}

public boolean d_query(String OQL_predicate, d_Collection result)
{
    return true;
}

public Object d_select_element(String OQL_predicate)
{
    return null;
}

public Iterator d_select(String OQL_predicate)
{
    return null;
}

public Iterator d_create_iterator(boolean stable)
{
    return null;
}

public BidirectionalIterator d_create_bidirectional_iterator(boolean stable) throws InvalidCollectionType
{
    return null;
}

public boolean d_is_empty()
{
    if (count()==0)
    {
        return true;
    }
    else return false;
}

public boolean d_is_ordered()
{
    return true;
}

public boolean dAllows_duplicates()
{
    return true;
}

public boolean d_containsElement(Object element)
{
    int flag=-1;
    for (int i=0;i<count();i++)
    {
        if (data[i]===element)
        {
            flag++;
        }
    }

if (flag >= 0)
    {
        return true;
    }
else return false;

public void d_insert_element(Object element)
    {
        int count1 = count1();
        if (count1 == data.length)
            {
                Object data1[] = new Object[data.length + 1];
                for (int i = 0; i < count1; i++)
                    {
                        data1[i] = data[i];
                    }
                data = data1;
            }
        data[count1] = element;
    }

public void d_remove_element(Object element) throws ElementNotFound
    {
        int lock = 0;
        int count1 = count1();
        for (int i = 0; i < count1; i++)
            {
                if (data[i] == element)
                    {
                        lock = i;
                        break;
                    }
        }
        try
            {
                d_remove_element_at(lock);
            } catch (InvalidIndex e1)
            {
                // do something
            }
    }

public class d_Bag_Models extends d_GeneralBag
    {
        public void remove_element_at(int index) throws InvalidIndex
            {
                d_remove_element_at(index);
            }

        public boolean is_empty() throws InvalidSize
            {
                return d_is_empty();
            }

        public boolean containsElement(Model element)
            {
                return (super.d_containsElement(element));
            }
public void remove_element(Model element) throws ElementNotFoundException{
    super.d_remove_element(element);
}

public void insert_element(Model element){
    super.d_insert_element(element);
}

public d_GeneralBag create_union(d_GeneralBag ABag, d_GeneralBag BBag){
    ABag.data = d_create_union(BBag);
    return ABag;
}

public d_GeneralBag create_intersection(d_GeneralBag ABag, d_GeneralBag BBag){
    ABag.data = d_create_intersection(BBag);
    return ABag;
}

public d_GeneralBag create_difference(d_GeneralBag ABag, d_GeneralBag BBag) throws
ElementNotFoundException{
    ABag.data = d_create_difference(ABag,BBag);
    return ABag;
}

public long occurrences_of(Model element) {
    return super.d_occurrences_of(element);
}
3. ODMG d_Dictionary

```java
class d_Dictionary {
    public boolean is_empty() {
        return true;
    }
}
```
public boolean d_contains_key(Object key){
    if (Dictionary.containsKey(key))
    {
        return true;
    }
    else return false;
}

public boolean d_exists_element(String OQL_predicate)
{
    return true;
}

public boolean d_query(String OQL_predicate, d_Collection result){
    return true;
}

public Object d_select_element(String OQL_predicate){
    return null;
}

public Iterator d_select(String OQL_predicate){
    return null;
}

public Iterator d_create_iterator(boolean stable){
    return null;
}

public BidirectionalIterator d_create_bidirectional_iterator(boolean stable) throws InvalidCollectionType{
    return null;
}

public boolean d_is_empty(){
    if (Dictionary.getEmpty() )
    {
        return true;
    }
    else return false;
}

public boolean d_is_ordered(){
    return false;
}

public boolean d_allows_duplicates(){
    return false;
}

public boolean d_containsElement(Object key){
    if (Dictionary.containsKey(key))
    {
        return true;
    }
    else return false;
}
public void d_insert_element(Object element) { }
public void d_remove_element(Object element) throws ElementNotFoundException { }

public class d_Dictionary_Models extends d_GeneralDictionary{

    void bind(Object key, Model value) throws DuplicateName
    {
        super.d_bind(key, value);
    }

    void unbind(Object key) throws KeyNotFoundException
    {
        d_unbind(key);
    }

    boolean contains_key(Object key)
    {
        return d_contains_key(key);
    }

    Object lookup(Object key) throws KeyNotFoundException
    {
        return d_lookup(key);
    }
}
4. ODMG d_list

```java
public interface d_Collection{
    public boolean d_is_empty();
    public boolean d_is_ordered();
    public boolean d_allows_duplicates();
    public boolean d_containsElement(Object element);
    public void dInsert_element(Object element);
    public void d_remove_element(Object element) throws ElementNot_found;
    public Iterator d_create_iterator(boolean stable);
    public BidirectionalIterator d_create_bidirectional_iterator(boolean stable) throws InvalidCollectionType;
    public Object d_select_element(String OQL_predicate);
    public Iterator d_select(String OQL_predicate);
    public boolean d_query(String OQL_predicate, d_Collection result);
    public boolean d_exists_element(String OQL_predicate);
}

abstract class d_List implements d_Collection{
    protected void expansion(int number){}
    protected void d_remove_element_at(int index) throws InvalidIndex{};
    protected void dInsert_element_after(int index, Object element) throws
        InvalidIndex{};
    protected void dInsert_element_before(int index, Object element) throws
        InvalidIndex{};
    protected void dInsert_element_first(Object element) throws InvalidIndex {};
    protected void dInsert_element_last(Object element) throws InvalidIndex{};
    protected void d_remove_first_element() throws ElementNot_found{};
    protected void d_remove_last_element() throws ElementNot_found{};
    protected Object d_retrieve_first_element() throws ElementNot_found
        {return null;};
    protected Object d_retrieve_last_element() throws ElementNot_found
        {return null;};
    protected Object d_retrieve_element_at(int index) throws InvalidIndex
        {return null;};
    protected void d_replace_element_at(Object element, int index) throws
        InvalidIndex{};
    protected void d_append(d_List otherList){};
    protected d_GeneralList concat(d_List other_List){return null;};
    public void d_insert_element(Object element){};
}

public class d_GeneralList extends d_List{
    public Object data[];
    public d_GeneralList(){
        data= new Object[5];
    }
    public int count1(){
        int count1=data.length;
        for (int i=0;i<data.length;i++){
            if (data[i]==null){
                count1--;
            }
        }
    }
```
return count1;
}
public int count2(d_GeneralList otherList)
{
    int count2=otherList.data.length;
    for (int i=0; i<otherList.data.length; i++)
    {
        if (otherList.data[i]==null)
        {
            count2--;
        }
    }
    return count2;
}
protected void expansion()
{
    Object data1[]=new Object[data.length+20];
    for (int i=0; i<count1();i++)
    {
        data1[i]=data[i];
    }
    data=data1;
}
public void d_remove_element_at(int index) throws InvalidIndex
{
    if (index==0)
    {
        for (int i=index; i<data.length-index-1; i++)
        {
            data[i]=data[i+1];
            data[i+1]=null;
        }
    }
    else
    {
        for (int i=index; i<data.length-1; i++)
        {
            data[i]=data[i+1];
            data[i+1]=null;
        }
    }
}
protected void d_insert_element_after( int index, Object element) throws InvalidIndex
{
    int count1=count1();
    if (count1==data.length ){
        expansion();
    }
    if (count1<data.length ){
        for (int i=count1; i>index;i--){
            data[i]=data[i-1];
        }
        data[index+1]=element;
    }
}
protected void d_insert_element_before( int index, Object element) throws InvalidIndex
{
```java
int count1 = count1;
if (count1 == data.length)
    {
        expansion();
    }
    if (count1 < data.length)
    {
        for (int i = count1; i >= index; i--)
        {
            data[i] = data[i - 1];
        }
        data[index] = element;
    }

protected void d_insert_element_first(Object element) throws InvalidIndex
{
    int count1 = count1;
    if (count1 == data.length)
        {
            expansion();
        }
    if (count1 < data.length)
    {
        for (int i = count1; i > 0; i--)
        {
            data[i] = data[i - 1];
        }
        data[0] = element;
    }
}

protected void d_insert_element_last(Object element) throws InvalidIndex
{
    int count1 = count1;
    if (count1 == data.length)
        {
            expansion();
        }
    if (count1 < data.length)
    {
        data[count1] = element;
    }
}

protected void d_remove_first_element() throws ElementNotFoundException
{
    int count1 = count1;
    if (count1 == data.length)
        {
            expansion();
        }
    data[count1] = null;
    for (int i = 0; i < count1; i++)
    {
```

data[i]=data[i+1];
}
}
protected void d_remove_last_element() throws ElementNotFoundException
{
    int count1=findAll();
    data[count1-1]=null;
}
protected void d_remove_element_at(int index) throws InvalidIndex
{
    int count1=findAll();
    if (count1==data.length )
    {
        expansion();
    }
    data[count1]=null;
    for (int i=index; i<count1;i++)
    {
        data[i]=data[i+1];
    }
}
protected Object d_retrieve_first_element() throws ElementNotFoundException
{
    return data[0];
}
protected Object d_retrieve_last_element() throws ElementNotFoundException
{
    return data[count1-1];
}
protected Object d_retrieve_element_at(int index) throws InvalidIndex
{
    return data[index];
}
protected void d_replace_element_at(Object element, int index) throws InvalidIndex
{
    data[index]=element;
}
protected d_GeneralList d_concat(d_GeneralList otherList) throws InvalidIndex
{
    d_append(otherList);
    return (this);
}
protected void d_append(d_GeneralList otherList) throws InvalidIndex
{
    expansion();
    for (int i=0; i<count2(otherList);i++)
    {
        d_insert_element_last(otherList.data[i]);
    }
}
public boolean d_exists_element(String OQL_predicate)
{
    return true;
}
public boolean d_query(String OQL_predicate, d_Collection result)
{
    return true;
}

public Object d_select_element(String OQL_predicate)
{
    return null;
}

public Iterator d_select(String OQL_predicate)
{
    return null;
}

public Iterator d_create_iterator(boolean stable)
{
    return null;
}

public BidirectionalIterator d_create_bidirectional_iterator(boolean stable) throws InvalidCollectionType
{
    return null;
}

public boolean d_is_empty()
{
    if (count()==0)
    {
        return true;
    }
    else return false;
}

public boolean d_is_ordered()
{
    return true;
}

public boolean d_allows_duplicates()
{
    return true;
}

public boolean d_containsElement(Object element)
{
    int flag=-1;
    for (int i=0;i<count();i++)
    {
        if (data[i]==element)
        {
            flag++;
        }
    }
    if (flag>=0)
    {
        return true;
    }
    else return false;
}

d tolerate                      

d_persister                     

d_insert_element(Model element){
int m,n;
int countl=countl();
    boolean flag=false;
if (countl==data.length)
{
    expansion();
    for (int i=0; i<countl;i++)
    {
        if (((Model)data[i]).gt(element))
        {
            flag=true;
            for (n=countl; n>i; n--)
            {
                data[n]=data[n-1];
            }
            data[n]=element;
            for (m=0; m<i; m++)
            {
                data[m]=data[m];
            }
        }
    }
    if (flag==false)
    { 
        for (m=0; m<countl; m++)
        {
            data[m]=data[m];
        }
        data[m]=element;
    }
    else
    {
        for (int i=0; i<countl;i++)
        {
            if (((Model)data[i]).gt(element))
            {
                flag=true;
                for (n=countl; n>i; n--)
                {
                    data[n]=data[n-1];
                }
                data[n]=element;
                for (m=0; m<i; m++)
                {
                    data[m]=data[m];
                }
            }
        }
    }
    if (flag==false)
    {
        data[countl]=element;
    }
}
public void d_remove_element(Object element) throws ElementNotFound {
    int index=0;
    int count=count1();
    for (int i=0; i<count; i++)
    {
        if(data[i]==element)
        {
            index=i;
            break;
        }
    }
    if (index==0)
    {
        for (int i=index; i<=data.length-index-2;i++)
        {
            data[i]=data[i+1];
        }
    }
    if (index!=0)
    {
        for (int i=index; i<=data.length-index-1;i++)
        {
            data[i]=data[i+1];
        }
    }
}

public class d_List_Models extends d_GeneralList{
    public void remove_element_at(int index) throws InvalidIndex
    {
        d_remove_element_at(index);
    }
    public void insert_element_after( int index, Model element) throws InvalidIndex
    {
        super.d_insert_element_after(index, element);
    }
    public void insert_element_before( int index, Model element) throws InvalidIndex
    {
        super.d_insert_element_before(index, element);
    }
    public void insert_element_first(Model element) throws InvalidIndex
    {
        super.d_insert_element_first(element);
    }
    public void insert_element_last(Model element) throws InvalidIndex
    {
        super.d_insert_element_last(element);
    }
    public void insert_element(Model element)
    {
        super.d_insert_element(element);
    }
}
public void remove_first_element() throws ElementNotFoundException
{
    d_remove_first_element();
}

public void remove_last_element() throws ElementNotFoundException
{
    d_remove_last_element();
}

public Model retrieve_first_element() throws ElementNotFoundException
{
    return (Model)super.d_retrieve_first_element();
}

public Model retrieve_last_element() throws ElementNotFoundException
{
    return (Model)super.d_retrieve_last_element();
}

public Model retrieve_element_at(int index) throws InvalidIndex
{
    return (Model)super.d_retrieve_element_at(index);
}

public void replace_element_at(Model element, int index) throws InvalidIndex
{
    d_replace_element_at(element, index);
}

public d_GeneralList concat(d_GeneralList otherList) throws InvalidIndex
{
    return (d_concat(otherList));
}

public void append(d_GeneralList otherList) throws InvalidIndex
{
    d_append(otherList);
}
5. ODMG d_Set

public interface d_Collection{
    public boolean d_is_empty();
    public boolean d_is_ordered();
    public boolean d_allows_duplicates();
    public boolean d_containsElement(Object element);
    public void d_insert_element(Object element);
    public void d_remove_element(Object element) throws ElementNotFound;
    public Iterator d_create_iterator(boolean stable);
    public BidirectionalIterator d_create_bidirectional_iterator(boolean stable) throws InvalidCollectionType;
    public Object d_select_element(String OQL_predicate);
    public Iterator d_select(String OQL_predicate);
    public boolean d_query(String OQL_predicate, d_Collection result);
    public boolean d_exists_element(String OQL_predicate);
}

abstract class d_Set implements d_Collection{
    protected void expansion(int number){}
    protected void d_remove_element_at(int index) throws InvalidIndex{ }
    protected Object[] d_create_union(d_Set otherSet){return null;}
    protected Object[] d_create_intersection(d_Set otherSet){return null;}
    protected Object[] d_create_difference(d_Set otherSet){return null;}
    protected boolean d_is_subset_of(d_Set otherSet){return true;}
    protected boolean d_is_proper_subset_of(d_Set otherSet){return true;}
    protected boolean d_is_superset_of(d_Set otherSet){return true;}
    protected boolean d_is_proper_superset_of(d_Set otherSet){return true;}
}

public class d_GeneralSet extends d_Set {
    public Object data[];
    public d_GeneralSet()
    {
        data = new Object[5];
    }

    public int count1(){
        int count1 = data.length;
        for (int i=0;i< data.length;i++){
            if (data[i]==null){
                count1--;
            }
        }
        return count1;
    }

    public int count2(d_GeneralSet otherSet)
    {
        int count2 = otherSet.data.length;
        for (int i=0;i< otherSet.data.length;i++)
        {
            if (otherSet.data[i]==null){
                count2--;
            }
        }
        return count2;
    }
}

- 77 -
public void expansion(){
    Object data1[]=new Object[data.length+20];
    for (int i=0; i<count1;i++){
        data1[i]=data[i];
    }
    data=data1;
}

public void d_remove_element_at(int index) throws InvalidIndex
{
    if (index==0)
    {
        for (int i=index; i<data.length-index-1;i++)
        {
            data[i]=data[i+1];
            data[i+1]=null;
        }
    }
    else
    {
        for (int i=index; i<data.length-1;i++)
        {
            data[i]=data[i+1];
            data[i+1]=null;
        }
    }
}

public Object[] d_create_union(d_GeneralSet oneSet, d_GeneralSet otherSet) throws ElementNotFoundException
{
    int count1=count2(oneSet);
    int count2=count2(otherSet);
    int m=count1;

    Object temp[]= new Object[m+count2];
    for( int i=0; i<count2; i++)
    {
        temp[i]=otherSet.data[i];
    }
    for( int i=0; i<count1; i++)
    {
        for (int j=0; j<count2(oneSet); j++)
        {
            if(oneSet.data[i]==otherSet.data[j])
            {
                m--;
                oneSet.d_remove_element(oneSet.data[i]);
            }
        }
    }
    for (int i=0; i<m;i++)
    {
        temp[count2+i]=oneSet.data[i];
    }
}
public Object[] d_creat_intersection(d_GeneralSet otherSet)
{
    int count1 = count1();
    int count2 = count2(otherSet);
    int m = 0;
    for (int i = 0; i < count1; i++)
    {
        for (int j = 0; j < count2; j++)
        {
            if (data[i] == otherSet.data[j])
            {
                m++;
            }
        }
    }
    Object temp[] = new Object[m];
    for (int i = 0; i < count1; i++)
    {
        for (int j = 0; j < count2; j++)
        {
            if (data[i] == otherSet.data[j])
            {
                temp[i] = data[i];
            }
        }
    }
    return (temp);
}

public Object[] d_creat_difference(d_GeneralSet oneSet, d_GeneralSet otherSet) throws ElementNotFound
{
    int count1 = count2(oneSet);
    int count2 = count2(otherSet);
    int m = count1;
    for (int i = 0; i < count2(oneSet); i++)
    {
        for (int j = 0; j < count2; j++)
        {
            if (oneSet.data[i] == otherSet.data[j])
            {
                m--;
                oneSet.d_remove_element(oneSet.data[i]);
            }
        }
    }
    Object temp[] = new Object[count2(oneSet)];
    for (int i = 0; i < m; i++)
    {
        temp[i] = oneSet.data[i];
    }
    return (temp);
}

int count1 = count2(oneSet);
int count2 = count2(otherSet);
    int m = count1;
    int n = count2;
    int k = 0;
    Object temp1[] = new Object[m+n];
    Object temp2[] = new Object[m+n];
    Object temp3[] = new Object[m+n];

    temp1 = oneSet.d_create_intersection(otherSet);
    temp2 = oneSet.d_create_union(otherSet);

    for (int j = 0; j < temp1.length; j++)
    {
        for (int i = 0; i < temp2.length; i++)
        {
            if (!((temp1[i] == temp2[i])))
            {
                temp3[i] = temp3[i];
            }
        }
    }

    for (int i = 0; i < count1; i++)
    {
        for (int j = 0; j < count2; j++)
        {
            if (oneSet.data[i] == otherSet.data[j])
            {
                m--;
                n--;
                temp1[i] = oneSet.data[i];
                k++;
            }
        }
    }
    Object temp[] = new Object[m+n];
    for (int i = 0; i < k; i++)
    {
        for (int j = 0; j < count1; j++)
        {
            if (oneSet.data[j] == temp1[i])
            {
                oneSet.d_remove_element_at(j);
            }
        }
    }

    for (int i = 0; i < k; i++)
    {
        for (int j = 0; j < count2; j++)
        {
            if (otherSet.data[j] == temp1[i])
            {
                - 80 -
            }
        }
    }
otherSet.d_remove_element_at(j);

public boolean d_is_superset_of(d_GeneralSet otherSet)
{
    int count=0;
    for (int i=0;i<count2(otherSet);i++)
    {
        for (int j=0;j<count1();j++)
        {
            if (data[i]==otherSet.data[j])
                count++;
        }
    }
    if ((count==count2(otherSet)) && (count1()>=count2(otherSet)))
    {
        return true;
    }
    else return false;
}

public boolean d_is_proper_superset_of(d_GeneralSet otherSet)
{
    int count=0;
    for (int i=0;i<count2(otherSet);i++)
    {
        for (int j=0;j<count1();j++)
        {
            if (data[i]==otherSet.data[j])
                count++;
        }
    }
    if ((count==count2(otherSet)) && (count1()>count2(otherSet)))
    {
        return true;
    }
    else return false;
}
public boolean d_is_subset_of(d_GeneralSet otherSet)
{
    int count=0;
    for (int i=0;i<count1();i++)
    {
        for (int j=0;j<count2(otherSet);j++)
        {
            if (data[i]==otherSet.data[j])
            {
                count++;
            }
        }
    }
    if ((count==count1()) & (count1()<=count2(otherSet)))
    {
        return true;
    }
    else return false;
} // END OF METHOD d_is_subset_of;

public boolean d_is_proper_subset_of(d_GeneralSet otherSet)
{
    int count=0;
    for (int i=0;i<count1();i++)
    {
        for (int j=0;j<count2(otherSet);j++)
        {
            if (data[i]==otherSet.data[j])
            {
                count++;
            }
        }
    }
    if ((count==count1()) & (count1()<count2(otherSet)))
    {
        return true;
    }
    else return false;
} // END OF METHOD d_is_proper_subset_of;

public boolean d_exists_element(String OQL_predicate)
{
    return true;
}

public boolean d_query(String OQL_predicate, d_Collection result)
{
    return true;
}
public Object d_select_element(String OQL_predicate)
{
    return null;
}

public Iterator d_select(String OQL_predicate)
{
    return null;
}

public Iterator d_create_iterator(boolean stable)
{
    return null;
}

public BidirectionalIterator d_create_bidirectional_iterator(boolean stable) throws InvalidCollectionType
{
    return null;
}

public boolean d_is_empty()
{
    if (count1==0)
    {
        return true;
    }
    else return false;
}

public boolean d_is_ordered()
{
    return true;
}

public boolean d_allows_duplicates()
{
    return true;
}

public boolean d_containsElement(Object element)
{
    int flag=-1;
    for (int i=0;i<count1;i++)
    {
        if (data[i]==element)
        {
            flag++;
        }
    }
    if (flag>=0)
    {
        return true;
    }
    else return false;
}

public void d_insert_element(Object element)
{
    int count1=count1;
    if (count1==data.length)
    {
        Object data1[]=new Object[data.length+1];
        for (int i=0; i<count1;i++)
            data1[i]...
{ 
data1[i]=data[i];
}
data=data1;
}
data[count1]=element;
}
public void d_remove_element(Object element) throws ElementNotFoundException
{
    int index=0;
    int count=count1();
    for (int i=0; i<count; i++)
    {
        if(data[i]==element)
        {
            index=i;
            break;
        }
    }
    if (index==0)
    {
        for (int i=index; i<data.length-index-2;i++)
        {
            data[i]=data[i+1];
        }
    }
    if (index!=0)
    {
        for (int i=index; i<data.length-index-1;i++)
        {
            data[i]=data[i+1];
        }
    }
}
}

class d_Set_Models extends d_GeneralSet{

    public void remove_element_at(int index) throws InvalidIndex
    {
        d_remove_element_at(index);
    }
    public boolean is_empty() throws InvalidSize
    {
        return d_is_empty();
    }
    public boolean containsElement(Model element)
    {
        return d_containsElement(element);
    }
    public void remove_element(Model element) throws ElementNotFoundException
    {
        super.d_remove_element(element);
    }
}
public void insert_element(Model element) {
    super.d_insert_element(element);
}

public d_GeneralSet create_union(d_GeneralSet OneSet, d_GeneralSet OtherSet) throws ElementNotFoundException {
    OneSet.data = d_create_union(OneSet, OtherSet);
    return OneSet;
}

public d_GeneralSet create_intersection(d_GeneralSet OneSet, d_GeneralSet OtherSet) {
    OneSet.data = d_create_intersection(OtherSet);
    return OneSet;
}

public d_GeneralSet create_difference(d_GeneralSet OneSet, d_GeneralSet OtherSet) throws ElementNotFoundException {
    OneSet.data = d_create_difference(OneSet, OtherSet);
    return OneSet;
}

public boolean is_superset_of(d_GeneralSet otherSet) {
    return (d_is_superset_of(otherSet));
}

public boolean is_proper_superset_of(d_GeneralSet otherSet) {
    return (d_is_proper_superset_of(otherSet));
}

public boolean is_subset_of(d_GeneralSet otherSet) {
    return (d_is_subset_of(OtherSet));
}

public boolean is_proper_subset_of(d_GeneralSet otherSet) {
    return (d_is_proper_subset_of(OtherSet));
}
References


VITA AUCTORIS

Name: Ye TIAN
PLACE OF BIRTH: Liaoning Province, P.R.China
YEAR OF BIRTH: 1973
EDUCATION: the 2\textsuperscript{th} High School of Shenyang, China
\hspace{1em}1989-1992
\hspace{1em}DaLian University of Technology, China
\hspace{1em}1992-1997 B.Sc
\hspace{1em}School of Computer Science
\hspace{1em}University of Windsor, Windsor, Ontario
\hspace{1em}1999-2001 M.Sc
\hspace{1em}School of Computer Science