Baylor University

Department of Computer Science

Master's Project

Extending Hibernate

Author:

Sweta Shrestha

Mentor:

Dr. Gregory D. Speegle

Committee Members:

Dr. Michael J. Donahoo

Dr. Lesley Wright

ABSTRACT

Extending Hibernate

Sweta Shrestha, M.S.

Chairperson: Dr. Gregory D. Speegle, Ph.D.

Hibernate is an open source ORM (Object Relational Mapping) tool. It maps objects into a relational database and vice versa. Hibernate persists objects of a class in a relational database so that they can be retrieved even after the application program terminates. Hibernate is a huge project which has been evolving ever since its first release in 2001. New features are added and previous flaws are fixed with every new version of Hibernate. This project adds a new feature to Hibernate to broaden the scope of use of one-to-many construct for a one-to-many association. This extension allows a user to use the one-to-many construct with a multiple table representation of a class as a part of the association. The project implements the execution path of the workaround currently used in Hibernate to provide this functionality.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE
Object Relational Mapping

1.1 Object Oriented Programming and Relational Databases

Object Oriented Programming (OOP) is used extensively in application program development and design. OOP implements powerful features like inheritance, polymorphism and encapsulation. Properties (called data members or attributes) that are related and the operations (called methods) that apply to these properties are encapsulated in a common structure called a *class*. Instances of a class are called *objects*. The object oriented paradigm has become so popular, that the concept of objects has surpassed the boundary of programming and many applications now require objects to be persisted in storage for future use.

A database is an organized repository that is capable of storing and retrieving a huge amount of data efficiently. Relational databases are based on the relational model and store data in tables. Structured Query Language (SQL) is the common standardized language [1] used to store and retrieve data from a relational database. Relational databases became available in the early 70's [6] and the popularity of SQL made relational databases the dominant platform for storing data. A majority of the databases used today are relational databases [6].

With the dominance of relational databases and the need to store objects for future use, an interface that communicates between relational databases and object oriented applications is required. Objects cannot be directly saved to and retrieved from relational databases because there are conceptual differences between OOP and relational databases. These differences are termed *impedance mismatch* [7]. ORM (Object Relational Mapping) is a new methodology for overcoming impedance mismatch. ORM tools store and retrieve objects from a relational database with minimal programming.

Three of the key issues of impedance mismatch related to the project are defined below [7].

I **Identity:** In Java, there are two ways to compare objects - one using "==" operator and the other using the equals() method inherited from the Object class. The equals operator, "==", tests whether two objects refer to the same memory location, while equals() compares the two objects based on their contents. In databases, the primary key of a row determines the identity of the row. Here we can see a subtle difference in the definition of identity in the OOP and in relational databases. For example, the following two statements extract an object of the class Person with id (primary key) 0001.

Person p1 = get Person whose id = 0001;

Person p2 = get Person whose id = 0001;

In this case, the database rows accessed in executing the two statements are the same. However, according to the Java rules, p1 == p2 should be false since these are two different Java objects.

II **Subclass:** Inheritance is an integral characteristic of OOP, but relational databases do not have any notion of inheritance. Inheritance must be enforced by the designer and involves constraints which are not commonly supported in RDBMS [10].

For example, the class Cat inherits from the class Animal. When instances of these classes are created, these instances automatically possess the correct properties. The objects of the class Animal possess the properties of the class Animal and the objects of the class Cat possess the properties defined in the class Cat, and public and protected properties from the class Animal. If these classes are persisted in a table in the database, the database will not have any knowledge which columns correspond to properties of a particular class. If these classes are persisted in different tables, there is no RDBMS concept to fetch records from both the tables when there is a request for retrieving all instances of class Cat.

III **Association:** Associations among objects are represented using references. If an object A has an association with an object B, then in OOP the object A has a reference to object B. In relational databases, associations are represented using foreign keys. With the help of the foreign keys, all the associated data can be retireved from another table. References are directional in nature. For example, if a class A has a reference to a class B, it is not necessarily true that the class B also has a reference to the class A. However, joins are not directional in nature. With a single join, all data can be accessed from either table.

There are other ways to integrate databases and OOP applications. JDBC is an industry standard from Sun Microsystems that allows Java programs to access and manipulate databases [8]. Open Database Connectivity (ODBC) is a standard from Microsoft for using different databases [9]. Object Oriented Database Management System (OODBMS) is an integration of OOP concepts and relational database concepts for complex data management services [14]. Objects can be directly persisted in an Object Oriented Database without any help from a third-party tool. Another approach is to use an Object Relational database that combines scalability and a support for complex data types [15].

1.2 Hibernate

Hibernate is a Java based open source ORM tool playing the role of a mediator between the database and OOP by mapping objects into a relational database and vice versa. This allows objects and their relationships to be made persistent in a relational database with minimal application programming. It can be used with any relational database which supports JDBC. The popularity of Hibernate is on the rise because of the following four reasons [3].

I The productivity of the system developer increases since the developer can focus on the logic of the system rather than worrying about how to persist and retrieve objects.

II Hibernate can be made to work with any database with a few changes in properties.

III Hibernate is free, and

IV The job market has a preference for job seekers with Hibernate knowledge.

Hibernate solves the identity problem by always returning the same reference to the object that is persisted by a row in the table. Hibernate solves the inheritance problem in four different ways which are discussed in detail in Section 2.3. Operators like *join*, *union* and multiple queries are used to extract the object of a right type from the database. The problem with association is handled by reading in the description of OOP design from the user. Joins are limited to being directional and have an orientation from a table (a class in a OOP design) to another table (another class in the OOP design) based on the description provided by the user. This limits the direction of navigation from one table to another.

Hibernate solves the impedance mismatch by using mapping files common to all applications in an enterprise. This allows the user to focus on the functionality of the program rather than the issues of how to persist objects in a relational database.

1.3 Problem Statement

Hibernate has been evolving ever since its conception. With the addition of new features comes the introduction of bugs in the code. Hibernate 3.2 has bugs varying from misusing of naming strategy [11] to not supporting reference to a property of an abstract class in a one-to-one association [12].

The objective of this project is to analyze and fix a bug (HHH-3095) present in Hibernate 3.2 which was reported on February 4, 2008. The bug - "Invalid queries when using subclasses and one-to-many associations" - arises from the way Hibernate treats a collection of objects that extends a base class [13]. An example is given in Figure 1.1 through Figure 1.4.

In the example, there are three classes - Cat, DomesticCat and Cage. A superclass Cat has a subclass DomesticCat. Each DomesticCat object belongs to a Cage object and a Cage object has a collection of DomesticCat objects. The association between the classes Cage and DomesticCat is bidirectional. The class Cat has one attribute, id. The class DomesticCat has one attribute, cage which is a reference to an object of the class Cage. The class Cage has two attributes, id and cats which is a collection of references to DomesticCat objects.

```java
package cat;

public class Cat {
  protected Long id;
  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }
}
```

Figure 1.1: Java class Cat as given in the bug report

```java
package cat;

public class DomesticCat extends Cat {
  protected Cage cage;
  public Cage getCage() {
    return cage;
  }

  public void setCage(Cage cage) {
    this.cage = cage;
  }
}
```

Figure 1.2: Java class DomesticCat as given in the bug report

```java
package cat;

import java.util.List;

public class Cage {
  protected Long id;
  protected List<DomesticCat> cats;
  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public List<DomesticCat> getCats() {
    return cats;
  }

  public void setCats(List<DomesticCat> cats) {
    this.cats = cats;
  }
}
```

Figure 1.3: Java class Cage as given in the bug report

```java
package cat;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class CatLoader {
  public static void main(String[] args) {
    Configuration cfg = new Configuration();
    cfg.configure(``hibernate.cfg.xml'');
    SessionFactory sessionFactory = cfg.buildSessionFactory();
    Session session = sessionFactory.getCurrentSession();
    session.beginTransaction();
    session.get(Cage.class, 1L);
    session.getTransaction().commit();
  }
}
```

Figure 1.4: Java class Catloader as given in the bug report

The following error is thrown when the class Catloader from Figure 1.4 is executed.

```
JDBCExceptionReporter:69 - could not initialize a collection:
[cat.Cage.cats\#1][select
cats0_.CAGE_ID as CAGE3_1_, cats0_.CAT_ID as CAT1_1_, cats0_.CAT_ID as CAT1_0_0_,
cats0_1_.CAGE_ID as CAGE2_2_0_
from CAT cats0_ inner join DCAT cats0_1_ on
cats0_.CAT_ID=cats0_1_.CAT_ID
where cats0_.CAGE_ID=?]
java.sql.SQLException: null, message from server:
"Unknown column in ‘cats0_.CAGE_ID’ ‘field list’ "
```

When the Cage object is accessed, the DomesticCat objects associated with the Cage object are also fetched. The query to fetch the associated DomesticCat objects from the database has an error because it has a column for Cat.Cage which does not exist in the table for Cat, but does exist in the table for DomesticCat.

The inheritance from Cat to DomesticCat is mapped using table per subclass which is discussed in Section 2.3, and the other subclasses of the class Cat, if added, are mapped using table per class hierarchy which is also explained in Section 2.3

At first, it seems the bug exists only when different inheritance mappings are used in a hierarchy. If a single inheritance mapping is used, the problem does not arise. This bug is important to fix because mixed inheritance is essential when it comes to mapping a large hierarchy to the database and performance is a key consideration. After a detailed study, the problem actually arises due to an irregularity in one-to-many association mappings in Hibernate. This project fixes the irregularity in a one-to-many association by taking into account the “table” attribute given in a mapping file which is ignored in the current Hibernate implementation.

CHAPTER TWO

Hibernate

2.1 Overview

Hibernate maps a Java class to a set of tables in the database. In order to do this for many applications, Hibernate needs to setup its environment. It requires a *configuration file* which provides information necessary to make database connections and a *mapping file* for the class. The Hibernate configuration file has an extension ".cfg.xml" which specifies the mapping files to use for all Java classes that need to be persisted. The file includes session variables such as the name of the database driver, the name of the database, the username, the password and boolean values for displaying SQL statements. The configuration file also contains the location of mapping file(s).

The mapping files have information on how a class is mapped to a table (or tables) in a database and how each attribute of a class maps to a column in the table(s). The mapping files have the extension ".hbm.xml".

There are other ways, like annotations and entity manager, in Hibernate to specify metadata for a transformation of information from relational database to OOP and vice-versa [7]. It should be noted here that this project does not deal with annotations and entity manager.

There are two ways to specify object persistence - a class may be persisted in either a single table or multiple tables. For example, consider a Java class Cat as shown in Figure 2.1 (which is also used in the project). Figure 2.2 and Figure 2.3 show the configuration file and the mapping file respectively for the class Cat.

```java
package project;

public class Cat {
  protected Long id;
  protected String breed;
  //getters and setters
  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getBreed() {
    return breed;
  }

  public void setBreed(String breed) {
    this.breed = breed;
  }
}
```

Figure 2.1: A Java class Cat

2.2 Retrieving Objects

One of the characteristics of Hibernate is that it has its own query language called Hibernate Query Language (HQL). HQL is similar to SQL but it is object oriented and deals with objects similar to the way SQL deals with tables. Hibernate supports queries written in both HQL and SQL [7].

A simple HQL query that fetches objects of a class Cat looks like - "from Cat". The queries "from Cat c" and "from Cat as c" are equivalent to "from Cat" but use an alias. HQL queries support inner join, left outer join, right outer join and full outer join. For example, consider the class Cat associated with a class House. To get all the Cats with their House object, the HQL query would be - "from Cat inner join House". For further details in HQL, refer to [4].

```
<! - - This specifies the xml version and other information regarding DTD.
- - In some editors this is auto generated.
- ->

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
   <session-factory>

      <!- - This part specifies all of the database specific attributes like
      - - password, username, the driver to use, etc
      - ->
      <property name="current_session_context_class"> thread </property>
      <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
      <property name="connection.url">jdbc:mysql://localhost/test_db_new</property>
      <property name="connection.password">password</property>
      <!- - SQL dialect - ->
      <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

      <!- - This specifies that the queries generated by Hibernate are to be displayed to
      - - standard output. This is for debugging purpose and should not be set for production.
      - ->
      <property name="show_sql">true</property>
      <property name="format_sql">true</property>

      <!- - mapping resources : all the mapping files are specified here - ->
      <mapping resource="Cat.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

Figure 2.2: A Hibernate configuration file

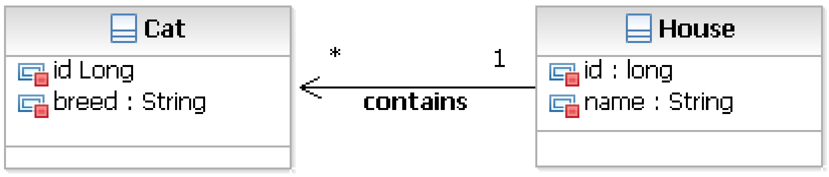
```
<!- - This specifies the xml version and Document Type Declaration (DTD). This remains same
- - for all hibernate mapping files. In some editors this is auto generated.
- ->

<?xml version="1.0" encoding="UTF-8">
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="project">
  <!- - It specifies a class named Cat is to be mapped to a table CAT. - ->
  <class name="Cat" table="CAT" >
    <!- - The data member "id" of class Cat is to be mapped to a column called "CAT_ID"
    - - and the ids are generated by the database itself in an increasing order. The id tag
    - - indicates that "CAT_ID" is a primary key for the table
    - ->
    <id name="id" column="CAT_ID" >
          <generator class="increment"/>
    </id>
    <!- - This specifies that the data member breed of the class Cat is to be mapped to a
    - - column called BREED. Note that the type attribute is optional. Hibernate can
    - - determines the type itself.
    - ->
    <property name = "breed" column = "BREED" />
  </class>
</hibernate-mapping>
```

Figure 2.3: A Hibernate mapping file for the class Cat in Figure 2.1

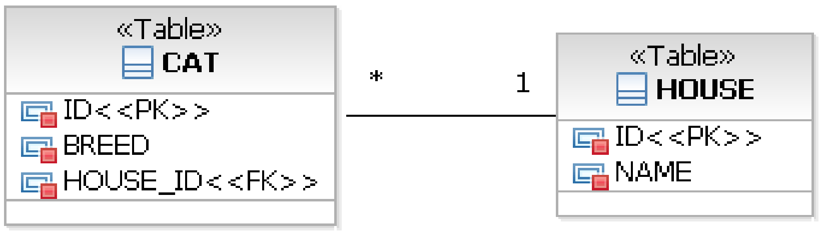
Hibernate Criteria Query is another approach to interact with objects in the relational database. It is a powerful API that has the functionality same as of the SQL and is completely object oriented. It is efficient to use for dynamic queries [17].

Fetching an object from the database involves a technique on how and when to retrieve the object and its associated objects. Hibernate uses the following four fetching techniques to retrieve objects and their associations. For illustrations of the fetching strategies, consider two classes - Cat and House - where the class House has a collection property of type Cat. The relationship is shown in the class diagram in Figure 2.4a and the corresponding tables in the database are shown in Figure 2.4b.

In *join fetching*, the objects and the associations are fetched by using a single *Select* statement and an *outer-join* (other types of joins can also be used but left outer join is the



(a)



(b)

Figure 2.4: (a)A simple class diagram, (b)Tables for classes in the class diagram

default). An SQL statement to extract an instance of class House with id = 1 for the class diagram in Figure 2.4a looks like the query shown below.

```
SELECT
  house.id, house.name, cat.id, cat.breed, cat.house_id
FROM
  house LEFT OUTER JOIN cat
ON
  house.id = cat.house_id
WHERE
  house.id = 1
```

Multiple Select statements are issued for fetching objects of a class and their corresponding associated objects in *select fetching*. In the example considered, a *Select* statement is first issued to fetch a House object with id = 1. Then after another Select statement is issued to fetch all the objects of Cat which are associated with the House object fetched by the previous Select statement. The queries issued are -

```
SELECT
  house.id, house.name
FROM
  house
WHERE
  house.id = 1

SELECT
  cat.id, cat.breed, cat.house_id
FROM
  cat
WHERE
  cat.house_id = 1
```

In *subselect fetching*, a second SELECT statement is issued to retrieve the collection for an entity (or all entities) retrieved from the previous fetch operation. The fourth type of fetching called *batch fetching*, fetches N collections (where N is specified by a user) for entities retrieved from the previous fetch.

Hibernate uses various techniques to decide when to fetch the associated objects. In *immediate fetching* or *eager fetching*, the associated objects are fetched as soon as the objects themselves are fetched. In *lazy fetching*, the collection or associated objects are fetched only when the operation on the collections is invoked. Lazy fetching helps in tuning the performance when there is a long chain of associations among classes and is the default fetching strategy. Another strategy called *extra-lazy fetching*, goes one step further than lazy fetching and fetches an object in the collection only as needed.

2.3 Inheritance Mapping in Hibernate

Inheritance allows one class (*subclass*) to have the same methods and data-members as another class (*superclass* or *extended* class) and tailor these inherited methods according to the subclass's need. The subclass can have its own properties that are not in its superclass. Inheritance is one of the impedance mismatch issues because relational databases do not have a concept of inheritance. Hibernate solves the problem of inheritance by using different mapping techniques for a hierarchy.

As stated in Section 2.1, a persistent class is mapped to a table in a database, according to the mapping files. If an inheritance exists between two classes then these two classes can be mapped in four different ways.

I Table per concrete class with implicit polymorphism.

II Table per concrete class

III Table per class hierarchy

IV Table per subclass

I **Table per concrete class with implicit polymorphism:** This is the most simple implementation. Under this strategy, there is one table for each instantiated class (it should be noted that abstract classes and interfaces do not need tables). The subclasses have tables which include all of the properties of their superclass. This implementation is good when the subclasses are queried often and when modification of the superclass in the future is unlikely. Since all of the attributes of a class including inherited attributes are placed in one table, only one table is accessed to persist the objects of a subclass. The attributes of the superclass are repeated in the tables of each subclass which makes it difficult to propagate the changes made in the attributes of the superclass in all of the subclasses' tables. If attributes are added or removed frequently from the superclass, the tables for the subclasses have to be changed frequently which is extra work. The XML mapping for this inheritance scheme requires replication of all inherited properties.

The example given in Figure 2.5 clarifies the concept. Figure 2.5 shows a class diagram where classes DomesticCat and ZooCat inherit from a class Cat. Figure A.1 and Figure A.2 show the Java classes - DomesticCat and ZooCat. The Java class Cat is same as in Figure 2.1. Figure 2.6 shows the database with three tables - one for each class - and the subclass's table includes all of the properties of the superclass. Here it can be seen that a query against "Cat" must be executed as several queries against each subclass. For example, to get all the objects of the class Cat, the following three queries are executed.

```
SELECT * FROM CAT;
SELECT * FROM ZCAT;
SELECT * FROM DCAT;
```

On the other hand a query against a subclass (ZooCat or DomesticCat) is executed as a single query.

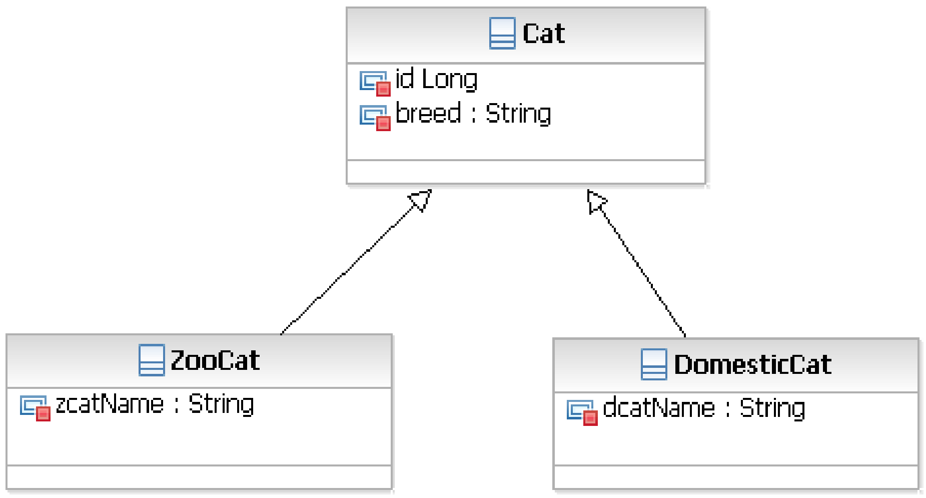


Figure 2.5: A class diagram indicating inheritance

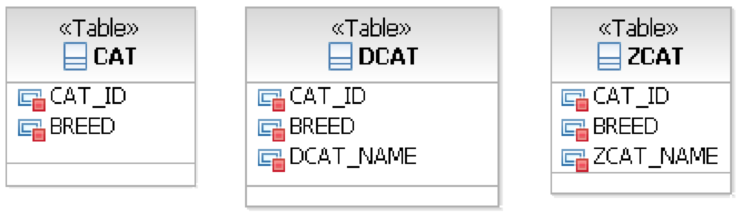


Figure 2.6: Tables in database for the class diagram in Figure 2.5 using "Table per concrete class with implicit polymorphism"

II **Table per concrete class:** This method results in the same tables as does "Table per concrete class with implicit polymorphism". However, it avoids the redundant

XML code as shown in Figure A.5 that shows the XML mappings for the class diagram shown in Figure 2.5. The query against a superclass is executed as a single query where the tables for subclasses are combined using a *UNION* operator in SQL. For example, to retrieve all instances of the class Cat, the following query is executed.

```
SELECT
  CAT_ID, BREED, DCAT_NAME, ZCAT_NAME
FROM(
  SELECT
    CAT_ID, BREED, null as DCAT_NAME, null as ZCAT_NAME
  FROM
    CAT
  UNION
  SELECT
    CAT_ID, BREED, DCAT_NAME, null as ZCAT_NAME
  FROM
    DCAT
  UNION
  SELECT
    CAT_ID, BREED, null as DCAT_NAME, ZCAT_NAME
  FROM
    ZCAT)
```

Only one query is executed in this inheritance mapping compared to three queries in “Table per concrete class with implicit polymorphism”. “Table per concrete class” is preferred to “Table per concrete class with implicit polymorphism” because the mapping file is shorter without redundant information.

III **Table per class hierarchy:** This implementation makes use of a discriminator - a value that determines the subclass of an instance. It maps the entire hierarchy to one table. The table includes columns for all properties of all classes in the hierarchy and a column for the discriminator. The table is much larger than in the other cases because the table stores all objects of all subclasses. This mapping provides good performance because insertion or retrieval of an instance of a superclass or subclass involves only one table. However, it has the drawback of requiring null values for the properties belonging to only one subclass. For the class diagram in Figure 2.5, the classes are mapped to the table shown in Figure 2.7. The table CAT has an extra attribute - CAT_TYPE which is the discriminator. The XML mappings are shown in Figure A.6 through Figure A.8. All instances of DomesticCat will have a value "D" for CAT_TYP as stated in the mapping of class DomesticCat and all instances of ZooCat will have a value of "Z" for CAT_TYP as stated in the mapping of class ZooCat.

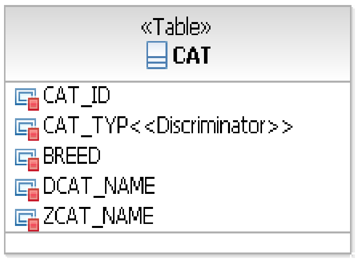


Figure 2.7: A table in database for the class diagram in Figure 2.5 using "Table per class hierarchy"

IV **Table per subclass:** In this strategy, each class (including abstract classes and interfaces) has its own table. The table contains columns for the properties that belong to the classes mapped to them and do not have columns for inherited properties. It uses foreign key constraints with associated tables to reference all inherited attributes. The primary key of the subclass table has a foreign key constraint with the primary key of the superclass table. The main benefit of using this mapping technique is that the tables in the database are normalized. This scheme is unacceptable when the chain of references requires too many joins even though Hibernate allows it. For the class diagram in Figure 2.5, the classes are mapped to a table as shown in Figure 2.8. The attributes DCAT.CAT_ID and ZCAT.CAT_ID, foreign keys, reference CAT.CAT_ID.

These strategies can be intermixed. The use of different inheritance mappings used within a class hierarchy tunes the performance of an application. For example, consider a case where there are four levels of the class hierarchy and each class has two subclasses

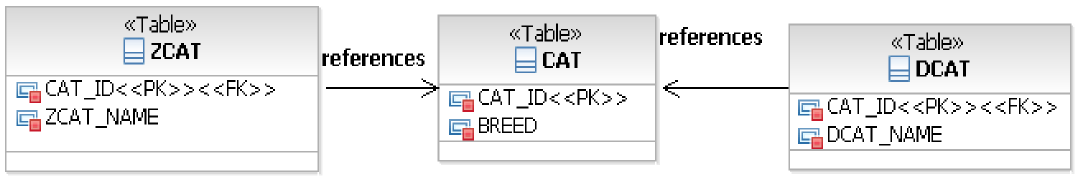


Figure 2.8: Tables in database for the class diagram in Figure 2.5 using “Table per subclass”

(which is not uncommon in a large system), the hierarchy contains fifteen classes. For efficiency, a user may want to map some classes into their own table but putting all of the other classes into a single table.

2.4 Collections in Hibernate:

A collection is a group of objects. Hibernate requires that persistent collection-valued fields are declared as an interface type in Java. A collection of objects can be of two types - collection of value type and a collection of entity references. Collections of value-type are persisted when an owner of the collection is persisted and deleted when the owner is deleted. To store a collection of value-type Hibernate requires a collection table. This collection table is hidden from the user. The references to objects, which form a collection, are preserved in the collection table.

Hibernate supports collections like set (and sorted set), map (and sorted map), bag, idbag, list and array (no longer used). Among these, bag and idbag do not have implementations in Java. Hibernate also allows users to implement their own collection interface.

For the collection of entity type, concepts of multiplicity and directionality are required. There are four types of multiplicity - One to Many, Many to One, One to One and Many to Many. Out of these multiplicities, One To Many and Many to Many form a collection. The relationship between two classes A and B are said to be bidirectional when both the classes have an association with each other, i.e. the class A has class B as its data member and the class B has the class A as its data member. The relationship between these two classes are said to be unidirectional when only of these classes has the other as its data member i.e. either the class A has the class B as its data member or the class B has the class A as its data member.

Consider the mapping given in Figure 2.9 where an instance of a class Cat is associated with an instance of a class House and an instance of a class House is associated with numerous instances of a class Cat. Each of these classes are mapped as an entity. The association is a bi-directional, one-to-many (from House to Cat) and the collection is of entity type.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="project">
  <class name="Cat" table="CAT">
    <id name="id" column="CAT_ID" >
      <generator class="increment"/>
    </id>
    <many-to-one name="house" class="House" column="HOUSE_ID">
  </class>

  <class name="House" table="HOUSE">
    <id name="id" column="HOUSE_ID" >
      <generator class="increment"/>
    </id>
    <set role="animals" inverse="true">
      <key column = "HOUSE_ID" />
      <one-to-many class="Cat">
    </set>
  </class>
</hibernate-mapping>
```

Figure 2.9: An XML mapping for a bidirectional one-to-many relation (from House to Cat)

CHAPTER THREE

Project Development

3.1 Problem Definition

As mentioned in Section 1.3, there is an irregularity in how Hibernate handles one-to-many associations. Bug HHH-3095 has mixed inheritance with a one-to-many association. The problem arises because the one-to-many type collection uses a one-to-many construct for specifying the collection (which seems reasonable). In the bug HHH-3095, to achieve mixed inheritance the class DomesticCat is persisted across another table (so that the class DomesticCat is mapped using a table per subclass) along with the table for the class Cat. Within Hibernate, mapping of a class on multiple tables can generate an error when there is a one-to-many collection.

Mapping a class across multiple tables is not uncommon. Attributes of the class can be scattered across multiple database tables. Spreading the attributes of a class in multiple tables has a benefit when the class is a part of a collection, which can be empty thereby avoiding the null values. This is especially true for a one-to-many association where a collection property is stored in the table for a class that makes up the collection. Bug HHH-3095 involves inheritances among classes which are mapped using “Table per class hierarchy” and “Table per subclass”, thus the class DomesticCat is persisted across two different tables.

To study the problem in detail, consider four classes - Cat, DomesticCat, ZooCat and House as shown in Figure 3.1. The superclass Cat has two subclasses - DomesticCat and ZooCat. Each DomesticCat object belongs to a House object. All of these classes are mapped as an entity. The association depicted here is bidirectional, many-to-one (from House to DomesticCat). The collection of DomesticCat in the class House is an entity collection.

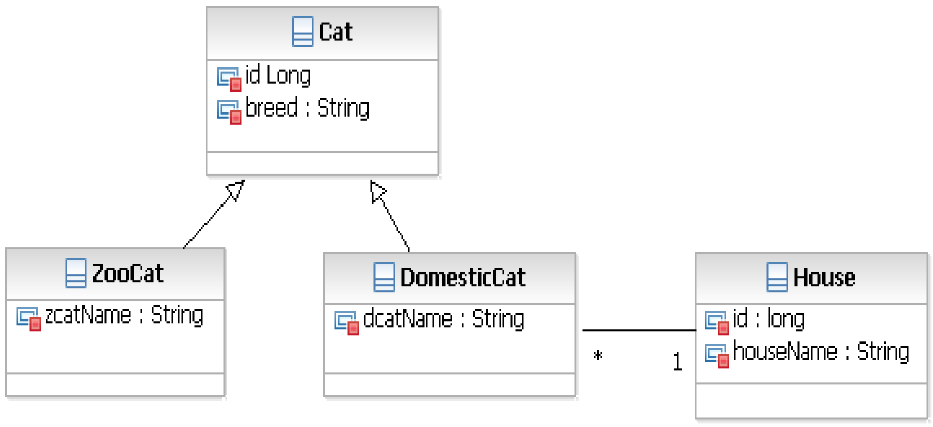


Figure 3.1: A class diagram used for the project

To find out the point of the problem, three programs are written that map inheritance among the classes in Figure 3.1 in three different ways - table per class hierarchy, table per subclass and a mixed implementation (which uses table per class hierarchy for one subclass and table per hierarchy for another subclass). Each implementation creates a list of DomesticCat objects for a House object. The Java classes in each of these implementations are the same and are listed in the Appendix. The only differences are the Hibernate mapping files and the configuration file. From this point onwards, the table per subclass implementation of the problem will be referred as *query program*, the table per class hierarchy implementation of the problem as *discriminator program* and the mixed implementation of the problem as *error program*. As its name implies, the error program demonstrates bug HHH-3095 [13]. In fact, it terminates with an SQL error.

The Java code for classes - Cat and ZooCat are the same as shown in Figure 2.1 and Figure A.2 respectively. Since the class DomesticCat has an association with the class House, the Java classes for DomesticCat and House are modified to include the association. The code for these classes are shown in Figure A.12 and Figure A.13 along with the code for the driver (named CatLoader.java) that creates and persists objects in the database, which is in Figure A.14.

These three programs are executed and each of them generates three SQL queries. The HQL query, which is same for all of these three programs, is given below:

“from DomesticCat dc order by dc.id asc”

The objective is to retrieve all the objects of DomesticCat from the database which are sorted in an ascending order by their id fields.

The reason for creating the three different programs is the bug involves mixed inheritance as well as a one-to-many association. The original bug code has “table per subclass” and “table per class hierarchy” implementations. Hence, three different programs are developed - two for the two implementations mentioned and one that has both of these implementations.

3.1.1 Query Program

The query program works properly. Figure 3.2 shows the relationships between tables after the classes in Figure 3.1 are mapped into a database in the query program.

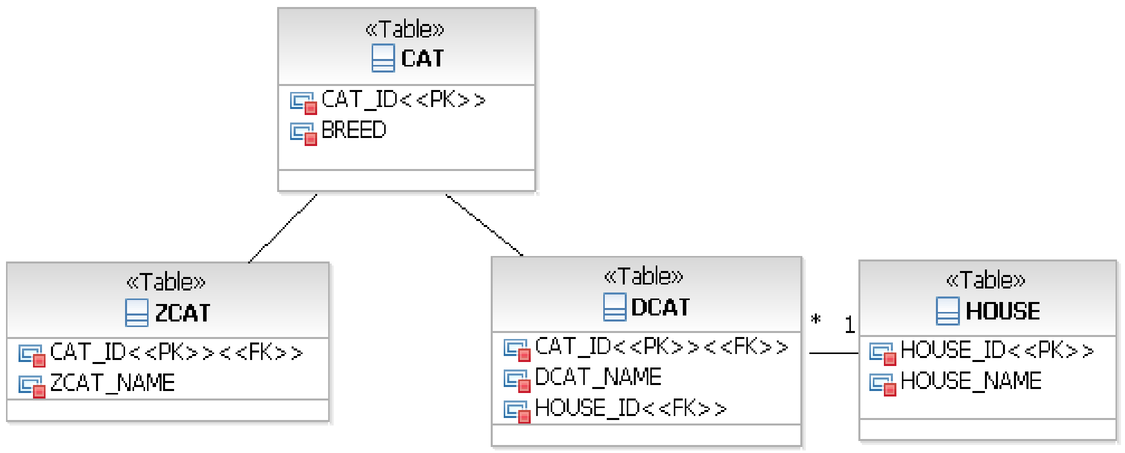


Figure 3.2: Table mappings for the class diagram in Figure 3.1 in query program

The XML mapping for the class Cat and the class ZooCat are the same as shown in the Figure A.9 and Figure A.11 respectively. The XML mapping for the class DomesticCat and the class House are as shown in Figure 3.3 and Figure 3.4 respectively.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="cat">
  <joined-subclass name="DomesticCat" extends="Cat" table="DCAT">
    <key column="CAT_ID"/>
    <property name = "dcatName" column = "DCAT_NAME" />
    <many-to-one name="house" column="HOUSE_ID" lazy="false" cascade="none"
    not-null="false"/>
  </joined-subclass>
</hibernate-mapping>
```

Figure 3.3: An XML mapping for the class DomesticCat in query program

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="cat">
  <class name="House" table="HOUSE">
    <id name="id" column="HOUSE_ID" >
      <generator class="increment"/>
    </id>
    <property name="houseName" column="HOUSE_NAME"/>
    <bag name="animals" lazy ="false" cascade="none" inverse="true">
      <key column="HOUSE_ID" not-null="false"/>
      <one-to-many class="DomesticCat"/>
    </bag>
  </class>
</hibernate-mapping>
```

Figure 3.4: An XML mapping for the class House used in the project

Figure 3.5 shows the log generated in the query program. The SQL queries executed in the query program are also explained.

Snippets of log generated for the query program.

```
1  INFO: Hibernate 3.2.6
2  INFO: Reading mappings from resource : cat/Cat.hbm.xml
3  INFO: Mapping class: cat.Cat -> CAT
4  INFO: Reading mappings from resource : cat/DomesticCat.hbm.xml
5  INFO: Mapping joined-subclass: cat.DomesticCat -> DCAT
6  INFO: Reading mappings from resource : cat/ZooCat.hbm.xml
7  INFO: Mapping joined-subclass: cat.ZooCat -> ZCAT
8  INFO: Reading mappings from resource : cat/House.hbm.xml
9  INFO: Mapping class: cat.House -> HOUSE
10 INFO: Configured SessionFactory: null
11 INFO: Mapping collection: cat.House.animals -> DCAT
```

Figure 3.5: Snippets of log generated for a query program

From the log generated in Figure 3.5 it can be seen that DomesticCat is mapped to a table DCAT on line 5. The class House's collection of DomesticCat is also mapped to the table DCAT on line 11. In order to retrieve the collection of DomesticCat objects, Hibernate queries the table DCAT.

First Hibernate tries to pull all the instances of DomesticCat from the table DCAT. The table DCAT persists the property-values of DomesticCat that are not inherited. To get the inherited property-values, Hibernate needs to access the table CAT. Since Cat-subclass inheritance is mapped using "Table per subclass", Query 1 has to perform a join on the primary keys of CAT and DCAT.

Query 1

```
SELECT
    domesticca0_.CAT_ID as CAT1_0_,
    domesticca0_1_.BREED as BREED0_,
    domesticca0_.DCAT_NAME as DCAT2_1_,
    domesticca0_.HOUSE_ID as HOUSE3_1_
FROM
    DCAT domesticca0_
INNER JOIN
    CAT domesticca0_1_
ON
    domesticca0_.CAT_ID=domesticca0_1_.CAT_ID
ORDER BY
    domesticca0_.CAT_ID ASC
```

The class DomesticCat has a data member of type House and non-lazy fetching is used, so to complete the DomesticCat instances pulled from Query 1, Hibernate has to pull their corresponding House objects. This is what Query 2 does where the placeholder (?) is filled by domesticca0_.HOUSE_ID extracted from the resultset of Query 1.

Query 2

```
SELECT
   house0_.HOUSE_ID as HOUSE1_3_0_,
   house0_.HOUSE_NAME as HOUSE2_3_0_
FROM
   HOUSE house0_
WHERE
   house0_.HOUSEID=?
```

Since there is a two way navigability between the tables DCAT and HOUSE and non-lazy fetching is done, the House instances resulting from the execution of Query 2 needs to find their corresponding DomesticCat objects. So for each House object, Hibernate executes Query 3 where the placeholders are replaced by house0_.HOUSE_ID from the resultset of Query 2.

Query 3

```
SELECT
   animals0_.HOUSE_ID as HOUSE3_1_,
   animals0_.CAT_ID as CAT1_1_,
   animals0_.CAT_ID as CAT1_0_0_,
   animals0_1_.BREED as BREED0_0_,
   animals0_.DCAT_NAME as DCAT2_1_0_,
   animals0_.HOUSE_ID as HOUSE3_1_0_
FROM
   DCAT animals0_
INNER JOIN
   CAT animals0_1_
ON
   animals0_.CAT_ID=animals0_1_.CAT_ID
WHERE
   animals0_.HOUSE_ID=?
```

3.1.2 *Discriminator Program*

Figure 3.6 shows the relationships between tables after they are mapped into a database in the discriminator program.

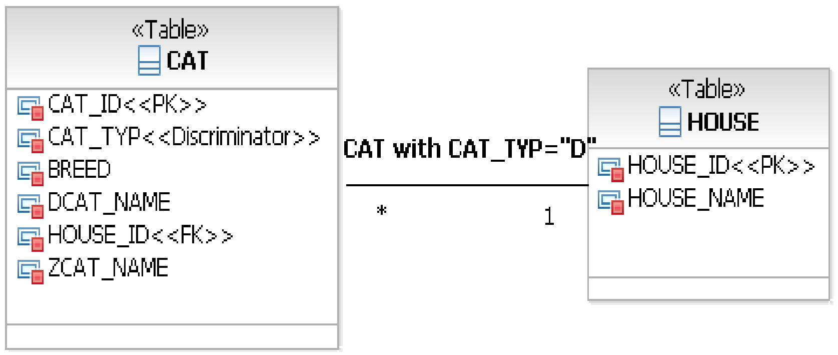


Figure 3.6: Table mappings for the class diagram in Figure 3.1 in the discriminator program

The XML mapping for the classes Cat, ZooCat and House are the same as shown in the Figure A.6, Figure A.8 and Figure 3.4 respectively. The XML mapping for the class DomesticCat is shown in Figure 3.7.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="cat">
  <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
    <property name = "dcatName" column = "DCAT_NAME" />
    <many-to-one name="house" column="HOUSE_ID" lazy="false" cascade="none"
    not-null="true"/>
  </subclass>
</hibernate-mapping>
```

Figure 3.7: An XML mapping for the class DomesticCat in the discriminator program

The discriminator program is executed with the resulting logs (Figure 3.8) and the queries.

Snippets of logs generated and SQL queries for the discriminator program.

```
 1 INFO: Hibernate 3.2.6
 2 INFO: Reading mappings from resource : cat/Cat.hbm.xml
 3 INFO: Mapping class: cat.Cat -> CAT
 4 INFO: Reading mappings from resource : cat/DomesticCat.hbm.xml
 5 INFO: Mapping subclass: cat.DomesticCat -> CAT
 6 INFO: Reading mappings from resource : cat/ZooCat.hbm.xml
 7 INFO: Mapping subclass: cat.ZooCat -> CAT
 8 INFO: Reading mappings from resource
 9 INFO: Mapping class: cat.House -> HOUSE
10 INFO: Mapping collection: cat.House.animals -> CAT
```

Figure 3.8: Snippets of log generated in the discriminator program

From the log in Figure 3.8, it can be seen that the classes Cat, DomesticCat and ZooCat are mapped to one table, CAT, on line 3, line 5 and line 7 respectively. Since DomesticCat is mapped to the table CAT, the class House's collection of DomesticCat is mapped to the table CAT on line 10.

The classes DomesticCat and ZooCat are mapped to the table CAT using discriminators, so Hibernate tries to select Cat objects with discriminators set to 'D' (which was set in the mapping file for DomesticCat). This gives a list of DomesticCat from the table CAT.

Query 4

```
SELECT
    domesticca0_.CAT_ID as CAT1_0_,
    domesticca0_.BREED as BREED0_,
    domesticca0_.DCAT_NAME as DCAT4_0_,
    domesticca0_.HOUSE_ID as HOUSE5_0_
FROM
    CAT domesticca0_
WHERE
    domesticca0_.CAT_TYP='D'
ORDER BY
    domesticca0_.CAT_ID ASC
```

The DomesticCat class has a data member of type House and non-lazy fetching is used, so to complete the DomesticCat instances pulled from Query 4, Hibernate has to pull their corresponding House objects. This is what Query 5 does where the placeholder is filled by domesticca0_.HOUSE_ID extracted from the resultset of Query 4.

Query 5

```
SELECT
    house0_.HOUSE_ID as HOUSE1_1_0_,
    house0_.HOUSE_NAME as HOUSE2_1_0_
FROM
    HOUSE house0_
WHERE
    house0_.HOUSE_ID=?
```

Since there is a two way navigability between the tables DCAT and HOUSE, and non-lazy fetching is used, the House instances resulting from the execution of Query 5 need to find their corresponding DomesticCat objects. So for each House objects, Hibernate executes Query 6. Since all DomesticCat objects are persisted in the table CAT, the table CAT is used in a query.

Query 6

```
SELECT
    animals0_.HOUSE_ID as HOUSE5_1_,
    animals0_.CAT_ID as CAT1_1_,
    animals0_.CAT_ID as CAT1_0_0_,
    animals0_.BREED as BREED0_0_,
    animals0_.DCAT_NAME as DCAT4_0_0_,
    animals0_.HOUSE_ID as HOUSE5_0_0_
FROM
    CAT animals0_
WHERE
    animals0_.HOUSE_ID=?
```

3.1.3 Error Program

Figure 3.9 shows the relationships between tables after they are mapped into the database in the error program. The inheritance mapping used in the error program is a combination of "Table per class hierarchy" and "Table per subclass". The subclass ZooCat and the superclass Cat are mapped using "Table per class hierarchy" and the inheritance between the subclass DomesticCat and the superclass Cat is mapped using "Table per subclass".

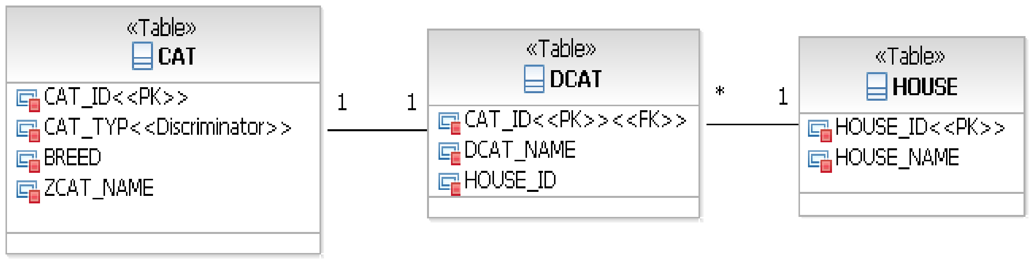


Figure 3.9: Table mappings for the class diagram in Figure 3.1 in the error program

The XML mapping for the classes Cat and ZooCat are the same as shown in Figure A.6 and Figure A.8 respectively. The mapping for DomesticCat differs from the mapping shown in Figure A.7 in that it has an additional line specifying that the inheritance with Cat is mapped with "Table per subclass". This is done by inserting a tag <*join*> which is not present in the mapping in Figure A.7 as can be seen in Figure 3.10. The XML mapping for the class House is unchanged, and can be found in Figure 3.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="cat">
  <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
    <join table="DCAT">
      <key column="CAT_ID"/>
      <property name = "dcatName" column ="DCAT_NAME" type = "string"/>
      <many-to-one name="house" column="HOUSE_ID" lazy="false" cascade="none"
      not-null="false"/>
    </join>
  </subclass>
</hibernate-mapping>
```

Figure 3.10: An XML mapping for the class DomesticCat in the error program

Following are the SQL queries generated along with the logs generated, Figure 3.11, when CatLoader.java is executed in the error program. Only the snippets of the logs generated, when mapping the classes and the collections to the tables, are shown here. The SQL queries are generated when processing the HQL query and accessing the objects loaded from the database.

Snippets of logs generated for the error program.

```
1  INFO: Hibernate 3.2.6
2  INFO: Reading mappings from resource : cat/Cat.hbm.xml
3  INFO: Mapping class: cat.Cat -> CAT
4  INFO: Reading mappings from resource : cat/DomesticCat.hbm.xml
5  INFO: Mapping subclass: cat.DomesticCat -> CAT
6  INFO: Mapping class join: cat.DomesticCat -> DCAT
7  INFO: Reading mappings from resource : cat/ZooCat.hbm.xml
8  INFO: Mapping subclass: cat.ZooCat -> CAT
9  INFO: Reading mappings from resource : cat/House.hbm.xml
10 INFO: Mapping class: cat.House -> HOUSE
11 INFO: Mapping collection: cat.House.animals -> CAT
```

Figure 3.11: Snippets of logs generated in the error program

From the logs in Figure 3.11, it can be seen that line 3 maps the class Cat to a table CAT. On line 5, the class DomesticCat is mapped as a subclass of the class CAT and is mapped to the table CAT. According to the mapping file, the attributes of the class DomesticCat are persisted in two tables - CAT and DCAT using a *join* tag. Hence, there exists a join between the tables CAT and DCAT. This is defined on line 6, by adding a join of the class DomesticCat and that join is mapped to the table DCAT. Adding a join means that the table DCAT should be considered while fetching objects of DomesticCat. Since the class DomesticCat is mapped to the table CAT (line 5) and the join for the class DomesticCat is mapped to the table DCAT, any collection of DomesticCat is by default mapped to the table CAT. This is seen on line 11 of the logs. The property *animals* of the class House is a collection of DomesticCat objects and hence this collection is mapped to the table for DomesticCat, CAT. The table CAT persists some of the attributes of the class DomesticCat but it does not persist the attribute “house” (which is persisted in the table for join i.e. DCAT). This is the original source of the error since getting the instances of a collection causes Hibernate to query the table CAT which does not contain all of the properties in the class DomesticCat. The mapping of the class DomesticCat to the table CAT is consistent with the fact that Cat.hbm.xml includes a discriminator property and the DomesticCat.hbm.xml also has a value “D” for the discriminator.

For collecting instances of DomesticCat, the table to which the class DomesticCat is mapped is queried i.e. the table CAT. Since the classes Cat, ZooCat and DomesticCat are mapped using discriminators, Hibernate tries to select Cat objects with discriminators set to 'D' (which was set in the mapping file for DomesticCat) in order to find the persisted DomesticCat objects. Since DomesticCat has some of its attributes mapped in a join table, DCAT, the *from* clause in Query 7 uses join on CAT and DCAT. Query 7 is correct and fetches instances of DomesticCat.

Query 7

```
SELECT
    domesticca0_.CAT_ID as CAT1_0_,
    domesticca0_.BREED as BREED0_,
    domesticca0_1_.DCAT_NAME as DCAT2_1_,
    domesticca0_1_.HOUSE_ID as HOUSE3_1_
FROM
    CAT domesticca0_
INNER JOIN
    DCAT domesticca0_1_
ON
    domesticca0_.CAT_ID=domesticca0_1_.CAT_ID
WHERE
    domesticca0_.CAT_TYP='D'
ORDER BY
    domesticca0_.CAT_ID ASC
```

The DomesticCat class has a data member of type House and non-lazy fetching is used, so to complete the DomesticCat instances pulled from Query 7, Hibernate has to pull their corresponding House objects. This is what Query 8 does where the placeholder is filled by domesticca0_1_.HOUSE_ID extracted from the resultset of Query 7. Query 8 is also correct.

Query 8

```
SELECT
    house0_.HOUSE_ID as HOUSE1_2_0_,
    house0_.HOUSE_NAME as HOUSE2_2_0_
FROM
    HOUSE house0_
WHERE
    house0_.HOUSE_ID=?
```

Since there is a two-way navigability between the class DomesticCat and the class House, and the non-lazy fetching is used, the House instances resulting from the execution of Query 8 must find their corresponding DomesticCat objects. So for each House objects, Hibernate executes Query 9. It is this query that crashes the program as it can be seen it is trying to extract the field HOUSE_ID from the table CAT (which does not have that field). The errors are present in the *select* clause and the *where* clause which are highlighted in Query 9.

Query 9

```
SELECT
    animals0_.HOUSE_ID as HOUSE5_1_,
    animals0_.CAT_ID as CAT1_1_,
    animals0_.CAT_ID as CAT1_0_0_,
    animals0_.BREED as BREED0_0_,
    animals0_1_.DCAT_NAME as DCAT2_1_0_,
    animals0_1_.HOUSE_ID as HOUSE3_1_0_
FROM
    CAT animals0_
INNER JOIN
    DCAT animals0_1_
ON
    animals0_.CAT_ID=animals0_1_.CAT_ID
WHERE
    animals0_.HOUSE_ID=?
```

3.2 Original Attempts to Solve the Bug

Determining the exact point of the problem is challenging. Prior to using the approach in Section 3.4, two different approaches failed to pass all test cases. The first approach is pertinent to inheritance. The original Java classes specified in the bug [13] use inheritance and an error occurs when mixed inheritance is used (Section 3.1). When only one particular type of inheritance mapping is used, no error occurs. This induced a mindset that Hibernate has a faulty way of handling collections when mixed inheritance mapping is used and hence, generates a wrong alias for a subclass in an SQL statement. A fix, which will be called Modification1 hereupon, modified the alias generation for tables by identifying their correct table number. The method that generates an alias for a table retrieves the column name of all of the properties possessed by a class and the corresponding number for each of these column names, both of which are set by Hibernate while parsing the mapping files. These numbers correspond to the join table numbers (this is how Hibernate generates alias). The join table number for a table is assigned while reading a mapping file and is assigned based on the depth of the nesting of a *join* tag associated with that table. This approach did not work with an unidirectional one-to-many association since a class that persists a collection may not have a property (that is related to a collection) and hence, no column name exists for the collection property in the class that persists the collection.

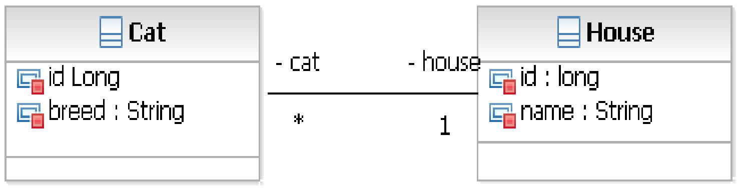
Another approach, which will be called Modification2 hereupon, considers the joins for a table of a given class. This approach, like the previous one, modifies the alias generation method for a table however, in a different way. It receives an alias generated for a table by the original Hibernate and the tablename, where the collection is persisted, as its arguments. It finds the class that persists the collection and iterates through an array of tablenames, called spaces, to find whether the argument tablename matches any tablename in the spaces. If the spaces have a tablename, then it adds a suffix to the alias (passed to the method as one of its arguments). The suffix thus added is a position of the tablename in the spaces. If the tablename is not included in the spaces array then the alias passed as an argument to the method is returned unchanged. This approach also does not work with a unidirectional association.

Both Modification1 and Modification2 produced correct result for the code given in Section 1.3. Hibernate test cases are run on Modification1 with a number of errors of 825 (out of 1273). It was not evident outright that the problem was with unidirectional one-to-many collection. Stepping through the original program code (Section 1.3) under Hibernate gave an understanding of Hibernate variables - joins, spaces - which led to the development of Modification2. Hibernate test cases are run on Modification2 with a number of errors of 49 (out of 1273). The testing result of Modification2 was an improvement over that of Modification1. However, Modification2 was behind Hibernate which has 39 testing errors. To find the exact point of a problem, failed test cases were separated out and were stepped through each line of code (one failed test case at a time) on both Hibernate and

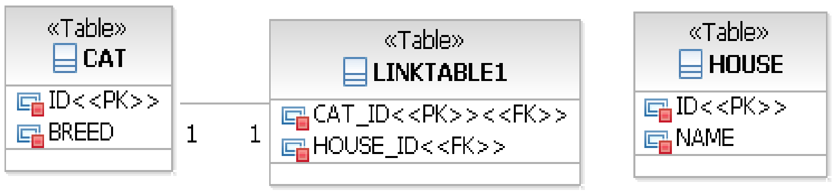
Modification2 simultaneously. After much strenuous work of going through each line of code (for each test case) and understanding changes made to Hibernate variables in the process of stepping, the problem became obvious. A table that persists the class (that makes up the collection) is set as the table for the collection and no joins are considered even when the collection is actually persisted on some join tables.

3.3 Root Cause of the Bug

In the original Hibernate code, if there is a one-to-many association between two classes, the collection is persisted on the class that makes up the collection. For example, consider a bidirectional association between the classes Cat and House as shown in Figure 3.12a. Since the class Cat forms a collection for the class House, the table on which the class Cat is persisted is responsible for persisting the collection as well. The corresponding tables in the database are shown in Figure 3.12b.



(a)



(b)

Figure 3.12: (a)Many-to-one association between Cat and House, (b)the class Cat persisted on multiple tables

To collect all of the Cat objects associated with a particular instance of the class House, tables LINKTABLE1 and CAT should be joined to get all such instances. Likewise to get an instance of the class House associated with a particular instance of the class Cat, the tables CAT and LINKTABLE1 must be joined.

The original Hibernate code works correctly with a one-to-many association when the class Cat is persisted in a single table. If the class Cat shown in Figure 3.12a is persisted in *multiple tables* as shown in the Figure 3.12b, then it becomes essential to specify the exact table where the collection is persisted. The problem arises due to the fact that it is NOT required in Hibernate to specify the name of the table for collections in a one-to-many association. In fact, any such table specification is IGNORED. In the error program, under the original Hibernate code, the class DomesticCat is persisted in multiple tables - CAT and DCAT as shown in Figure 3.10. Only the table DCAT has a column that persists the collection. However, Hibernate assigns the table CAT as the collection table and since the table attribute is ignored in processing one-to-many collections, it is impossible for the user to specify the correct table DCAT. This project extends Hibernate by allowing a user to specify tables for one-to-many associations so that a user can explicitly specify the table to use for the collection.

3.3.1 Hibernate Collection Processing

The Hibernate object which is responsible for storing a proper table for a collection (and hence resulting in an incorrect tablename as mentioned in the previous paragraph) is *collection* which is an instance of the class "org.hibernate.mapping.Collection". The object "collection" stores information specified for a collection by a mapping file. Two variables of the object collection - *collection.element* and *collection.collectionTable* - hold key information. The variable collection.element stores information regarding the type of the association (e.g. one-to-many, many-to-many), the strategy to fetch the collection, the referenced class (the class that makes up the collection), the referenced property of the referenced class, etc. The other variable, collection.collectionTable, stores a reference to the table that persists the collection.

Hibernate treats a collection in two passes. In the first pass, Hibernate sets up the attributes of a collection as given in the mapping file. A specification for a collection in a mapping file has attributes like lazy fetching, user provided SQL queries, etc. These attributes are read and appropriate Hibernate variables are set in the first pass. If a collection is many-to-many, then the collection table is also set, but if the collection is one-to-many, the collection table is set in the second pass. The reason for setting the collection table for a one-to-many association in the second pass is that during the first pass not all the classes have been mapped and hence, resolving references to a class (that has not yet been mapped) is not possible. In the second pass, the collection table (if a collection is one-to-many) is set and references to classes that makes up a collection are resolved. Consider the example construct for one-to-many given in Figure 3.13.

```
<set name="persons" lazy="false" cascade="none"
     table = "HOUSE_PERSON" fetch="select">
  <key column="HOUSE_ID" not-null="false"/>
  <one-to-many class="Person" />
</set>
```

Figure 3.13: An example of one-to-many construct

In the example given in Figure 3.13, the collection type used is a "set". The attribute *name*, specifies the name of the collection which is "persons" in this case. The attribute *cascade* specifies whether to enable operations to cascade to child entities or not. Its value of none means there is no cascading operation. The attribute *fetch* here specifies that the collection should be fetched by issuing a select statement. The tag *key* specifies that the column "HOUSE_ID" is a foreign key in the collection table. The tag *one-to-many* specifies that the collection consists of objects of the class "Person". The attribute *table* specifies the table used to store the collection. In this example, the original Hibernate code would by default set the collection table to be the table on which the class Person is persisted instead of the table "HOUSE_PERSON" as specified by the table attribute.

3.3.2 Hibernate Workaround Processing

Hibernate has a workaround for a one-to-many association when a collection is persisted on a table other than the one where the class that makes up the collection is persisted. The workaround requires using a many-to-many construct with a unique constraint set to true. By setting the unique attribute to true, Hibernate does not allow any instance of a class, that makes up the collection, to be associated with more than one instance of a class that owns the collection. This in effect imposes a one-to-many constraint in a many-to-many association. Note that the many-to-many construct used to represent the one-to-many collection must have the *unique* attribute set to true. The unique attribute is set to false by default.

Consider the one-to-many construct shown in Figure 3.13. The workaround for using a different table for collection other than the default one is shown in Figure 3.14. The unique attribute is set to true and the collection table will be set to the one specified in the many-to-many construct. In this example, collection table is set to *HOUSE_PERSON* and the column *PERSON_ID* in the table HOUSE_PERSON is unique.

```
<set name="persons" lazy="false" cascade="none"
     table = "HOUSE_PERSON" fetch="select">
  <key column="HOUSE_ID" not-null="false"/>
  <many-to-many class="Person" column="PERSON_ID" unique="true" />
</set>
```

Figure 3.14: An example of many-to-many construct with *unique* attribute set to true

3.4 Fix to the Bug

This project extends the capability of Hibernate to consider the table attribute for one-to-many associations. The new implementation considers the table attribute and follows the execution path of many-to-many with a unique constraint, which is described in Section 3.3.2. The tablename declared in the mapping file is stored in a new Hibernate variable called *explicitTableName*, while the default used by the original Hibernate is stored in a new variable *implicitTableName*. These local Java variables are added to modified Hibernate code in the method bindCollectionSecondPass() of the class org.hibernate.cfg.HbmBinder.

To correctly integrate the user specified table into Hibernate, there are three cases to consider. They are -

I The table attribute is absent, hence the value of explicitTableName is null.

II The table attribute is present, but the value of explicitTableName and the value of implicitTableName are the same.

III The table attribute is present and, the value of explicitTableName and the value of implicitTableName are different.

It is already mentioned in Section 3.3.1 that a collection is processed in two passes. In all of these cases, modified Hibernate code treats a one-to-many collection as one-to-many in the first pass and reads attributes specified in the collection construct into a node. The value of the variable collection.element (Section 3.3) is set to OneToMany. In the second pass, the modified Hibernate code determines which of the three cases exists. The variable explicitTableName is set as the second pass for a collection is started. In the method bindCollectionSecondPass() the value of the attribute table from a node, which was previously populated during the first pass, is read and stored in explicitTableName. If explicitTableName is null, then the first case holds and the the collection is treated exactly as in the current implementation of Hibernate, i.e. the value of the Hibernate variable collection.collectionTable (Section 3.3) is set to the table whose name is specified by implicitTableName. However, it should be noted here that an error can still occur if the table that actually stores the collection (and which is not specified by the user) and the table given by the implicitTableName are not the same. This is considered a design error (See Figure A.20).

If explicitTableName is not null, modified Hibernate determines the value of implicitTableName (note that the original Hibernate code does that too but at a later point in the second pass). Since all of the classes are already mapped, modified Hibernate can easily determine the name of the table on which the class, which makes up the collection, is persisted. The value of implicitTableName is not computed in the first pass because it might so happen that the class making up the collection is not already mapped and hence no information on that table exists. During the second pass all the classes are already mapped and the name of the table where a collection is persisted can be determined. This is the main reason for processing one-to-many collections in the second pass in modified Hibernate code. Modified Hibernate compares the value of explicitTableName and implicitTableName, if they are same, the collection is treated as one-to-many and if they are different, the collection is treated as many-to-many with a unique constraint. Under the second condition, the value of the variable collection.collectionTable is set to a table whose name is given by explicitTableName. Modified Hibernate (and the original Hibernate as well) keeps a list of all tables and it searches for a table in its list of tables whose name is specified by explicitTableName. Also, the value of the variable collection.element is set to null, allowing the modified Hibernate to set to type ManyToOne later. For a one-to-many collection, collection.element is set to a type one-to-many in the first pass while for many-to-many, collection.element is set to null in the first pass. So when a one-to-many collection acts like a many-to-many with a unique constraint in the second pass, the value of collection.element is set to null. The variable collection.element is set to a type ManyToOne when a many-to-many collection is further processed.

The snippets of code that have been added to the original Hibernate code are shown in Figure 3.15 and Figure 3.16.

```
public static void bindCollectionSecondPass(Element node,Collection
collection, java.util.Map persistentClasses, Mappings mappings,
java.util.Map inheritedMetas)throws MappingException {
  //Check whether a given collection is of type one-to-many.
  if ( collection.isOneToMany() ) {
    //Get the one-to-many node which was previously parsed from
    //the mapping file
    Element oneToManyNode = node.element( ``one-to-many'' );
    //Get the attribute table from a given node
    Attribute tableNode = node.attribute( ``table'' );
    //explicitTableName : value of the attribute tableNode if present
    String explicitTableName = null;
    //If there is a tableNode then get the value of the node and
    //convert it to a lowercase for a comparison
    if ( tableNode != null ) {
      explicitTableName = mappings.getNamingStrategy().
                          tableName( tableNode.getValue()).toLowerCase();
    }
    /* Get the name of the class (which comprises the collection) from
     * a one-to-many tag, then get the name of the table for that class
     * and convert it to a lower case for a comparison */
    String implicitTableName = mappings.getClass(
           getClassName(oneToManyNode.attributeValue(``class''),
               mappings))!= null?mappings.getClass(getClassName(
                   oneToManyNode.attributeValue(``class''),mappings)
                   ).getTable().getName().toLowerCase():null;
    /* if the user given name of the collection table is not same
     * as the one computed by original Hibernate then the collection must
     * a many-to-many with a unique constraint, and hence the type of the
     * be treated as collection which was previously
     * one-to-many is changed */
    if(explicitTableName != null && implicitTableName!= null &&
      !explicitTableName.equals(implicitTableName)) {
       //see Figure ref{fig:setUpManytoMany}
       setUpManyToMany(node, collection, persistentClasses,
                       mappings, inheritedMetas, tableNode);
    }
  }
   .....
}
```

Figure 3.15: Original Hibernate method bindCollectionSecondPass(..) changed to fit in Modified Hibernate

```
1  public static void setUpManyToMany(Element node, Collection collection,
2    java.util.Map persistentClasses, Mappings mappings,
3    java.util.Map inheritedMetas, Node tableNode) {
4
5    String tableName = mappings.getNamingStrategy()
6                .tableName( tableNode.getValue() );
7    // Reading attributes for a many-to-many.
8    // These lines are from the original Hibernate code.
9    Attribute schemaNode = node.attribute( ‘‘schema’’ );
10   String schema = schemaNode == null ?
11                  mappings.getSchemaName() : schemaNode.getValue();
12   Attribute catalogNode = node.attribute( ‘‘catalog’’ );
13   String catalog = catalogNode == null ?
14                  mappings.getCatalogName() : catalogNode.getValue();
15   Table table = mappings.addTable(
16                schema,catalog, tableName, getSubselect( node ),false);
17   collection.setCollectionTable( table );
18   bindComment(table, node);
19   // End of the original Hibernate code
20   // For a one-to-many collection, collection.element is set to a type
21   // one-to-many in the first pass.
22   collection.setElement(null);
23   log.info(
24        ‘‘Converting one-to-many to many-to-many : Mapping collection: ’’
25         + collection.getRole() +
26          ‘‘ --> ’’ + collection.getCollectionTable().getName()
27   );
28 }
```

Figure 3.16: A new method setUpManytoMany(..) added in a modified Hibernate

The method *bindCollectionSecondPass* (Figure 3.15) is called for all collections during the second pass. The original method "bindCollectionSecondPass" has been modified to implement the bugfix. If the collection requires treatment as a many-to-many collection with a unique constraint, then the modified Hibernate code calls the newly added method *setUpManyToMany*, shown in Figure 3.16, to re-read the collection node and set the parameters as done for a many-to-many collection. This execution changes the type of the collection - from one-to-many to many-to-many (later on). On line 22 of Figure 3.16, collection.element is set to null so that it no longer is set to OneToMany type. During later phase of the second pass, collection.element is set to ManyToOne Type for a one-to-many collection that needs to be treated as a many-to-many collection with a unique constraint.

From this point, the collection will not act as one-to-many (but rather as a many-to-many with a unique constraint) even though the XML mapping for the collection is a one-to-many. This is exactly the same behavior as the workaround.

Another part of code which is modified is the method *initOuterJoinFetchSetting* as given in the Figure 3.17. This method is responsible for assigning the fetching property for a collection. By default, this method sets up eager left join fetching for many-to-many, hence a collection which acts as a many-to-many with a unique constraint should follow this path as well. The method "initOuterJoinFetchSetting" is modified so as to set the eager join fetching, by default, for a one-to-many collection that acts as a many-to-many collection with a unique constraint. In the code given in the Figure 3.17, the "if" statement on a line 18 is different from that in the original Hibernate code. The expression after the "||" operator checks if the collection is specified as a one-to-many in a mapping file and it needs to be treated as a many-to-many collection with a unique constraint. For a many-to-many collection, the value of the variable collection.element is always of type ManyToOne. So any one-to-many collection that should act as a many-to-many collection with a unique constraint must have collection.element set to a type ManyToOne and this is done later in the second pass.

The last part of the code that has been modified is given in Figure 3.18. The method *bindColumns* is responsible for reading the data (which was previously read from the mapping file and stored in a tree like structure) stored in a node and generating columns for a table. Since the construct for a one-to-many does not have the *column* attribute, the column(s) for the collection table has to be generated from the table where the class (that makes up the collection) is persisted. The method has been changed in the modified Hibernate code with the addition of the code (from line 1 through 45 in Figure 3.18) that generates columns for a one-to-many node that need to be treated as a many-to-many node with a unique attribute set to true. The *while* loop on line 10 in Figure 3.18 iterates through the column(s) of the table that persists the class making up the collection. Line 1 through 45 are newly added codes while everything that is already there (from the original Hibernate) goes inside the *else* block on line 45.

```java
private static void initOuterJoinFetchSetting(
      Element node, Fetchable model, Mappings mappings) {
  Attribute fetchNode = node.attribute( ``fetch'' );
  final FetchMode fetchStyle;
  boolean lazy = true;
  if ( fetchNode == null ) {
    Attribute jfNode = node.attribute( ``outer-join'' );
    if ( jfNode == null ) {
      /* If a collection is of type many-to-many as specified
       * in an XML mapping OR
       * if a collection is of type one-to-many as specified
       * in an XML mapping BUT
       * the collection should be treated as a many-to-many
       * with a unique constraint then TRUE.
       * In the latter case, the collection is already set
       * to a type ManyToOne
      */
      if (``many-to-many''.equals( node.getName() )  ||
          (``one-to-many''.equals( node.getName() )  &&
          ``ManyToOne''.equals(model.getClass().getSimpleName()))) {
        //NOTE SPECIAL CASE: a
        // default to join and non-lazy for the ``second join''
        // of the many-to-many
        lazy = false;
        fetchStyle = FetchMode.JOIN;
      }
      .......
      .......
  model.setFetchMode( fetchStyle );
  model.setLazy(lazy);
}
```

Figure 3.17: A method initOuterJoinFetchSetting() in modified Hibernate

```
// one-to-many node that acts as a many-to-many node with a unique
// constraint. A many-to-many collection is internally
// treated it as a many-to-one class by the original Hibernate
if(``one-to-many''.equals(node.getName()) &&
   ``ManyToOne''.equals(simpleValue.getClass().getSimpleName())) {
  String classNodeValue = node.attributeValue(``class'');
  Iterator iter mappings.getClass(getClassName(classNodeValue,mappings)).
      getIdentifierProperty().getValue().getColumnIterator();
  int count = 0;
  while ( iter.hasNext() ) {
    //get an identifier from the class that makes up the collection
    Column columnIdentifier = (Column) iter.next();
    Column column = new Column();
    column.setValue( simpleValue );
    column.setTypeIndex( count++ );
    //set the variables as done by the method bindColumn
    column.setLength(columnIdentifier.getLength());
    column.setScale(columnIdentifier.getScale());
    column.setPrecision(columnIdentifier.getPrecision());
    column.setNullable(isNullable);
    column.setUnique(columnIdentifier.isUnique());
    column.setCheckConstraint(columnIdentifier.getCheckConstraint());
    column.setDefaultValue(columnIdentifier.getDefaultValue());
    column.setSqlType(columnIdentifier.getSqlType());
    column.setComment(columnIdentifier.getComment());
    // end of setting the variables as done by the bindColumn
    // get the name of the identifier column
    String logicalColumnName = mappings.getNamingStrategy().
        logicalColumnName(columnIdentifier.getName(), propertyPath);
         column.setName( mappings.getNamingStrategy().
         columnName(logicalColumnName ) );
    if ( table != null ) { //which is always true in the second pass
      table.addColumn( column );
          mappings.addColumnBinding( logicalColumnName, column, table );
    }
    simpleValue.addColumn( column );
    // column index
    bindIndex(null,table,column,mappings);
    bindIndex(ode.attribute(``index''),table,column,mappings);
    //column unique-key
    bindUniqueKey(null,table,column,mappings);
    bindUniqueKey(node.attribute(``unique-key''),table,column,mappings);
  }
}
else {
```

Figure 3.18: A method bindColumns(..) in modified Hibernate

3.5 Backward Compatibility of the Fix

This project adds to the capability of the Hibernate 3.2.6 (and Hibernate 3.3.2 as well) by broadening the scope of use of one-to-many construct. Current Hibernate ignores a table attribute even if it is specified for a one-to-many collection whereas modified Hibernate takes the table attribute into account. Specifying the wrong table in an XML mapping leads to an error in modified Hibernate. There can be six possible scenarios with a specification of a table attribute.

I A correct table attribute value is specified even when it is not necessary. Modified Hibernate will compare the table attribute value with the default table name for the collection. Since these two value are same, modified Hibernate would follow the execution path for a one-to-many collection as taken by the original Hibernate.

II An incorrect table attribute value is specified even when it is not necessary. The incorrect table name from an XML mapping would not match the default table name for the collection and hence, the collection table is set to an erroneous value (the one given by the table attribute). This results in an error under modified Hibernate and not in the original Hibernate.

III A correct table attribute value is specified when it is required. The consequence is same as in the first scenario for modified Hibernate but the original Hibernate fails here.

IV An incorrect table attribute value is specified when it is required. The incorrect table name from an XML mapping would not match the default table name for the collection and hence, the collection table is set to an erroneous value (the one given by the table attribute). This results in an error under modified Hibernate. The original Hibernate also fails here.

V No table attribute value is specified and it is not necessary. Modified Hibernate will treat the one-to-many collection the same as the original Hibernate.

VI No table attribute value is specified even when it is required. Modified Hibernate will act no different from the original Hibernate and since the table attribute is required here but not specified, this would result in an erroneous outcome for both the original Hibernate and the modified Hibernate.

These scenarios show that the responsibility of specifying the correct table lies entirely upon the user. If the user specifies the wrong table assuming the original Hibernate ignores the table attribute, errors will occur. To make modified Hibernate backward compatible with the original Hibernate, users should decide whether to specify the table attribute or not in the first place and if yes, should specify the correct table. If the user wants the backward compatibility of the modified Hibernate with already developed projects, it is recommended to remove table attributes from all one-to-many collections, which is ignored in the original Hibernate.

3.6 Validation of a Fix to the Bug

The Hibernate class HbmBinder is modified so as to fix the bug. The patch for the class HbmBinder for the modified Hibernate is generated using Unix *diff* and *patch* commands. The patch is then integrated into the original Hibernate code to form the modified Hibernate. The patch generated is shown in Figures A.21 through A.26.

For a proper validation of the modified Hibernate, it is tested against the Hibernate test suite and additional test cases that are built for this project. In Hibernate 3.2.6, there are 1273 JUnit test cases. For validation purposes, the original Hibernate is tested against these test cases, out of which 39 test cases produced an error and 14 test cases ended up with a failure. Some of the test cases in the Hibernate package are expected to fail i.e. they are written for testing a bad case. The modified Hibernate is also run against the same set of test cases and the result of running the JUnit test cases on modified Hibernate is consistent with that of the original Hibernate code.

The test reports are generated in an XML format (by the Eclipse IDE which is used for the project). The XML test reports generated for the original Hibernate and the modified Hibernate are processed by an XSL script that compares the test results from the original Hibernate code and the modified Hibernate code, and displays the errors and failures in modified Hibernate code along with an indication whether the failure or the error occurs in the original Hibernate as well. Table 3.1 shows the list of errors in modified Hibernate. The first column and the second column in Table 3.1 specifies the name of the test case that failed and the name of the class where the failed test case is defined respectively. The value *ok* on the third column *Issue* indicates whether the failed test case occurred in the original Hibernate as well. As it can be seen, here all of the modified Hibernate failed test cases are failed by the original Hibernate as well.

AllTests		
Failures : 14		
Test Name	**Test Class**	**Issue**
testLazy	org.hibernate.test.bidi.AuctionTest	ok
testOneToOnePropertyRef	org.hibernate.test.cuk.CompositePropertyRefTest	ok
testUpdateDetachedParentNo ChildrenToNullFailureExpected	org.hibernate.test.event.collection.BrokenCollection EventTest	ok
testBooleanHandling	org.hibernate.test.hql.BulkManipulationTest	ok
testEmptyInListFailureExpected	org.hibernate.test.hql.HQLTest	ok
testMaxindexHqlFunctionInEleme ntAccessorFailureExpected	org.hibernate.test.hql.HQLTest	ok
testMultipleElementAccessor OperatorsFailureExpected	org.hibernate.test.hql.HQLTest	ok
testKeyManyToOneJoinFailure Expected	org.hibernate.test.hql.HQLTest	ok
testDuplicateExplicitJoinFailure Expected	org.hibernate.test.hql.HQLTest	ok
testLoadEntityWithEagerFetching ToKeyManyToOneReference-BackToSelfFailureExpected	org.hibernate.test.keymanytoone.bidir.component. EagerKeyManyToOneTest	ok
testCreate	org.hibernate.test.legacy.FooBarTest	ok
testCreateUpdate	org.hibernate.test.legacy.FooBarTest	ok
testFind	org.hibernate.test.legacy.FooBarTest	ok
testPersistentLifecycle	org.hibernate.test.legacy.FooBarTest	ok

Table 3.1: Comparison of the failures in the modified Hibernate with that in the original Hibernate

The result of running the XSL script on the modified Hibernate code for error comparison is shown in Table 3.2 and Table 3.3. Since the number of errors i.e. 39 is large, the list of errors is split into two tables. The meaning of the respective columns in Table 3.2 and Table 3.3. are same as that in Table 3.1.

AllTests		
Errors : 1-26/39		
Test Name	**Test Class**	**Issue**
testSaveOrphanDeleteChildWith ParentFailureExpected	org.hibernate.test.cascade.BidirectionalOneToMany CascadeTest	ok
testSaveParentNullChildrenFailure Expected	org.hibernate.test.event.collection.BrokenCollection EventTest	ok
testUpdateParentNoChildrenTo NullFailureExpected	org.hibernate.test.event.collection.BrokenCollection Eve ntTest	ok
testIntegrityViolation	org.hibernate.test.exception.SQLExceptionConver sionTest	ok
testBadGrammar	org.hibernate.test.exception.SQLExceptionConver sionTest	ok
testFormulaJoin	org.hibernate.test.formulajoin.FormulaJoinTest	ok
testParameterTypeMismatch Fail-ureExpected	org.hibernate.test.hql.ASTParserLoadingTest	ok
testCriteriaAggregationReturn TypeFailureExpected	org.hibernate.test.hql.CriteriaHQLAlignmentTest	ok
testLockModeTypeRead	org.hibernate.test.jpa.lock.JPALockTest	ok
testLockModeTypeWrite	org.hibernate.test.jpa.lock.JPALockTest	ok
testStaleVersionedInstanceFound InQueryResult	org.hibernate.test.jpa.lock.RepeatableReadTest	ok
testStaleVersionedInstanceFound OnLock	org.hibernate.test.jpa.lock.RepeatableReadTest	ok
testStaleNonVersionedInstance FoundInQueryResult	org.hibernate.test.jpa.lock.RepeatableReadTest	ok
testStaleNonVersionedInstance Found OnLock	org.hibernate.test.jpa.lock.RepeatableReadTest	ok
testLoadingAndSerializationOf Configuration	org.hibernate.test.legacy.ConfigurationPerformance Test	ok
testQuery	org.hibernate.test.legacy.FooBarTest	ok
testOneToOneGenerator	org.hibernate.test.legacy.FooBarTest	ok
testCollectionOfSelf	org.hibernate.test.legacy.FooBarTest	ok
testReachability	org.hibernate.test.legacy.FooBarTest	ok
testVersionedCollections	org.hibernate.test.legacy.FooBarTest	ok
testMap	org.hibernate.test.legacy.MapTest	ok
testSelfManyToOne	org.hibernate.test.legacy.MasterDetailTest	ok
testExample	org.hibernate.test.legacy.MasterDetailTest	ok
testMultiTableCollections	org.hibernate.test.legacy.MultiTableTest	ok
testMultiTableManyToOne	org.hibernate.test.legacy.MultiTableTest	ok
testReturnPropertyComponent RenameFailureExpected	org.hibernate.test.legacy.SQLLoaderTest	ok

Table 3.2: Comparison of the first twenty-six errors in the modified Hibernate with that in the original Hibernate

AllTests		
Errors : 27-39/39		
Test Name	**Test Class**	**Issue**
testOptimisticLockDirty	org.hibernate.test.optlock.OptimisticLockTest	ok
testOptimisticLockAll	org.hibernate.test.optlock.OptimisticLockTest	ok
testOptimisticLockDirtyDelete	org.hibernate.test.optlock.OptimisticLockTest	ok
testOptimisticLockAllDelete	org.hibernate.test.optlock.OptimisticLockTest	ok
testReadOnlyOnProxiesFailure Expected	org.hibernate.test.readonly.ReadOnlyTest	ok
testScalarStoredProcedure	org.hibernate.test.sql.hand.custom.mysql.MySQL CustomSQLTest	ok
testParameterHandling	org.hibernate.test.sql.hand.custom.mysql.MySQL CustomSQLTest	ok
testEntityStoredProcedure	org.hibernate.test.sql.hand.custom.mysql.MySQL CustomSQLTest	ok
testCompositeIdJoinsFailure Expected	org.hibernate.test.sql.hand.query.NativeSQLQueries Test	ok
testAutoDetectAliasing	org.hibernate.test.sql.hand.query.NativeSQLQueries Test	ok
testConcurrentCachedDirty Queries	org.hibernate.test.tm.CMTTest	ok
testSave	org.hibernate.test.typeparameters.TypeParameter Test	ok
testLoading	org.hibernate.test.typeparameters.TypeParameter Test	ok

Table 3.3: Comparison of the last thirteen errors in the modified Hibernate with that in the original Hibernate

To further test modified Hibernate, five test cases are built and run on both modified Hibernate and the original Hibernate. The XML mappings for the these test cases are shown in Figure A.15 through Figure A.19. Table 3.4 shows the outcomes of running these five test cases in the modified Hibernate and the original Hibernate.

MyTests		
Testcase	**Original Hibernate**	**Modified Hibernate**
Test Figure A.15	Failed	Passed
Test Figure A.16	Failed	Passed
Test Figure A.17	Failed	Passed
Test Figure A.18	Passed	Passed
Test Figure A.19	Failed	Failed
Test Figure A.20	Failed	Failed

Table 3.4: Results of running the testcases built for this project on the original Hibernate and the modified Hibernate

The first case, Figure A.15 tests for composite primary key and a unidirectional one-to-many association. A class Orderline has a composite id, *id*, of type OrderLineId and a property *name*. A class Order has *id*, *orderName* and *orderLine* as its properties among which orderLine is a collection of Orderline objects. There is a unidirectional one-to-many association from the classes Oder to Orderline. The collection is persisted in the table called *LINKTABLE*.

In the next two test cases, Figure A.16 and Figure A.17, classes House and Person are mapped to tables *HOUSE* and *PERSON*, and there exists a one-to-many association from the class House to Person. The collection of Person objects is persisted in the table *HOUSE_PERSON* which is explicitly stated in an XML mapping i.e. the table HOUSE_PERSON has a column HOUSE_ID that references the column HOUSE_ID of the table HOUSE. The original Hibernate assumes the collection table is PERSON instead of HOUSE_PERSON, which is the correct one. The difference between these two tests is Figure A.16 is bidirectional and Figure A.17 is unidirectional.

The fourth test case Figure A.18 is same as the second test case except that the collection is persisted in the table PERSON. The table for collection is specified as PERSON in the XML file even though it is not required because the collection is persisted in the table PERSON as assumed in the original Hibernate. The successful run of this test case shows that when a correct table is specified in the XML mapping even when it is not necessary (i.e the collection table assumed by the original Hibernate is same as the one specified in the mapping file), modified Hibernate works with success. In the fifth test case, Figure A.19, the wrong collection table name is specified in the mapping file. The value of the table attribute should be "HOUSE_PERSON" and not "PERSON". Since a wrong table name is specified by the user, the test case fails. The test case in Figure A.19 shows that it is the responsibility of the user to specify the correct table for the collection. Failure to do so would result in an error.

In the sixth and the last test case, Figure A.20, the table name is not specified even when it is required because the one-to-many collection, *persons*, is persisted on the table, "HOUSE_PERSON". Since the table attribute is not given, modified Hibernate, like the original Hibernate, assumes the collection is persisted on the table “PERSON” which results in a failure.

3.7 Summary of Results

Table 3.5 shows the results of running the testcases in different versions of Hibernate. Hibernate 3.2.6 is used to develop a project. The latest version of Hibernate used for production is 3.3.2 (Hibernate 3.5.0-Beta-2 is still under development). Before applying the final modification to the original Hibernate, two different approaches are taken (Section 3.2). The two approaches - Modification1 and Modification2 - are tested against the same test suites as modified Hibernate, which implements a correct approach to extend the Hibernate 3.2.6, is. The class HbmBinder from the modified Hibernate is integrated in Hibernate 3.3.2 and the testcases under Hibernate 3.3.2 are run, the results of which can be seen in Table 3.5. The integration of the class HbmBinder to Hibernate 3.3.2 here requires a few changes in importing packages.

Tested code	Number of test cases	Number of test cases failed	Number of failures
Hibernate 3.2.6	1273	39	14
Modification1 applied to the original Hibernate 3.2.6 (Section 3.2)	1273	825	2
Modification2 applied to the original Hibernate 3.2.6 (Section 3.2)	1273	49	14
Modified Hibernate 3.2.6	1273	39	14
Hibernate 3.3.2	1425	62	31
Modified Hibernate 3.3.2	1425	62	31

Table 3.5: Summary of all tests conducted

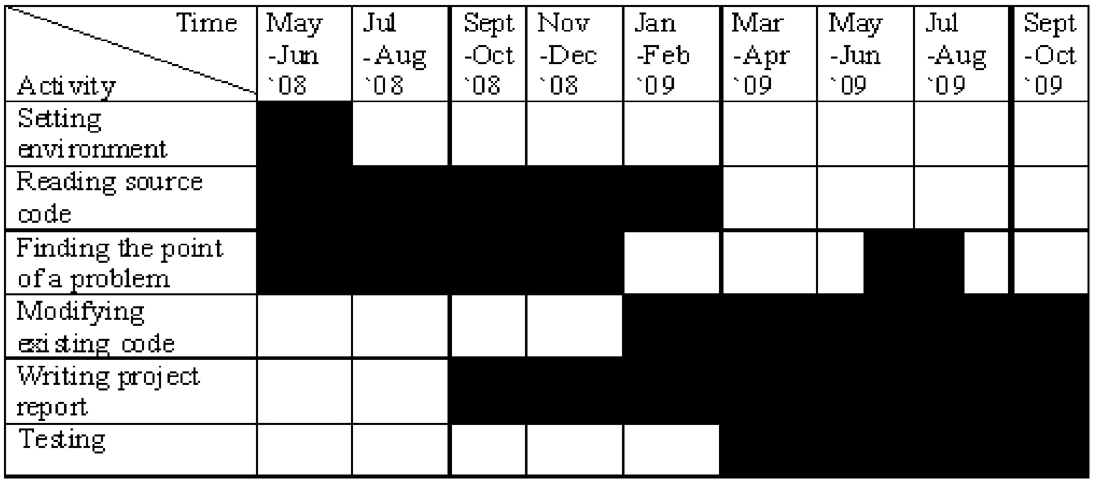
The original Hibernate code has 1273 test cases out of which it fails 14 test cases and results in an error in 39 test cases. The first approach to fix the bug, Modification1, has a large number of errors – 825 – however, it has three fewer failures compared to the original Hibernate. Modification2 has only 49 errors and 14 failures. The final approach, the modified Hibernate code, has same number of errors and failures as the original Hibernate as specified in the fifth row of the Table 3.5. The latest version of Hibernate used in production, Hibernate 3.3.2, has more test cases than Hibernate 3.2.6. The last two rows of the table compares the results of running testcases on Hibernate 3.3.2 and the Hibernate 3.3.2 with HbmBinder class from modified Hibernate incorporated to it. Note the results are the same. This shows that the fix which is developed for Hibernate 3.2.6 can be applied to Hibernate 3.3.2 as well.

CHAPTER FOUR

Conclusion

4.1 Timeline

Hibernate is an ambitious venture with about one thousand and seventy six Java classes. Hibernate 3.2.6 is used to develop this project. The latest version of Hibernate (3.3.2) still has bug “Invalid queries when using subclasses and one-to-many associations”. The extension of Hibernate, which is the objective of this Master’s project, involves understanding about three hundred lines of code which are not well documented. Documentation of source code is an important phase of the software development cycle and enables the reader of the source code to understand the meaning and functioning of modules in the software. Since most of the Hibernate code is not documented, a huge amount of time was required to understand the methods involved in the instantiation of bug in the Hibernate code. The activity timeline for the project is given in Figure 4.1.



Time / Activity	May -Jun ‘08	Jul -Aug ‘08	Sept -Oct ‘08	Nov -Dec ‘08	Jan -Feb ‘09	Mar -Apr ‘09	May -Jun ‘09	Jul -Aug ‘09	Sept -Oct ‘09
Setting environment	■								
Reading source code	■	■	■	■	■				
Finding the point of a problem	■	■	■	■			■	■	
Modifying existing code					■	■	■	■	■
Writing project report			■	■	■	■	■	■	■
Testing						■	■	■	■

Figure 4.1: Project timeline

4.2 Activities

I Setting Environment : This phase involved setting up configurations to work with Hibernate and Eclipse IDE, setting up HSQL DB and MySQL.

II Reading source code: As mentioned earlier, Hibernate consists of more than one thousand classes and has poor documentation. Digging through the source code and familiarization with the workflow are essential, but at the same time this is tedious and time consuming. Although the patch for the extension of Hibernate as intended by this project is nearly 160 lines, going through numerous methods that directly or indirectly affect the patch was painstaking. The UML diagrams of various classes and packages aided in getting the exact picture of the Hibernate source code. UML diagrams helps in visualizing an entire system or a part of a system from different perspectives and from different levels. Class diagrams and package diagrams were helpful for this project. Rational Software Architect (RSA) was a tool used for reverse engineering. By using RSA, UML diagrams for Hibernate were reverse engineered from the Hibernate source code.

III Finding the point of problem: The exact point of the problem (which is responsible for the wrong code generation) is located. Lack of proper documentation in the original Hibernate source codes led to a several false starts. Originally, it is thought that a wrong alias is assigned to a table. Finally, the original source of a problem i.e. the irregularity in one-to-many construct is found out.

IV Modifying existing code: To fix the bug, the existing source code is modified numerous times with different approaches. The initial approach focuses on inheritance. As mentioned in 1.3, initially the problem was thought to be with inheritance. The first approach tried to find the exact table, among the tables in a class hierarchy, whose rows form a part of a collection. A new class is added that had the responsibility of generating a proper aliases for tables. Modifications are done at the lowest level where the impacts of changes would be minimal. The new class is modified many times as a result of running test cases. Since inheritance is not the problem, the modified source code failed additional test cases. The final approach is related to the irregularity in the usage of a one-to-many association. The final approach imitates a many-to-many association whenever there is a "table" value for a one-to-many association and the given "table" value is different from the one that Hibernate determines for a one-to-many association.

V Writing a project report: Involves the preparation of this report itself.

VI Testing: There is a whole suite of test cases available for the Hibernate source code. Finally, I added my own set of test cases for testing the bug fix. Testing was done thoroughly for every modification done to the code.

4.3 Conclusion

In this project, a new feature is added to Hibernate which allows the use of one-to-many construct with a join table on the many side of the association. The work is motivated from Bug HHH-3095 [13] in Hibernate. Prior to the new extension to Hibernate 3.2.6 (which is called modified Hibernate), Hibernate 3.2.6 (which is called the original Hibernate) does not support using a one-to-many construct for a one-to-many association when join tables are used on the many side of the association. However, there is a workaround for it. In the original Hibernate, users define many-to-many construct with a unique attribute set to true. The workaround, to use many-to-many for a one-to-many association, does not seem apparent to many users. The project takes this into account and lets the users define one-to-many constructs for all one-to-many associations. Clearly, this is more natural than the workaround. Modified Hibernate does not require any extra information on mapping metadata, rather it extends the original Hibernate by complying within the framework definitions in the original Hibernate. In particular, modified Hibernate makes use of a table attribute specified for a collection which is ignored in the original Hibernate. Hence, no violations of Hibernate rules are made.

Test cases show that modified Hibernate fixes the problem specified in the bug [13]. Modified Hibernate is run on the Hibernate test suites. The test results are consistent with the original Hibernate i.e. the only failures in modified Hibernate are also in original Hibernate. Both versions of Hibernate 3.2.6 are tested under new test cases that involve one-to-many associations and the test results for both are as expected. Hibernate 3.3.2, the latest version of Hibernate that can be used for production, is also integrated with the modification made for Hibernate 3.2.6. The modifications made to Hibernate 3.3.2 are also successful in that Hibernate 3.3.2 and Hibernate 3.3.2 with the modification produced the same test suite results.

Since Hibernate 3.3.2 is the latest Hibernate version in use, a patch is generated for Hibernate 3.3.2. The patch is submitted to Hibernate group in their issue tracking system. Hibernate is being developed over time - its functionality increasing with every new version. The integration of the patch submitted to Hibernate 3.3.2 will extend its evergrowing capability.

APPENDICES

APPENDIX A

APPENDIX

```
package cat;
public class DomesticCat extends Cat {
  protected String dcatName;
  //getters and setters
  public String getdcatName() {
    return this.dcatName;
  }
  public void setdcatName(String dcatName) {
    this.dcatName = dcatName;
  }
}
```

Figure A.1: Java class DomesticCat from Figure 2.5

```
package cat;
public class ZooCat extends Cat {
  protected String zcatName;
  //getters and setters
  public String getzcatName() {
    return this.zcatName;
  }
  public void setzcatName(String zcatName) {
    this.zcatName = zcatName;
  }
}
```

Figure A.2: Java class ZooCat from Figure 2.5

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
“-//Hibernate/Hibernate Mapping DTD//EN”
“http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd”>
<hibernate-mapping >
  <class name=“DomesticCat” extends=“Cat” table=“DCAT”>
    <id name=“ID” column=“CAT_ID”>
      <generator class=“increment”/>
    </id>
    <property name=“breed” column = “BREED”/>
    <property name=“dcatName” column = “DCAT_NAME”/>
  </class>
</hibernate-mapping>
```

Figure A.3: XML mapping for a class DomesticCat in A.1 using “table per concrete class with implicit polymorphism”

```
<?xml version=“1.0”?>
<!DOCTYPE hibernate-mapping PUBLIC
“-//Hibernate/Hibernate Mapping DTD//EN”
“http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd”>
<hibernate-mapping >
  <class name=“ZooCat” extends=“Cat” table=“ZCAT”>
    <id name=“ID” column=“CAT_ID”>
      <generator class=“increment”/>
    </id>
    <property name=“zcatName” column = “ZCAT_NAME”/>
  </class>
</hibernate-mapping>
```

Figure A.4: XML mapping for a class ZooCat in A.2 using “table per concrete class with implicit polymorphism”

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping >
   <class name="Cat" table="CAT" >
      <id name="ID" column="CAT_ID">
         <generator class="increment"/>
      </id>
      <property name="breed" column = "BREED"/>
      <union-subclass name="DomesticCat" table="DCAT">
         <property name="dcatName" column = "DCAT_NAME"/>
      </union-subclass>
      <union-subclass name="ZooCat" table="ZCAT">
         <property name="zcatName" column = "ZCAT_NAME"/>
      </union-subclass>
   </class>
</hibernate-mapping>
```

Figure A.5: XML mapping for the classes in Figure 2.5 using "table per concrete class"

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping >
   <class name="Cat" table="CAT" batch-size="100">
      <id name="id" column="CAT_ID" >
         <generator class="increment"/>
      </id>
      <property name="breed" column = "BREED"/>
      <discriminator column="CAT_TYPE" />
   </class>
</hibernate-mapping>
```

Figure A.6: XML mapping for the class Cat in Figure 2.5 using "Table per class hierarchy"

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping >
  <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
    <property name="dcatNam" column = "DCAT_NAME"/>
  </subclass>
</hibernate-mapping>
```

Figure A.7: XML mapping for the class DomesticCat in Figure 2.5 using "Table per class hierarchy"

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping >
  <subclass name="ZooCat" extends="Cat" discriminator-value="Z">
    <property name="zcatName" column = "ZCAT_NAME"/>
  </subclass>
</hibernate-mapping>
```

Figure A.8: XML mapping for the class ZooCat in Figure 2.5 using "Table per class hierarchy"

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping >
  <class name="Cat" table="CAT" >
    <id name="ID" column="CAT_ID">
      <generator class="increment"/>
    </id>
    <property name="breed" column="BREED"/>
  </class>
</hibernate-mapping>
```

Figure A.9: XML mapping for the class Cat from Figure 2.5 using "Table per subclass"

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="cat">
  <joined-subclass name="DomesticCat" extends="Cat" table="DCAT">
    <key column="CAT_ID"/>
    <property name="dcatName" column="DCAT_NAME"/>
  </joined-subclass>
</hibernate-mapping>
```

Figure A.10: XML mapping for the class DomesticCat from Figure 2.5 using "Table per subclass"

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="cat">
  <joined-subclass name="ZooCat" extends="Cat" table="ZCAT">
    <key column="CAT_ID"/>
    <property name="zcatName" column="ZCAT_NAME"/>
  </joined-subclass>
</hibernate-mapping>
```

Figure A.11: XML mapping for the class ZooCat from Figure 2.5 using "Table per subclass"

```
package cat;
public class DomesticCat extends Cat {
  protected String dcatName;
  protected House house;
  //getters and setters
  public String getdcatName() {
    return this.dcatName;
  }
  public void setdcatName(String dcatName) {
    this.dcatName = dcatName;
  }
  public House getHouse() {
    return house;
  }
  public void setHouse(House house) {
    this.house = house;
  }
}
```

Figure A.12: Java class DomesticCat used in the project

```java
package cat;

import java.util.ArrayList;
import java.util.Collection;

public class House {
  protected Long id;
  protected Collection<DomesticCat> animals = new ArrayList();
  protected String houseName;
  //getters and setters
  public Long getId() {
    return id;
  }
  public void setId(Long id) {
    this.id = id;
  }
  public Collection<DomesticCat> getAnimals() {
    return animals;
  }
  public void setAnimals(Collection<DomesticCat> animals) {
    this.animals = animals;
  }
  public String getHouseName() {
    return this.houseName;
  }
  public void setHouseName(String houseName) {
    this.houseName = houseName;
  }
}
```

Figure A.13: Java class House used in the project

```
package cat;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class CatLoader {
  public static void main(String[] args) {
    Configuration cfg = new Configuration();
    cfg.configure(``hibernate.cfg.xml'');
    SessionFactory sessionFactory = cfg.buildSessionFactory();
    Session session = sessionFactory.getCurrentSession();
    session.beginTransaction();
    List<DomesticCat> dcats = session.createQuery(
      ``from DomesticCat dc order by dc.id asc'').list();
    System.out.println(``The size of the list of domestic cats is ''
                      + dcats.size());
    for(Object m : dcats){
      DomesticCat loadedDCats = (DomesticCat) m;
      System.out.println(loadedDCats.getId()+`` and ''
                        + loadedDCats.getHouse().getId());
    }
    session.getTransaction().commit();
  }
}
```

Figure A.14: Java class CatLoader used in the project

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="mypackage">
   <class name="Orderline" table="ORDERLINE">
      <composite-id name="id" class="OrderLineId">
         <key-property name="lineId" column="LINEID"/>
         <key-property name="orderId" column="ORDERID"/>
         <key-property name="customerId" column="CUSTOMERID"/>
      </composite-id>
      <property name="name" column="NAME"/>
   </class>

   <class name="Order" table="ORDERTABLE">
      <id name="id" column="ORDER_ID" type="java.lang.Long">
         <generator class="increment"/>
      </id>
      <property name="orderName" column="ORDER_NAME"/>
      <set name="orderLines" table="LINKTABLE">
         <key>
            <column name="ORDER_ID"/>
         </key>
         <one-to-many class="Orderline"/>
      </set>
   </class>
</hibernate-mapping>
```

Figure A.15: TestCase1 - CompositeId test

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="mypackage">
  <class name="House" table="HOUSE" >
    <! - - id - - >
    <id name="id" column="HOUSE_ID" type="java.lang.Long">
      <generator class="increment"/>
    </id>
    <property name="housename" column="HOUSE_NAME"/>
    <set name="persons" lazy="false" cascade="all" table = "HOUSE_PERSON"
      fetch="select" inverse="true">
      <key column="HOUSE_ID" not-null="false"/>
      <one-to-many class="Person" />
    </set>
  </class>

  <class name="Person" table="PERSON" >
    <!– id –>
    <id name="id" column="PERSON_ID" type="java.lang.Long">
      <generator class="increment"/>
    </id>
    <property name = "name" column = "PERSON_NAME" type = "string"/>
    <join table="HOUSE_PERSON">
      <key column="PERSON_ID" />
      <many-to-one name="myhouse" column = "HOUSE_ID" lazy="false" not-null="false"/>
    </join>
  </class>
</hibernate-mapping>
```

Figure A.16: TestCase2 - Bidirectional One-To-Many with Join

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="mypackage">
   <class name="Person" table="PERSON" >
      <!- id ->
      <id name="id" column="PERSON_ID" type="java.lang.Long">
         <generator class="increment"/>
      </id>
      <property name = "name" column = "PERSON_NAME" type = "string"/>
   </class>

   <class name="House" table="HOUSE" >
      <!- id ->
      <id name="id" column="HOUSE_ID" type="java.lang.Long">
         <generator class="increment"/>
      </id>
      <property name="housename" column="HOUSE_NAME"/>
      <set name="persons" lazy="false" cascade="none" table = "HOUSE_PERSON"
         fetch="select">
         <key column="HOUSE_ID" not-null="false"/>
         <one-to-many class="Person" />
      </set>
   </class>
</hibernate-mapping>
```

Figure A.17: TestCase3 - Unidirectional One-To-Many without Join but an Association mapped to a different table

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="mypackage">
  <class name="House" table="HOUSE" >
    <! - - id - - >
    <id name="id" column="HOUSE_ID" type="java.lang.Long">
      <generator class="increment"/>
    </id>
    <property name="housename" column="HOUSE_NAME"/>
    <set name="persons" lazy="false" cascade="all" table = "PERSON"
      fetch="select" inverse="true">
      <key column="HOUSE_ID" not-null="false"/>
      <one-to-many class="Person" />
    </set>
  </class>

  <class name="Person" table="PERSON" >
    <!– id –>
    <id name="id" column="PERSON_ID" type="java.lang.Long">
      <generator class="increment"/>
    </id>
    <many-to-one name="myhouse" column = "HOUSE_ID" lazy="false" not-null="false"/>
    <join table="HOUSE_PERSON">
      <key column="PERSON_ID"/>
      <property name = "name" column = "PERSON_NAME" type = "string"/>
    </join>
  </class>
</hibernate-mapping>
```

Figure A.18: TestCase4 - Bidirectional One-To-Many which does not require table attribute in set but it is specified

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="mypackage">
  <class name="House" table="HOUSE" >
    <! - - id - - >
    <id name="id" column="HOUSE_ID" type="java.lang.Long">
      <generator class="increment"/>
    </id>
    <property name="housename" column="HOUSE_NAME"/>
    <set name="persons" lazy="false" cascade="all" table = "PERSON"
      fetch="select" inverse="true">
      <key column="HOUSE_ID" not-null="false"/>
      <one-to-many class="Person" />
    </set>
  </class>

  <class name="Person" table="PERSON" >
    <!– id –>
    <id name="id" column="PERSON_ID" type="java.lang.Long">
      <generator class="increment"/>
    </id>
    <property name = "name" column = "PERSON_NAME" type = "string"/>
    <join table="HOUSE_PERSON">
      <key column="PERSON_ID" />
      <many-to-one name="myhouse" column = "HOUSE_ID" lazy="false" not-null="false"/>
    </join>
  </class>
</hibernate-mapping>
```

Figure A.19: TestCase5 - Bidirectional One-To-Many that requires table attribute in set but the wrong table name is specified

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="mypackage">
  <class name="House" table="HOUSE" >
    <! - - id - - >
    <id name="id" column="HOUSE_ID" type="java.lang.Long">
      <generator class="increment"/>
    </id>
    <property name="housename" column="HOUSE_NAME"/>
    <set name="persons" lazy="false" cascade="all" inverse="true"
      fetch="select">
      <key column="HOUSE_ID" not-null="false"/>
      <one-to-many class="Person" />
    </set>
  </class>

  <class name="Person" table="PERSON" >
    <!– id –>
    <id name="id" column="PERSON_ID" type="java.lang.Long">
      <generator class="increment"/>
    </id>
    <property name = "name" column = "PERSON_NAME" type = "string"/>
    <join table="HOUSE_PERSON">
      <key column="PERSON_ID" on-delete="cascade"/>
      <many-to-one name="myhouse" column = "HOUSE_ID" lazy="false" not-null="false"/>
    </join>
  </class>
</hibernate-mapping>
```

Figure A.20: TestCase6 - Bidirectional One-To-Many that requires table attribute in set but no table name is specified

```
--- HbmBinder.java        2009-10-19 23:07:46.000000000 -0500
+++ HbmBinder_Sweta.java          2009-10-19 23:07:55.000000000 -0500
@@ -1029,7 +1029,49 @@
            final Mappings mappings) throws MappingException {

      Table table = simpleValue.getTable();
+     //one-to-many node that acts as a many-to-many node with a
+     //unique constraint
+     //if the collection is mapped as a many-to-many, Hibernate internally
+     //treats it as a many-to-one class
+     if(``one-to-many''.equals(node.getName()) && ``ManyToOne''.
        equals(simpleValue.getClass().getSimpleName())){
+       String classNodeValue = node.attributeValue(``class'');
+       Iterator iter = mappings.getClass(getClassName(classNodeValue,
            mappings)).getIdentifierProperty().getValue().getColumnIterator();
+       int count = 0;
+       while ( iter.hasNext() ) {
+         //get an indentifier column from the class that
+         //makes up the collection
+         Column columnIdentifier = (Column) iter.next();
+         Column column = new Column();
+         column.setValue( simpleValue );
+         column.setTypeIndex( count++ );
+         //set the variables as done by the method bindColumn
+         column.setLength(columnIdentifier.getLength());
+         column.setScale(columnIdentifier.getScale());
+         column.setPrecision(columnIdentifier.getPrecision());
+         column.setNullable(isNullable);
+         column.setUnique(columnIdentifier.isUnique());
+         column.setCheckConstraint(columnIdentifier.getCheckConstraint());
+         column.setDefaultValue(columnIdentifier.getDefaultValue());
+         column.setSqlType(columnIdentifier.getSqlType());
+         column.setComment(columnIdentifier.getComment());
+         // end of setting the variables as done by the bindColumn
+         // get the name of the identifier column
+         String logicalColumnName = mappings.getNamingStrategy().
+            logicalColumnName(columnIdentifier.getName(), propertyPath);
+         column.setName( mappings.getNamingStrategy().
+         columnName(logicalColumnName ) );
+         if ( table != null ) { //which is always true in the second pass
+           table.addColumn( column );
+           mappings.addColumnBinding( logicalColumnName, column, table );
+         }
+         simpleValue.addColumn( column );
```

Figure A.21: The patch generated

```
+         // column index
+         bindIndex( null, table, column, mappings );
+         bindIndex( node.attribute(``index''),table,column,mappings);
+         //column unique-key
+         bindUniqueKey( null, table, column, mappings );
+         bindUniqueKey( node.attribute(``unique-key''),table,column,
                        mappings);
+       }
+     }
+     else{
        // COLUMN(S)
        Attribute columnAttribute = node.attribute(``column'');
        if ( columnAttribute == null ) {
@@ -1097,7 +1139,7 @@
        bindIndex( node.attribute(``index''),table,column,mappings);
        bindUniqueKey( node.attribute(``unique-key''),table,column,
                     mappings);
     }
-
+   }
    if ( autoColumn && simpleValue.getColumnSpan() == 0 ) {
      Column column = new Column();
      column.setValue( simpleValue );
@@ -1378,8 +1420,7 @@

     // FETCH STRATEGY
-    initOuterJoinFetchSetting( node, collection );
-
+    initOuterJoinFetchSetting( node, collection,mappings );
     if (``subselect''.equals( node.attributeValue(``fetch'') ) ) {
        collection.setSubselectLoadable(true);
        collection.getOwner().setSubselectLoadableCollections(true);
@@ -1557,7 +1598,7 @@
        boolean isNullable, Mappings mappings) throws MappingException {

     bindColumnsOrFormula( node, manyToOne, path, isNullable, mappings );
-    initOuterJoinFetchSetting( node, manyToOne );
+    initOuterJoinFetchSetting( node, manyToOne, mappings );
     initLaziness( node, manyToOne, mappings, true );
     Attribute ukName = node.attribute( ``property-ref'' );
@@ -1640,7 +1681,7 @@
              ForeignKeyDirection.FOREIGN_KEY_FROM_PARENT :
              ForeignKeyDirection.FOREIGN_KEY_TO_PARENT );
```

Figure A.22: The patch generated (contd..)

```
-    initOuterJoinFetchSetting( node, oneToOne );
+    initOuterJoinFetchSetting( node, oneToOne, mappings );
     initLaziness( node, oneToOne, mappings, true );

     oneToOne.setEmbedded( ‘‘true’’.equals(
             node.attributeValue(‘‘embed-xml’’) ) );
@@ -1954,15 +1995,18 @@
     return typeNode.getValue();
   }

- private static void initOuterJoinFetchSetting(Element node,
        Fetchable model) {
+
+ private static void initOuterJoinFetchSetting(Element node,
      Fetchable model, Mappings mappings) {

     Attribute fetchNode = node.attribute( ‘‘fetch’’ );
     final FetchMode fetchStyle;
     boolean lazy = true;
     if ( fetchNode == null ) {
       Attribute jfNode = node.attribute( ‘‘outer-join’’ );
       if ( jfNode == null ) {
-        if ( ‘‘many-to-many’’.equals( node.getName() ) ) {
-          //NOTE SPECIAL CASE:
+          //if a node is a many-to-many OR a node is a one-to-many but
+          //needs to be treated as a many-to-many node
+          //with a unique constraint
+          if ( ‘‘many-to-many’’.equals( node.getName() )    ||
              (‘‘one-to-many’’.equals( node.getName() ) &&
               ‘‘ManyToOne’’.equals(model.getClass().getSimpleName()))
){
+
+            //NOTE SPECIAL CASE:a
             // default to join and non-lazy for the ‘‘second join’’
             // of the many-to-many
             lazy = false;
@@ -1999,7 +2043,6 @@
     model.setFetchMode( fetchStyle );
     model.setLazy(lazy);
   }
-
  private static void makeIdentifier(Element node, SimpleValue model,
        Mappings mappings) {
      // GENERATOR
```

Figure A.23: The patch generated (contd..)

```
@@ -2383,14 +2426,60 @@
      referenced.addProperty( ib );
    }
  }
+ public static void setUpManyToMany(Element node,Collection collection ,
+   java.util.Map persistentClasses , Mappings mappings ,
+   java.util.Map inheritedMetas , Attribute tableNode) {
+   String tableName;
+   tableName = mappings.getNamingStrategy().
+              tableName( tableNode.getValue() );
+   //Reading attributes for a many-to-many. These lines are from the
+   //original Hibernate code.
+   Attribute schemaNode = node.attribute( ''schema'' );
+   String schema = schemaNode == null ?mappings.getSchemaName() :
+                   schemaNode.getValue();
+   Attribute catalogNode = node.attribute(''catalog'');
+   String catalog = catalogNode == null ?mappings.getCatalogName() :
+                    catalogNode.getValue();
+   Table table = mappings.addTable(
+                 schema,
+                 catalog,
+                 tableName,
+                 getSubselect( node ),
+                 false
+                 );
+   collection.setCollectionTable( table );
+   bindComment(table, node);
+   collection.setElement(null);
+   log.info(
+     ''Converting one-to-many to many-to-many : Mapping collection: ''
+     + collection.getRole() +"->'' +
+     collection.getCollectionTable().getName());
+   }
```

Figure A.24: The patch generated (contd..)

```
  /**
   * Called for all collections
   */
  public static void bindCollectionSecondPass(Element node,
    Collection collection, java.util.Map persistentClasses,
    Mappings mappings, java.util.Map inheritedMetas)
    throws MappingException {
-
+   //Check whether a given collection is of type one-to-many.
+   if ( collection.isOneToMany() ) {
+     //Get the one-to-many node which was previously parsed from
+     //the mapping file
+     Element oneToManyNode = node.element( ``one-to-many'' );
+     //Get the attribute table from a given node
+     Attribute tableNode = node.attribute( ``table'' );
+     //explicitTableName : stores the value of the
+     //attribute tableNode if present
+     String explicitTableName = null;
+     //If there is a tableNode then get the value of the node
+     //and convert it to a lowercase for a comparision
+     if ( tableNode != null ) {
+       explicitTableName = mappings.getNamingStrategy().
            tableName( tableNode.getValue() ).toLowerCase();
+     }
+     //Get the name of the class (which comprises the collection)
+     //from a one-to-many tag, then get the name of the table for
+     //that class and convert it to a lower case for a comparision
+     String implicitTableName = mappings.getClass(
        getClassName(oneToManyNode.attributeValue(``class''),mappings))
        !=null? mappings.getClass(
+       getClassName(oneToManyNode.attributeValue(``class''),
          mappings)).getTable().getName().toLowerCase():null;
+     //if the user given name of the collection table is not same
+     //as the one computed by original Hibernate
+     //then the collection must be treated as a many-to-many with a
+     //unique constraint, and hence the type of the collection
+     //which was previously one-to-many is changed
+     if(explicitTableName != null && implicitTableName!= null &&
        !explicitTableName.equals(implicitTableName)){
+       setUpManyToMany(node, collection, persistentClasses,
+         mappings, inheritedMetas, tableNode);
+     }
+   }
```

Figure A.25: The patch generated (contd..)

```
     if ( collection.isOneToMany() ) {
       OneToMany oneToMany = (OneToMany) collection.getElement();
       String assocClass = oneToMany.getReferencedEntityName();
@@ -2459,7 +2548,7 @@
             mappings
         );
     }
-    else if (``many-to-many''.equals( name ) ) {
+    else if (``many-to-many''.equals( name ) ||
         (``one-to-many''.equals(name) && !collection.isOneToMany()) ) {
         ManyToOne element = new ManyToOne(
       collection.getCollectionTable() );
       collection.setElement( element );
       bindManyToOne(
```

Figure A.26: The patch generated (contd..)

BIBLIOGRAPHY

[1] http://en.wikipedia.org/wiki/SQL

[2] http://www.mkyong.com/hibernate/why-i-choose-hibernate-for-my-project/

[3] http://java.dzone.com/news/hibernate-best-choice

[4] Hibernate 3.2.6 Reference Manual

[5] http://iproving.ca/space/Technologies/Hibernate/SQL+vs+HQL+with+the+Session+Cache

[6] http://en.wikipedia.org/wiki/RDBMS

[7] Christian Bauer and Gavin King, "Java Persistence with Hibernate"

[8] http://java.sun.com/javase/technologies/database/

[9] http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx

[10] http://www.agiledata.org/essays/mappingObjects.html#MappingInheritance

[11] http://opensource.atlassian.com/projects/hibernate/browse/HHH-4077

[12] http://opensource.atlassian.com/projects/hibernate/browse/HHH-987

[13] http://opensource.atlassian.com/projects/hibernate/browse/HHH-3095

[14] http://www.aspfree.com/c/a/Database/Introduction-to-RDBMS-OODBMS-and-ORDBMS/1/

[15] http://www.acm.org/crossroads/xrds7-3/ordbms.html

[16] http://sourceforge.net/project/stats/detail.php?group_id=40712&ugn=hibernate&type=prdownload&mode=alltime&file_id=113508

[17] http://www.javalobby.org/articles/hibernatequery102/?source=archives