Reasoning About Java Persistence

Alan Barrett

B.Sc in Applied Sciences in Computer Science & Physics

A dissertation submitted in part fulfillment of the degree of
M.Sc. in Advanced Software Engineering
under the supervision of Dr. Joseph Kiniry

UCD

School of Computer Science and Informatics

February, 2008

Contents

Abstract

Over the years there have been a number of competing Java persistence frameworks in the commercial and open source worlds developed to enable Java persistence. The first attempt to standardise Java persistence was with the introduction of entity beans. Entity beans though were considered too heavyweight and complicated, and they could only be used in Java EE application servers. A number of lightweight persistence frameworks such as Hibernate and TopLink gained widespread adoption in the Java community due to the issues with entity beans. The introduction of the new standard for Java persistence, the Java Persistence API (JPA) has standardised these frameworks into one standard lightweight framework for use in all Java environments. Having a standard persistence framework which has been widely accepted by the Java community has opened the possibility of static and dynamic reasoning tools to build on the solid foundations of the JPA persistence framework.

This dissertation explores Java persistence with particular emphasis on the Java Persistence API (JPA) introduced as part of the EJB 3.0 specification. A case study on converting an existing commercial JEE Web application from using the Hibernate persistence framework to using JPA is presented. Particular emphasis in the case study is given to the use and meaning of Java annotations in JPA with a view to statically reasoning applications using JPA. Tools for assisting with the definition of object relational mappings are also discussed. Reasoning about Java persistence is explored using the Java Modeling Language (JML) to mathematically reason about database nullness and database column lengths for strings.

Acknowledgements

I would like to personally thank my supervisor Dr. Joe Kiniry for his time, encouragement, and enthusiasm. Joe it has been a pleasure working with you.

I'd also like to take this opportunity to thank the other staff of UCD who I have had the pleasure of learning from over the last two and a half years. This course has exceeded my expectations and has revitalised my interest and passion for computing.

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Persistence is defined as "the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created)" [4].

Modern programming languages, including Java, provide an intuitive, object-oriented view of application-level business entities, while the enterprise data underlying these entities is predominantly relational in nature. Many attempts have been made to bridge relational and object-oriented technologies, or to replace one with the other, but the gap between the two is one of the realities of enterprise computing today [3]. Persistence frameworks provide the bridge between relational data and Java objects, enabling developers to fully exploit the object-oriented paradigm to develop their software and fully utilise existing relational databases technology to persist the data.

The case study presented in chapter 3 to convert a commercial J2EE web application to using the Java Persistence API persistence (JPA) framework was undertaken to develop an application which could be used to test reasoning about Java persistence, and also to gain an understanding of the difficulties and challenges of using the new JPA persistence framework with commercial applications. JPA was chosen as the persistence framework for the case study as JPA is the new Java standard for persistence. JPA was developed by the leading Java persistence framework providers and has been widely accepted by the Java community.

Chapter 4 evaluates some of the Java tools available to support the development of JPA applications such as Dali [26] and MyEclipse [8].

Chapter 5 explores reasoning about Java persistence using the Java Modeling Language (JML) [1] to mathematically reason about database nullness and database column lengths for strings.

1.1 Outline of this Document

The rest of the dissertation is split into five chapters and appendices of sample source code. Chapter 2 presents a history of Java persistence, a high level overview of the Hibernate and JPA persistence frameworks, an overview of the Java annotations used in the case study, and an overview of extended static checking. Chapter 3 details the case study on converting an existing commercial J2EE Web application from using the Hibernate persistence framework to using the standard JPA framework. The issues identified in the conversion of the application are discussed, together with an evaluation of the conversion. Chapter 4 evaluates some of the Java persistence tools available to reduce the development time of JPA applications. Chapter 5 explores reasoning about database nullness and string column lengths in JPA applications. Chapter 6 presents the conclusions of this dissertation. The appendices list examples of the Hibernate XML mapping configuration files for the original case study application, JavaBeans with JPA annotations for the mappings, and Data Access Objects for the refactored case study application.

Chapter 2

Background

2.1 Introduction

This chapter presents a history of how Java persistence has developed from simple persisting data to files, to the new Java Persistence API (JPA) which is part of the EJB 3.0 specification. A high level overview of the Hibernate and JPA persistence frameworks used in the case study in chapter 3 is presented. A description of each of the JPA & EJB 3.0 Java annotations used in the case study is then listed, as well as an introduction to static checking with a view to statically reasoning JPA applications.

2.2 History of Java Persistence

The early versions of Java included persistence in the form of input/output streams that could be connected to external files stored on disk. This basic form of persistence allowed data to be saved to disk as data files but not as objects. JDK 1.1 provided the next step forward with the addition of Java object serialization and Java Database Connectivity (JDBC).

Java object serialization allows objects to saved to permanent storage mediums such as files in binary form, allowing objects to be deserialized at a later time recreating the objects from the stored binary form. The limitations of serialization stems from the fact that to serialize an object, all objects that the object refers to must be serialized at storage time. Using serialization an individual object cannot be modified without the entire byte stream of all referenced object being serialized and stored.

JDBC allows data to be stored on relational databases using the SQL language. JDBC can be used to store binary data such as images on relational databases but relational databases do not support the storage of Java objects.

The next steps built on the success of JDBC to try and solve the impedance mismatch between the object and relational model. The first attempt was SQLJ which allows SQL to

be embedded in Java programs. A pre-processor transforms the SQLJ program into a Java program with the JDBC calls added. Whilst SQLJ allows for the embedding of SQL it does not allow the developer to write the application purely in object form.

Object-Relational Mapping (ORM) frameworks also builds on JDBC but has the advantage that it allows the developer to develop purely in object form. The ORM framework in essence generates Java code which converts the objects into rows on relational tables and vice versa. Java 2 Enterprise Edition (J2EE) version one introduced the entity bean which was a very complex ORM persistence framework. The EJB 2.0 and 2.1 specification improved upon entity beans but there was not widespread adoption by the Java community due to its complexity and the rise of competing simpler to use lightweight ORM tools. The most popular early commercial implementations of the ORM framework was Oracle TopLink, with Hibernate leading the open source community implementation.

The most recent step has been the creation of the standardised Java Persistence API (JPA) which is part of the EJB 3.0 specification. JPA was designed by the leading ORM vendors (Hibernate, TopLink, JDO). Oracle provided TopLink Essentials which is the reference implementation for JPA. Hibernate have also released a version of Hibernate compatible with JPA. JPA has delivered POJO (Plain Old Java Object) persistence which greatly simplifies the development of persisted objects.

2.3 Hibernate Persistence Framework

The Hibernate project was started in 1999 by Gavin King, and in 2003 it was moved to the JBoss Corporation [9]. Hibernate is composed of three main modules; Hibernate core, Hibernate Annotations, and Hibernate EntityManager.

Hibernate Core is the foundation of Hibernate and the other Hibernate modules are built on Hibernate Core. It is composed of an API for persisting objects, object relational mapping data stored in XML format, Hibernate Query Language (HQL) for querying in a format similar to SQL but reference objects instead of tables, and a Criteria API for queries. Hibernate Core can run in a standalone Java SE environment, within a servlet container, or within a JEE compliant application server.

Hibernate Annotations builds on top of Hibernate Core. It allows developers to embed Java annotations directly into the Java source code replacing the XML mapping files. All of the standard JPA annotations are supported as well as a set of Hibernate extension annotations which support Hibernate features which are not part of JPA.

Hibernate EntityManager is another optional Hibernate module that can be used with Hibernate Core. It implements the JPA standard developer API for finding, adding, and deleting persistent Java objects. In essence Hibernate EntityManager is a small wrapper

around Hibernate Core that provides JPA compatibility [3].

To develop a standard JPA application using the Hibernate implementation, Hibernate Core and Hibernate EntityManager must both be used. If the application is to use the standard JPA annotations then all three modules must be used. In the case study presented in chapter 3 the original application only used Hibernate Core.

2.4 Java Persistence API (JPA) Persistence Framework

JPA version 1.0 is a Java persistence standard introduced as part of the EJB 3.0 specification to enable POJOs (Plain Old Java Objects) to be persisted in relational form on relational databases [22]. The JPA standard is composed of four main areas:

1. Object relational mapping information can be specified as annotations on the Java classes or using an XML configuration file.

2. Standard developer API for finding, adding, and deleting persistent Java objects.

3. Standard Java Persistence Query language to create queries against entities stored in a relational database. The syntax of the queries is similar to SQL except the table and column names are replaced with the object class and attribute names.

4. Different JPA implementations (such as TopLink Essentials and Hibernate JPA) can be used by applications without requiring changes to the application code.

JPA can be used standalone in the Java SE environment or within a Java EE compliant container. There are a number of JPA compliant implementations available such as TopLink Essentials, Hibernate, and Open JPA. In the case study presented in chapter 3 the converted application uses the TopLink Essentials implementation of JPA.

2.5 Java Annotations Used in Case Study

Java Annotations are metadata added to Java source code, and were incorporated in Java 1.5 under JSR 175, "A Metadata Facility for the Java Programming Language" [20]. Prior to this the the first type of metadata added to Java programs were special javadoc tags used to generate API documentation in HTML format from Java source code. JSR 220, "Enterprise JavaBeans 3.0" added the set of standard JPA and EJB3.0 annotations. The full list of these annotations can be found in JSR220 [21].

Table 2.1 lists each of the JPA annotations used in the case study with a short description.

Name	Description
@Entity	Specifies that the class is an entity to be persisted.
@Table	Specifies the primary table for the annotated entity.
@Column	Specifies a mapped column for a persistent property or field.
@Id	Specifies the primary key property or field of an entity.
@IdClass	Specifies a composite primary key class that is mapped to multiple fields or properties of the entity.
@Embeddable	Defines a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity.
@EmbeddedId	Is applied to a persistent field or property of an entity class or mapped superclass to denote a composite primary key that is an embeddable class. The embeddable class must be annotated as Embeddable.
@Temporal	Specified for persistent fields of type Date and Calendar.
@OneToOne	Defines a single-valued association to another entity that has one-to-one multiplicity.
@OneToMany	Defines a many-valued association with one-to-many multiplicity.
@ManyToMany	Defines a many-valued association with many-to-many multiplicity.
@JoinTable	This annotation is used in the mapping of associations. It is specified on the owning side of a many-to-many association, or in a unidirectional one-to-many association.
@NamedQuery	Specifies a named query in the Java Persistence query language, which is a static query expressed in meta data.
@PersistenceContext	Expresses a dependency on an EntityManager persistence context.

Table 2.1: JPA annotations used in case study

@Local	Declares the local business interface(s) for a session bean.
@Remote	Declares the remote business interface(s) for a session bean.
@Stateless	Component-defining annotation for a stateless session bean.
@EJB	Indicates a dependency on the local or remote view of an Enterprise Java Bean.

Table 2.2: EJB 3.0 annotations used in case study

Table 2.2 lists each of the EJB 3.0 annotations used in the case study together with a short description of the annotation.

2.6 Java Modeling Language (JML)

Java Modeling Language (JML) is a behavioral specification language that can be used to specify the behavior of Java modules. The aim of JML is to provide a specification language that is easy for Java developers and supported by a wide range of tools. Using JML a developer can specify the detailed design of Java classes and interfaces by annotating the source with special JML annotations.

JML annotations are written as special annotation comments in Java code using either `//@` or between `/*@ ... @*/` so that the Java compiler ignores the JML annotations. The core JML specifications are preconditions, postconditions, and invariants. All three specifications are expressed as boolean expressions in JMLs extension to the Java expression syntax.

The standard JML tool suite comprises of a type checker (jml), a JML compiler (jmlc) that compiles JML into runtime checks, a runtime assertion checker (jmlrac), an extended Javadoc tool (jmldoc) that generates documentation including JML specifications, and a unit test generator (jmlunit) [14].

2.7 Extended Static Checking (ESC)

Extended Static Checking (ESC) is the process of translation of the program source and its specifications into verification conditions; these are passed to a theorem prover, which in turn either verifies that no problems are found or generates a counter example indicating a potential bug [6]. ESC is not a replacement for unit testing but should be used in tandem with unit testing to help to earlier identify potential problems with the code.

The Extended Static Checker for Java version 2 (ESC/Java2) is an extended static checking tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal JML annotations [13]. ESC/Java2 performs two main functions. Firstly it identifies common programming errors such as null

pointer dereferences and invalid class casts. Secondly it attempts to verify the Java code is correct with respect to its JML specifications. Users can control the types and amount of checking the ESC/Java2 tool performs. The tool is available as a standalone command line tool or as an Eclipse plugin.

ESC/Java2 has been designed to be neither sound nor complete to keep the tool simple and to improve performance [11]. As the tool is unsound it emits false positives (misses errors actually present in the program it is analysing), and is it is unsound it emits false negatives (warns of potential errors when it is impossible for this error to occur in the execution of the program).

Chapter 3

Case Study

3.1 Introduction

This chapter presents the case study on converting an existing commercial JEE Web application from using the Hibernate persistence framework to using the standard Java persistence framework JPA.

The commercial application used in the case study is a JEE Web application used to receive parts and containers in plants across Europe for a multinational manufacturing company. To protect client confidentiality any references to the client has been removed from the application. The application has been renamed "Reasoning About Persistence", and is referred to as RAP in this document. The case study will focus on the User and Plant maintenance functionality of the Web application. The User maintenance functionality is used to maintain the information about the users authorized to use the system. The Plant maintenance functionality is used to maintain information about each of the plants using the application to receive parts and containers. A plant refers to a factory where the parts are assembled into the finished product.

3.2 Objective

The objective of the case study is to re-factor a real life Web application to use the standard Java persistence framework, which can then be used as a test application to test tools to reason about Java persistence. By utilising the standard Java persistence framework the application is no longer tied to a vendor specific persistence framework. The results section 3.8.1 discusses the issues identified in the re-factoring of the application to utilise JPA.

3.3 Requirements

The existing application is firstly to be converted from using Oracle 10g as its relational database, to using the open-source MySQL relational database. This is necessary to enable the existing MySQL database used by the Systems Research Group in UCD to be used in the case study. The existing application is to be fully retested after the application is ported from my Oracle to MySQL to ensure the application is functioning correctly prior to the conversion to use JPA.

The scope of the application to be used in the case study is limited to the plant and user maintenance functionality of the existing application. The decision to limit the scope was taken due to the size of the existing application and the limited time available to complete the case study. The scope of the application was also carefully chosen to ensure each of the main types of entity relationships (standalone, one-to-one, one-to-many, and many-to-many) were covered in the scope.

- The functional requirements of the application are that from a user perspective the application works as before.
- The non-functional requirements are:
  - The converted application is to use the TopLink Essentials implementation of JPA instead of the Hibernate framework as its persistence framework.
  - The existing database schema should be used.
  - The converted application is to replace the object relational mapping XML configuration files with JPA Java annotations. Also where appropriate EJB 3.0 Java annotations should be used. The use of the Java annotations is required to enable static reasoning of the application.

3.4 Architecture of Existing Application

The architecture of the application is composed of a JEE version 1.4 application server running WebLogic and a relational database running Oracle 10g. The applications uses the open source Apache Struts 1 Web application framework [7], to implement the model-view-controller (MVC) architectural pattern [18]. The application screens are written using JavaServer Pages (JSP) which generate dynamic HTML which is sent as a response to requests from the users browser [23]. The Data Access Object (DAO) design pattern is used to provide an interface to the persistence operations without exposing the persistence database framework [19]. Hibernate is the persistence framework used in the application to persist objects to the Oracle relational database. No EJB's are used in the application.

3.5 Architecture of Converted JPA Application

The architecture of the converted application is composed of a JEE version 5 application server running GlassFish, and a relational database running MySQL. The use of the Hibernate persistence framework is replaced by the JPA persistence framework to persist objects onto the MySQL database. No changes are required to the JavaServer Pages (JSP) or Struts configuration.

Stateless session EJBs are used instead of classes in the concrete DAO classes to enable the application to utilise container managed transactions (CMT) instead of the existing bean managed transactions (BMT). Using CMT simplifies reduces application effort by removing the need to specify begin and commit transactions explicitly.

3.6 Models

This section shows the the application domain model, and the physical data model of the RAP application. The domain model is a UML class diagram describing the various entities (classes) and the relationship between the entities. The physical data model describes how these entities are represented as tables in the physical database.

3.6.1 Domain Model

Figure 3.1 shows the domain model of the RAP application. Each entity in the model is implemented as an individual JavaBean with standard getter and setter methods. Table 3.1 describes each of the entities in the domain model. In the model the AuditLog entity has no relationship to other entities. The relationship between User and UserLogin is an example of a bidirectional one-to-one mapping. The relationship between Plant and PlantDock is an example of a bidirectional one-to-many mapping. The relationship between Plant and User is an example of a bidirectional many-to-many mapping.

3.6.2 Physical Data Model

Figure 3.2 shows the physical data model of the RAP application. Table 3.2 lists the entities and the table the entity is mapped to. The one-to-one mapping between the User and UserLogin entities has been implemented by the USER_IDENTIFIER column of the USER_LOGIN table which is the primary key and also a foreign key to the USER table. The bidirectional one-to-many relationship between Plant and PlantDock has been implemented by the PLANT_CODE column of the PLANT_DOCK table which is part of the compound primary key and also a foreign key to the PLANT table. The bidirectional many-

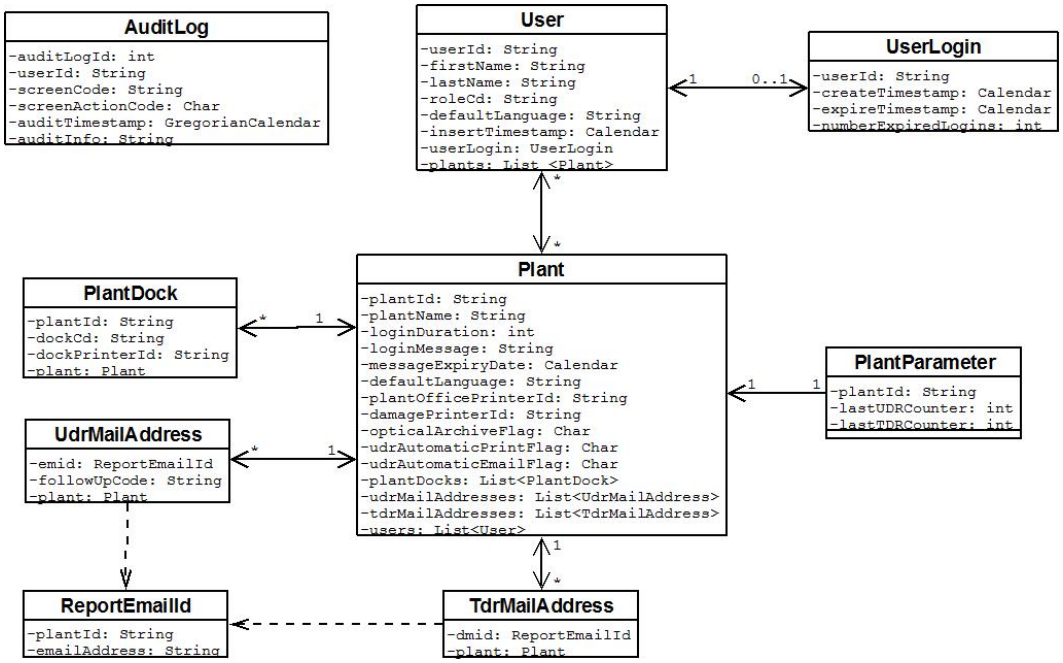

Figure 3.1: RAP domain model

Entity	Description
AuditLog	Represents an entity maintaining an audit trail of user activity in the application.
User	Stores information about the users of the application.
UserLogin	Stores login information about the users of the application.
Plant	Stores information about each assembly plant the application is used in.
PlantDock	Stores information about each dock (location) in the plant that parts are received in.
PlantParameter	Stores additional information about plants that is regularly updated by the application.
ReportEmailId	Represents the compound key of the UdrMailAddress and TdrMAilAddress entities.
UdrMailAddress	Represents a shipment report recipients e-mail address.
TdrMailAddress	Represents a damage report recipients e-mail address.

Table 3.1: Entity descriptions

to-many relationship between the User and Plant has been implemented by the use of the USER_PLANT join table.

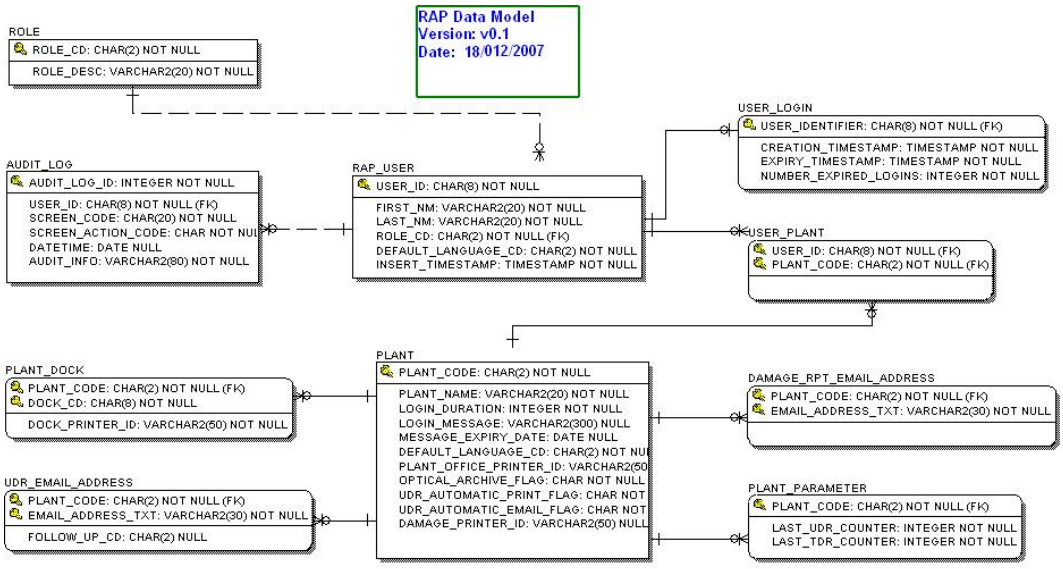

Figure 3.2: RAP physical data model

Entity	Table
AuditLog	AUDIT_LOG
User	RAP_USER
UserLogin	USER_LOGIN
Plant	PLANT
PlantDock	PLANT_DOCK
PlantParameter	PLANT_PARAMETER
UdrMailAddress	UDR_EMAIL_ADDRESS
TdrMailAddress	TDR_EMAIL_ADDRESS

Table 3.2: Entity to table mapping

3.7 Technical Design

The technical design section details the design to implement the requirements of the RAP application. The design was split into three steps:

1. Conversion of the application from using Oracle database to using MySQL database. This step was performed with the original application still using Hibernate. The existing application was fully retested after the application was re-configured to use MySQL.

2. Replacement of the Hibernate ORM XML configuration files with JPA Java annotations.

3. Replacement of the Hibernate API with the Java Persistence API.

3.7.1 Conversion from Oracle DB to MySQL DB

This subsection details the steps taken to configure the Hibernate application to use a MySQL database instead of Oracle:

1. Hibernate requires that rows on tables need to be uniquely identified. Using Oracle, uniqueness for rows is achieved using an Oracle sequence for the primary key column on the table. MySQL does not support sequences. MySQL has a AUTO_INCREMENT facility which can be specified on a column which must be the primary key. To configure Hibernate to use auto increment instead of the sequence, the generator class value is changed from "sequence" to "identity" in each of the Hibernate XML mapping files. E.g

```
<id name="auditLogId" column="AUDIT_LOG_ID" type="int">
<generator class="identity"/>
</id>
```

2. In the Hibernate.cfg.xml, replace the Oracle connection properties with the MySQL connection properties:

```
<property name="connection.url">
        jdbc:mysql://kind.ucd.ie:3306/database
</property>
<property name="connection.username">myUsername</property>
<property name="connection.password">myPassword</property>
<property name="connection.driver_class">
        com.mysql.jdbc.Driver
</property>
```

3. In Hibernate.cfg.xml specify class of MySQLDialect. This is important to set as otherwise Hibernate will not generate the correct SQL for the MySQL database. If the dialect is specified more than once in the XML file then the last one is used.

```
<property name="dialect">
        org.hibernate.dialect.MySQLDialect
</property>
```

3.7.2 Object Relational Mapping (ORM) Annotations

This subsection details the JPA Java annotations used to map the objects to the relational tables and vice versa. The object relational mapping annotations tells the persistence framework how the entities in the RAP domain model in Figure 3.1, are to be transformed into physical tables represented in the RAP physical data model in Figure 3.2. The ORM annotations were added to the existing JavaBeans used by Hibernate in the existing application. The use of the ORM annotations replaced the use of the individual Hibernate XML configuration files for each entity used in the existing Hibernate application.

3.7.2.1 Table Mapping

The first step to to persist an entity is to mark the class with the `@Entity` annotation. This tells the persistence engine that this class is to be persisted onto the database. The optional `@Table` annotation specifies the name of the table the entity is to be persisted to. If the `@Table` annotation is not present then the table name is assumed to be the same as the class name. Listing 3.1 shows how the AuditLog entity is persisted onto the AUDIT_LOG table. The AuditLog entity is used to log audit information on the use of the application.

Listing 3.1: Mapping an Entity to a table

```
@Entity
@Table(name = "AUDIT_LOG")
public class AuditLog implements Serializable {

...
}
```

3.7.2.2 Column Mapping

The next step is to map the class attributes to the table columns. The optional `@Column` annotation is used to map the attribute name to the column name of the table. If the `@Column` annotation is not present then the attribute name is assumed to be the same as the column of the table. The length and nullable elements of the `@Column` annotations are not used by the JPA framework but is used by the TopLink JPA DDL generation tool to automatically generate the database schema. Defining the length and nullable elements also allows for the opportunity to statically test the application to ensure the length and null setting of the attribute is never violated.

The optional `@Temporal` annotation is used to map temporal attributes to DATE, TIME, or TIMESTAMP column types. Listing 3.2 is an example of how the AuditLog attributes are mapped to the columns of the AUDIT_LOG table.

Listing 3.2: Mapping Attributes to Columns

```
@Entity
@Table(name = "AUDIT_LOG")
public class AuditLog implements Serializable
{
        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        @Column(name = "AUDIT_LOG_ID", nullable = false)
        private int auditLogId;

        @Column(name = "USER_ID", length = 8, nullable = false)
        private String userId;
        @Column(name = "SCREEN_CODE", length = 20, nullable = false)
        private String screenCode;
        @Column(name = "SCREEN_ACTION_CODE",length = 1,nullable=false)
        private char screenActionCode;
        @Column(name = "DATETIME")
        @Temporal(TemporalType.TIMESTAMP)
        private GregorianCalendar auditTimestamp;
        @Column(name = "AUDIT_INFO", length = 80, nullable = false)
        private String auditInfo;
...
}
```

3.7.2.3 Primary Key Mapping

Every entity to be persisted must have a mapping to a primary key in the table. The `@Id` annotation is used to identify the persistent identifier. The persistent identifier is the key that uniquely identifies an entity instance and distinguishes it from all other instances of the same class. For entities with compound (more than one) primary keys the `@IdClass` or `@EmbeddedId` annotations are used to define the embedded class that defines the primary key.

The optional `@GeneratedValue` annotation is used to specify that the JPA persistence provider is to automatically generate the persistent identifier value for every instance of the entity. The strategy element of the annotation is used to specify which strategy is to be used to generate the identifier. The three strategies are table id generation, database sequence generation, and database identity generation. The original application using Oracle used database sequence generation. MySQL does not support sequences so the database identity generation strategy has been used in the RAP application.

In listing 3.2 the auditLogId attribute is designated the persisted identifier of the AuditLog entity, and maps to the primary key column AUDIT_LOG_ID of the AUDIT_LOG table.

The next step is to map the relationships between the entities.

3.7.2.4 One-To-One Mapping

The relationship between User and UserLogin is an example of a bidirectional one-to-one mapping. The User entity stores the details of every user of the RAP application. The UserLogin entity stores additional login information about the user. For each User entity there is zero or one corresponding UserLogin entity, and for every UserLogin entity there must be one corresponding User entity.

The `@OneToOne` annotation is used to describe the one-to-one mapping between two entities. The entity with the join column is referred to as the owning side. The entity without the join column is referred to as the inverse side. Listing 3.3 shows the UserLogin class which is the owning side of the one-to-one relationship with the User class. The `@PrimaryKeyJoinColumn` annotation is used with the `@OneToOne` annotation, as the USER_IDENTIFIER column on the USER_LOGIN table is both the primary key and also the foreign key join column to the USER table. The user attribute of UserLogin contains a reference to the User object that UserLogin is mapped to.

Listing 3.3: One-to-one Relationship from UserLogin to User

```
@Entity
@Table(name = "USER_LOGIN")
public class UserLogin implements Serializable {
        @Id @Column(name = "USER_IDENTIFIER", length = 8)
        private String userId;

        @OneToOne
        @PrimaryKeyJoinColumn
        private User user;
...
}
```

The inverse mapping of the one-to-one mapping is displayed in listing 3.4. The “mappedby” element of the *@OneToOne* annotation is used by the inverse side to indicate the owning side of the relationship. The value of the “mappedby” element must be the attribute name of the owning entity containing the join column.

3.7.2.5 Many-To-One Mapping

The relationship between PlantDock and Plant is an example of a bidirectional many-to-one mapping. The Plant entity stores the details of every plant that the RAP application is used to receive parts and containers. The PlantDock entity stores information about each individual dock in the plant. The dock is the area in the plant where the carriers arrive to unload their shipment of parts. For each Plant entity there can be many related PlantDock entities, and for every PlantDock entity there must be one corresponding Plant entity.

Listing 3.4: One-to-one inverse Relationship from User to UserLogin

```
@Entity
@Table(name = "RAP_USER")
public class User implements Serializable {
        @Id @Column(name = "USER_ID", length = 8, nullable = false)

        @OneToOne(mappedBy="user", fetch=FetchType.EAGER)
        private UserLogin userLogin;
...
}
```

The `@ManyToOne` annotation is used to describe the many-to-one mapping between two entities. The many-to-one side of the relationship has the join column, and is referred to as the owning side. Listing 3.5 shows the PlantDock class which is the owning side of the many-to-one relationship with the Plant class. The *plant* attribute of PlantDock contains a reference to the Plant object that PlantDock is mapped to.

Listing 3.5: Many-to-one Relationship from PlantDock to Plant

```
@Entity
@Table(name = "PLANT_DOCK")
@IdClass(PlantDockId.class)
public class PlantDock implements Serializable {
        // insertable,updateable=false needed to ensure mapping read only
        @Id@Column(name="PLANT_CODE",insertable=false,updatable=false
                        ,length = 2)
        private String plantId;
        @Id@Column(name="DOCK_CD", length = 8, nullable = false)
        private String dockCd;

        // needed to resolve duplicate insert using key columns;
        // overrides earlier mapping for PLANT_CODE
        @ManyToOne
        @JoinColumn(name="PLANT_CODE")
        private Plant plant;
...
}
```

3.7.2.6 One-To-Many Mapping

The inverse mapping of the many-to-one mapping is displayed in listing 3.6. The `@OneToMany` annotation is used to describe the one-to-many mapping between two entities. Similar to the one-to-one mapping, the "mappedby" element of the `@OneToMany` annotation is used by the inverse side to indicate the owning side of the relationship. The *plantDocks* attribute of Plant contains a list of references to the PlantDock objects that the Plant entity is mapped

to.

Listing 3.6: One-to-many Relationship from Plant to PlantDock

```
@Entity
@Table(name = "PLANT")
public class Plant implements Serializable {
        @Id @Column(name = "PLANT_CODE", length = 2, nullable = false)
        private String plantId;

        @OneToMany(fetch=FetchType.EAGER, mappedBy="plant",
                                cascade=CascadeType.ALL)
        private List <PlantDock> plantDocks;
...
}
```

3.7.2.7 Many-To-Many Mapping

The final relationship mapping is the many-to-many mapping. This is the least used relationship mapping. The relationship between Plant and User is an example of a bidirectional many-to-many mapping. The relationship is a many-to-many mapping as a plant can have many authorised users, and a user can be authorised to use the RAP application in many plants.

The `@ManyToMany` annotation is used to describe the many-to-many mapping between the two entities. The `@JoinTable` annotation is used to describe the join table used to implement the many-to-many-mapping. Listing 3.7 shows the User class which is the owning side of the many-to-many relationship with the Plant class. In a many-to-many mapping relationship either side of the relationship can be designated the owner as neither side has a join column due to the use of the join table. The plants attribute of User contains a list of references to the Plant objects that the user is authorised to use the application.

The inverse mapping of the many-to-many mapping is displayed in Listing 3.8. The "mappedby" element of the `@ManyToMany` annotation is used by the inverse side to indicate the owning side of the relationship. The value of the "mappedby" element must be the attribute name of the owning entity containing the `@JoinTable` annotation describing the join table. The users attribute of Plant contains a list of references to the User objects that are authorised for that plant to use the RAP application.

3.7.3 Entity Manager Operations

The entity manager is at the core of JPA. It provides the API to persist, remove, and find objects on/to the database. The persistence framework ties the entity manager API to the object relational mappings to perform persistence operations. In the RAP application the

Listing 3.7: Many-to-many Relationship from User to Plant

```
@Entity
@Table(name = "RAP_USER")
public class User implements Serializable {

        @Id @Column(name = "USER_ID", length = 8, nullable = false)
        private String userId;

        @ManyToMany (cascade=CascadeType.PERSIST, fetch=FetchType.EAGER)
        @JoinTable(name = "USER_PLANT",
                joinColumns = @JoinColumn(name = "USER_ID"),
                inverseJoinColumns = @JoinColumn(name = "PLANT_CODE"))
        private List <Plant> plants;
...
}
```

Listing 3.8: Many-to-many Inverse Relationship from Plant to User

```
@Entity
@Table(name = "PLANT")
public class Plant implements Serializable {
        @Id     @Column(name = "PLANT_CODE", length=2, nullable=false)
        private String plantId;

        @ManyToMany(fetch=FetchType.EAGER, mappedBy="plants",
                                cascade=CascadeType.ALL)
        private List<User> users;
...
}
```

existing data access objects (DAO) classes were modified to use the JPA. In the Hibernate API the Hibernate Session provides the API to persist, remove, and find objects on/to the database. Converting from the Hibernate API to JPA is in most cases straightforward. In this subsection the main persistence API are discussed.

3.7.3.1 EJB 3.0 Dependency Injection

Dependency injection is the process of automatically looking up a container resource such as a database connection, and setting it into the class [12]. The container injects the resolved dependency into the class. This process removes the need to manually lookup container resources via Java Naming and Directory Interface (JNDI), and has been added within the EJB 3.0 specification.

There are two forms of dependency injection, field injection and setter injection. Field injection involves the server looking up the dependency in the environment naming context, and assigning the result directly into the annotated field of the class. An example of field

injection is when the `@PersistenceContext` annotation is used to inject an entity manager into a EntityManager object. Setter injection involves annotating a setter method instead of a field. In the case study only field injection is used.

3.7.3.2 Obtaining an Entity Manager

To obtain a container-managed entity manager in JPA the `@PersistenceContext` annotation is used to tell the container to inject an entity manager into the *entityManager* field. The ”unitName” element specifies the name of the persistence unit defined in the persistence.xml configuration file. Listing 3.9 demonstrates how the stateless AuditDAOBean obtains an entity manager.

Listing 3.9: Obtaining a container-managed entity manager

```
@Stateless
public class AuditDAOBean implements AuditDAO, AuditDAORemote {

        public AuditDAOBean()m{
        }

        @PersistenceContext(unitName="jpa_rap2")
        private EntityManager entityManager;
...
}
```

3.7.3.3 Conversion from Bean Managed Transactions to Container Managed Transaction

The RAP application Data Access Objects (DAO) were converted to stateless session beans to simplify the database transaction coding. With the use of stateless session beans and dependency injection, the persistence context managing the managed objects is automatically propagated across a Java Transaction API (JTA) transaction to any other stateless session DAO beans that this DAO may create. The advantage of using container managed transactions is that there is no need for the developer to specifically request the transaction to begin or commit or rollback. When the DAO method is completed the database transaction is automatically committed to the database if the container has not marked the transaction to be rolled back. Listing 3.10 is an example of how the RAP application persists a plant to the database using container managed transactions.

In the original Hibernate application, the DAO were simple classes which made no use of the container to manage the transaction or use of dependency injection to obtain the Hibernate Session. Listing 3.11 is an example of how the Hibernate application persisted a plant to the database by use of bean managed transactions.

Listing 3.10: Container Managed Transaction

```
@Stateless
public class PlantDAOBean implements PlantDAO, PlantDAORemote {

@PersistenceContext(unitName="jpa_rap2")
private EntityManager entityManager;

public void addPlant(Plant newPlant) throws Exception {
        try {
                entityManager.persist(newPlant);
                entityManager.flush();
        }
        catch (PersistenceException pe) {
                throw new RAPDAOException(pe,
                   "error.Log.PlantDAOaddPlant", newPlant.getPlantId());
        }
        catch (Exception e) {
                throw e;
        }
        finally {
        }
}
...
}
```

3.7.3.4 Persistence Exception Handling

In JPA the *PersistenceException* is thrown by the persistence provider when a problem occurs. All instances of PersistenceException except for instances of NoResultException and NonUniqueResultException causes the current transaction, if one is active, to be marked for rollback. In Hibernate persistence, the *HibernateException* is thrown by Hibernate when a problem occurs. Re-factoring the exception handling was very straightforward as the use of the *HibernateException* was replaced with *PersistenceException*.

The `flush()` method of the EntityManager interface is performed in all the Data Access Objects to synchronise the persistence context to the underlying database. The flush() method tells the persistence provider to run any pending SQL against the database. Performing the flush() at the end of each DAO ensures that should a PersistenceException be thrown the exception will be thrown in the DAO that caused the exception, enabling more detailed information about the cause of the problem to be logged by the DAO exception handling routine.

There is one subtle difference between Hibernate and JPA when an object was not found on the database. In Hibernate a *HibernateException* (ObjectNotFoundException) is thrown when a `Session.load()` fails to select a row with the given primary key (identifier value), whereas in JPA no exception is thrown and the returned object is set to null. This

Listing 3.11: Bean Managed Transaction

```
public class PlantDAO
{
public void addPlant(Plant newPlant) throws Exception
{
        Transaction tx = null;
        Session  localSession = HibernateHelper.getSession();
        try {
                tx = localSession.beginTransaction();
                localSession.save(newPlant);
                tx.commit();
        }
        catch (HibernateException he)
        {
                if (tx != null) tx.rollback();
                Logger.getLogger().logError("error.Log.PlantDAOaddPlant",
                     he, newPlant.getPlantId());
                throw new DAOException(he, newPlant.getPlantId());
        }
        catch (Exception e) {
                if (tx != null) tx.rollback();
                        throw e;
        }
        finally {
                tx = null;
        }
}
...
}
```

necessitated some minor changes to the code to check if the result was null instead handling the exception.

3.7.3.5 Persist an Object

The `persist()` method of the entity manager interface is used to persist an object. Any POJO (Plain Old Java Object) whose class has been marked as a persistent class can be passed as a parameter to the `persist()` method. Listing 3.12 lists the `auditEvent()` method which persists AuditLog objects to the AUDIT_LOG table.

In Hibernate the `save()` method of the Hibernate Session interface is used to persist an object to the database.

Listing 3.12: Method to persist an AuditLog object

```
public void auditEvent(String userId,String screenCode,char eventCode,
                GregorianCalendar gc,String auditMessage)
                throws Exception {

        AuditLog auditLog = new AuditLog(0,userId,screenCode,
                eventCode,gc,auditMessage);

        try {
                entityManager.persist(auditLog);
        }
        catch (PersistenceException pe) {
                throw new RAPDAOException(pe,
                                "error.Log.AuditError", screenCode);
        }
}
```

3.7.3.6 Find an Object

The `find()` method of the entity manager interface is used to load an object. Any POJO (Plain Old Java Object) whose class has been marked as a persistent class can be passed as a parameter to the `find()` method together with its primary key value. Listing 3.13 lists the `getPlant()` method which returns a managed entity instance of the plant read from the PLANT table.

In Hibernate the `get()` or `find()` method of the Hibernate Session interface is used to load an object from the database.

Listing 3.13: Method to find a plant object

```
/**
        * get a Plant object for the passed in plantID.
        * Catch and log any failures.
        * @param plantId : ID of plant to be retrieved.
*/
public Plant getPlant(String plantId) throws Exception {
        Plant thisPlant = null;
        try {
          thisPlant= (Plant) entityManager.find(Plant.class,plantId);
        }
        catch (PersistenceException persistenceException) {
                throw new RAPDAOException(persistenceException,
                        "error.Log.PlantDAOgetPlant", plantId);
        }
        catch (Exception e)
        {
                throw e;
        }
        finally {
        }
        return thisPlant;
}
```

3.7.3.7 Update an Object

The `merge()` method of the entity manager interface is used to update an object which has previously been persisted to the database. Any POJO (Plain Old Java Object) whose class has been marked as a persistent class can be passed as a parameter to the `merge()` method. Similarly in Hibernate the `merge()` method of the Hibernate Session interface is used to update a persisted object to the database.

3.7.3.8 Delete an Object

The `remove()` method of the entity manager interface is used to delete a persisted object. Any POJO (Plain Old Java Object) whose class has been marked as a persistent class can be passed as a parameter to the `remove()` method. Listing 3.14 lists the `deletePlant()` method which deletes the persisted plant object from the Plant table whose persistent identifier is passed as parameter to the method.

In Hibernate the `delete()` method of the Hibernate Session interface is used to delete a persisted object from the database.

Listing 3.14: Method to delete a persisted Plant object from Plant table

```
public void deletePlant(String plantId) throws Exception {
        Plant thisPlant = null;

        try {
                thisPlant = entityManager.find(Plant.class, plantId);
                entityManager.remove(thisPlant);
                entityManager.flush();
        }

        catch (PersistenceException pe) {
        throw new RAPDAOException(pe,
                        "error.Log.PlantDAOdeletePlant", plantId);
        }
}
```

3.7.4 Crtieria API

Hibernate features an criteria query API. The `createCriteria()` method of the Hibernate Session interface is used to create a Hibernate Criteria on the object. Restrictions can be added to the criteria to only select object matching the criteria. The list can also be ordered by use of the `addOrder()` method. Listing 3.15 is an example of a Hibernate criteria query to get all plants ordered by ascending plant id.

Version 1.0 of JPA used in the case study does not support the criteria query API. All use of the criteria query in the original application was changed to use JPA queries in the refactored application. Version 2.0 of JPA under JSR 317 “Java Persistence 2.0” has proposed the inclusion of the Criteria API within the standard persistence framework [25].

3.7.5 Query Annotations

JPA supports two query languages for retrieving persistent data from the database. The first and primary query language is the Java Persistence Query Language (JPQL) which is a database-independent query language that operates on the object attributes rather than the columns in the tables. The other is standard SQL queries which are run directly against the database. In this case study only JPQL is used.

Hibernate also has its own Hibernate Query Language (HQL) which is similar to JPQL and enables queries using inheritance, polymorphism and association. The original application used the Hibernate criteria API instead of HQL to retrieve objects from the database.

Queries in JPQL can be constructed into two ways. The first way is Dynamic Query Definition where the query is defined dynamically by passing the query string to the *createQuery()* method of the EntityManager interface. The second way is Named Query Defini-

Listing 3.15: Example Hibernate Criteria API method to to get all plants

```
public ArrayList getAllPlants() throws Exception {
        Session localSession = HibernateHelper.getSession();
        ArrayList allPlants = new ArrayList();
        try {
                Criteria crit = localSession.createCriteria(Plant.class);
                //sort by ascending plant code
                crit.addOrder(Order.asc("plantId"));
                List listOfUserPlants = crit.list();
                allPlants = (ArrayList) listOfUserPlants;
        }
        catch (HibernateException he) {
                Logger.getLogger()
                  .logError("error.Log.PlantDAOgetAllPlants", he);
                throw new DAOException(he);
        }
        catch (Exception e) {
                throw e;
        }
        finally {
        }
        return allPlants;
}
```

tion which was used for all queries in the refactored application. The `@NamedQuery` and `@NamedQueries` annotations are used to define named queries and can be placed on the class definition for any entity. Listing 3.16 is an example of how the "Plant.getAllPlants" named query to retrieve all plant objects from the database is defined. To execute the named query the *createNamedQuery()* method of the EntityManager interface is used to create and execute the query. Listing 3.17 is an example method which creates and executes the named query "Plant.getAllPlants".

Listing 3.16: Annotation to defined Named Query

```
@NamedQuery(name = "Plant.getAllPlants",
        query = "Select␣p␣" + "FROM␣Plant␣p␣" +
                "ORDER␣BY␣p.plantId")
```

3.8 Testing

The functional testing of the application was performed by manually entering test data on the plant and user maintenance screens. The tests cases performed on the existing application were repeated on the new RAP application until all functional tests passed.

Listing 3.17: Method to create and execute Named Query

```
public Vector <Plant> getAllPlants() throws Exception {

        Vector <Plant> allPlants = new Vector<Plant>();
        try {
                List <Plant> listOfUserPlants =
                  entityManager.createNamedQuery("Plant.getAllPlants")
                    .getResultList();
                allPlants = (Vector <Plant>) listOfUserPlants;
        }
        catch (PersistenceException pe) {
                throw new RAPDAOException(pe,
                  "error.Log.PlantDAOgetAllPlants");
        }
  catch (Exception e) {
                throw e;
        }
        finally {
  }
        return allPlants;
}
```

The Results subsection discusses the issues identified during testing of the RAP application.

3.8.1 Results

The biggest issues from the testing phase were in creating instances of stateless session DAO beans, and implementing one-to-many & many-to-one relationships between entities:

1. During initial testing of the re-factored application problems were encountered in the use of the `@EJB` dependency injection annotation to inject the stateless session bean DAO objects into the Struts Action classes. The problem was due to JAVA EE 5 limiting dependency injection only to managed classes such as EJBs, Interceptors, and Servlets. Amending the Struts Actions classes to use the Java Naming and Directory Interface (JNDI) to create instances of the stateless session DAO beans resolved the problem.

2. The most difficult issue to resolve was in creating the one-to-many mappings between entities. In the original application there were a number of unidirectional one-to-many relationships, e.g. relationship between Plant and PlantDock. Version 1.0 of JPA only supports unidirectional one-to-many relationships between entities if the relationship is implemented by use of a join table containing the primary keys of both entities. Hibernate on the other hand allows unidirectional one-to-many relationships

between entities to be implemented by use of a foreign key. As one of the requirements of the case study was to use the existing database schema it was decided to make all the one-to-many mappings in the RAP application bidirectional rather than creating additional join tables to implement the mapping. The bidirectional mapping between Plant and PlantDock for example was achieved by adding the @ManyToOne mapping to PlantDock in Listing 3.5, and adding the "mappedBy" parameter to the @OneToMany annotation in Listing 3.6 to make the mapping bidirectional. This deficiency of JPA is being addressed in version 2.0 of JPA under JSR 317 which will allow unidirectional one-to-many to use foreign keys mappings to implement the relationship, thereby removing the need to make the one-to-many mappings bidirectional.

3. Issues were also identified when testing with many-to-one entity relationships where the join column implementing the mapping was also a primary key or part of a compound primary key. An example of this problem was when adding a new dock for a plant on the plant maintenance screen. This resulted in a PersistenceException being thrown due to a duplicate insert into PLANT_DOCK table.

   The problem was identified to be JPA treating the primary key of the PlantDock entity and the foreign key join column of the many-to-one mapping to the Plant entity as two separate mappings to the same column on the PLANT_DOCK table. When a new PlantDock object was persisted to the database the two mappings collided causing the duplicate insert.

   In JPA 1.0 there is no standard means of specifying in a many-to-one relationship that the compound primary key is also a foreign key. There is a work around solution where the primary key mapping of the column which is both a primary and foreign key can be set to read-only. This is achieved by setting the `insertable` and `updatable` parameters of the `@Column` annotation to be false. Listing 3.5 demonstrates the solution by making the plantId attribute a read-only mapping to the PLANT_CODE column on the PLANT_DOCK table thereby avoiding a mapping collision with the `@ManyToOne` mapping to the PLANT_CODE column. With this work around solution the duplicate insert onto the PLANT_DOCK table no longer occurred.

A number of minor issues from the testing phase are as follows:

1. Problems were encountered deploying the application as an Enterprise ARchive (EAR) file due to Windows version of GlassFish application server installation path including spaces. This issue was resolved by reinstalling the GlassFish application to a directory path not including whitespaces.

2. Problems running the application on the GlassFish server were also due to the application server being unable to find the Xerces class. This was resolved by removing the xerces.jar & xercesimpl.jar from the WEB-INF/lib of the deployed Web application.

3. In the User and Plant DAOs there were initial problems in linking the User and Plant entities together when a User or Plant entity was persisted. Resolution was to ensure for many-to-many relationships that both sides are updated to point to each other before the merge operation is performed.

4. Problems using connection pooling from behind the UCD database firewall was also identified. This issue was resolved by reducing the idle timeout period that connections can remain idle in the pool from 300 to 10 seconds.

3.9 Evaluation

Whilst all the requirements of the case study were fully met a number of design compromises had to be made due to functionality in the Hibernate framework not being part of JPA version 1.0 :

- To implement the one-to-many relationships whilst not changing the database schema to add join tables required making the one-to-many relationships bidirectional. Version 2.0 of JPA is expected to remove this requirement and enable unidirectional one-to-many mappings using foreign keys.

- JPA version 1.0 not having a Criteria API was less of an issue and required the use of the Java Persistence Query Language. With version 2.0 of JPA incorporating a Criteria API this issue will be resolved in the near future.

Having to manually retest the application after each code change was a major issue. Having automated unit testing such as the JUnit framework to write repeatable tests for the original application would have significantly reduced the time to refactor the application using JPA.

Use of annotations was found to be much more intuitive than maintaining separate XML files with the object relational mappings for the persisted entities. Whilst doing the annotations for USER_PLANT it became obvious that it would be much simpler to use a many-to-many mapping directly between the User and Plant entities, rather than using intermediate one-to-many mappings between the User & UserPlant, and Plant & UserPlant.

Having all the object relational mapping contained in standard Java annotations on the JavaBeans opens up the possibility of utilising automated tools such as ESC/Java2 to statically test the application.

The next chapter will evaluate a number of persistence tools to speed up development of JPA applications.

Chapter 4

Java Persistence Tools

4.1 Introduction

This chapter evaluates some of the Java persistence tools available to assist in the development time of JPA applications.

4.2 Dali

The goal of the Dali version 1.0 JPA Tools Project is to provide an Eclipse-based tool to define and edit standard JPA Object-Relational Mappings [26]. Dali is a sub-project of the Web Tools Platform Project. It also provides initial mapping wizards that automatically create entity classes from an existing database.

To evaluate the Dali tool, the "Generate Persistent Entities from Tables" wizard in Dali was tested with the case study RAP database. The individual JavaBeans (Entities) were created with the JPA annotations to map the objects to the tables, but none of the more complicated JPA annotations to map one-to-one, one-to-many, or many-to-many relationships were mapped in the generated JavaBean entities. Using the Dali mapping wizards the relationship mappings can be manually created and edited. The Dali tool also offered no facility to automatically create the Data Access Objects from the database. Overall the tool scores highly on presentation offering graphical representations of the entities in the JPA Structure view and being open source. The tool though scores low on automated mapping generation facilities. The tool is particulary suitable for developers new to JPA.

4.3 MyEclipse

MyEclipse is a commercial version of Eclipse [8]. It provides an integrated JPA development environment with support for the Toplink Essentials and Hibernate 3.2 JPA implemen-

tations. It is functionally similar to Dali but with additional wizards that not only generate the entity classes from an existing database, but also generated the associated data access objects.

To evaluate the MyEclipse tool, a free one month demonstration licence of MyEclipse version 6.0 was used. The "MyEclipse JPA Reverse Engineering" wizard was tested using the case study RAP database. The tool generated the individual JavaBeans (Entities) with the correct JPA annotations to map the objects to the tables, and also generated all of the more complicated JPA annotations to map the one-to-many, many-to-one and many-to-many relationships between the entities. The tool also generated individual DAOs for each entity with methods to enable pagination for query results. The tool handled the two main testing issues identified in the case study correctly, by making all one-to-many mappings bidirectional, and ensuring no mapping collisions in many-to-one relationship. The tool also generates a "EntityManagerHelper class" to handle basic methods to create EntityManager instances and database transaction methods. Overall the tool has excellent presentation and automated tools. The tool is suitable for developers new to JPA and experienced JPA developers, though a yearly licence fee for the tool is required.

4.4 Schema Comparison Tool

The schema comparison tool is a tool to identify differences between an existing database schema and the schema the JPA entities represent. By highlighting differences in the expected schema, it highlights the areas the developer need to look at for potential problems in how the objects are being mapped to the database. The TopLink DDL generation tool generates the DDL schema for a persistence application at deployment time to the application server. During testing of the RAP case study application additional join columns were noticed in the generated schema for entities with many-to-one mappings when the mappings were incorrectly specified. By comparing the schema generated by TopLink to the expected schema, errors in the use of the JPA annotations are easily spotted. The `diff` file comparison tool or similar tools can be used to easily highlight the difference between the schemas.

4.5 Hibernate Validator

Hibernate Validator allows developers to specify invariant constraints for a domain model [10]. An invariant constraint is a rule that a given element (field, property or bean) has to comply to. The constraints are defined by use of Hibernate Validator specific annotations which similar to the JPA annotations are embedded in the JavaBeans defining the persistent enti-

ties. The tool can be used as an add on to the JPA persistence framework allowing objects to be validated according to Hibernate Validator specific annotations, before the object is persisted to the database. When used with the Hibernate implementation of the JPA framework it can automatically add constraints to the database schema (DDL generation) according to the Hibernate Validator specific annotations used to define the constraints on the object. The developers of Hibernate Validator are contributing to JSR 303 "Bean Validation" which is expected to standardise Java annotations for JavaBean validation [24].

Listing 4.1 is an example of how additional constraints can be added to the AuditLog entity using Hibernate Validator. The `@NotNull` annotation on attribute *userId* checks that the *userId* is not null before the entity is persisted to the database. The `@Length` annotation checks if the string length match the range prior to persisting the entity to the database.

Listing 4.1: Hibernate Validator example

```
@Entity  @Table(name = "AUDIT_LOG")
public class AuditLog implements Serializable
{
        @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
        @Column(name = "AUDIT_LOG_ID", nullable = false)
        private int auditLogId;

        @Column(name = "USER_ID", length = 8, nullable = false)
        @NotNull  // Hibernate Validator annotation to check not null
        @Length(max = 8) //Hibernate Validator annotation to check length
        private String userId;

        @NotNull  // Hibernate Validator annotation to check not null
        @Column(name = "SCREEN_CODE", length = 20, nullable = false)
        @Length(max = 20) // Hibernate Validator annotation
        private String screenCode;

        @Column(name="SCREEN_ACTION_CODE", length=1, nullable=false)
        private char screenActionCode;

        @Column(name = "DATETIME")
        @Temporal(TemporalType.TIMESTAMP)
        private GregorianCalendar auditTimestamp;

        @Column(name = "AUDIT_INFO", length = 80, nullable = false)
        @Length(max = 80) //Hibernate Validator annotation to check length
        @NotNull  // Hibernate Validator annotation to check not null
        private String auditInfo;
...
}
```

Chapter 5

Reasoning about Persistence

5.1 Introduction

Reasoning about Java persistence builds on the extensive research done already on reasoning about Java classes [2] [17] [5] [16]. In the context of Java persistence, reasoning is the process of understanding how the Java program persists its objects and then determining whether this matches with the developers specification. By being able to reason about persistence in Java programs, the correctness or incorrectness of the implementation of persistence can be automatically determined by use of theorem provers. This will lead to higher quality persistence applications with fewer database runtime exceptions.

This chapter explores reasoning about persistence with particular emphasis on reasoning about database nullness, and database column lengths for strings.

5.2 Reasoning about Database Column Lengths

Columns of type `String` are represented as either CHAR or VARCHAR data types on databases. Both data types have an associated length attribute which determines the maximum length of characters that will be stored on the database. As previously discussed in Section 3.7.2.2, the length attribute of the JPA `@Column` annotation is used to specify the maximum number of characters the column on the database can store. In JPA version 1.0, the length attribute is only used by the TopLink JPA DDL generation tool when generating the database schema at time of deployment. This section will discuss how JML can be used to reason about database column lengths.

5.2.1 JML Invariants to Capture Database Length Attribute

To model database length in the Java Modeling Language (JML), the length attribute of the JPA `@Column` is translated into a JML invariant. A JML invariant is a predicate about a class field that is always true. In Figure 5.1 an invariant on the userId attribute specifies that the length of the userId String must always be less than or equal to eight characters long.

Listing 5.1: JML Invariant to Capture Database Length

```
@Column(name = "USER_ID", length = 8, nullable = false)
//@ invariant userId.length() <= 8
private String userId;
```

5.2.2 Testing

The JML invariant can be used with the JML suite of tools to create static checks, unit tests, and runtime checks to verify that the length of the string is never greater than the maximum length allowed. Extended static checkers such as ESC/Java2 statically check the Java program to ensure the invariant is always true. The JML unit test generator (jmlunit) generates unit test cases from the JML invariant, which can be used to unit test the program. The JML compiler (jmlc) verifies at runtime that the predicate is always true by translating the JML invariant into assertions which are checked at runtime [5].

5.3 Reasoning about Database Nullness

This section will discuss how to reason about database nullness using JML. The meaning and ambiguity of database Null or Unknown is first discussed, followed by how JML models can be used to mathematically to specify Null. A means of tracking whether a persisted class has been persisted or not, via a JML model field is then discussed. An example is presented which ties the individual pieces together to perform the reasoning.

5.3.1 SQL's Notion of Nullness

A column of a database can be set to either "Null" or "NOT Null". ISO SQL Null defines that a data value on a column marked as Null is "Unknown". Null is different from an empty string or the numerical value 0.

5.3.1.1 Ambiguities in the Semantics of Unknown

There are a number of ambiguities in the semantics of "Unknown":

- For an “int” or “char” type what does it mean to be unknown ? Null cannot be set to primitive data types such as “int” and “char” in Java, so a default value must be used to represent “Null” in these types.
- When an object is deserialised from columns in a table with nullable columns, what will happen to the attributes of the object which are primitive Java types such as int or char ? Should “Null” for a field on a database which is mapped to a attribute of type “int” be set to zero when the object is deserialized ? The most prudent solution would be to set all basic Java types to the same value they would be set to, if no initial value was specified at the time the object was instantiated. For example, an int would be default to zero.
- In programming languages such as Java, a null object reference means that the object reference is not pointing to any object. This is different to the SQL notion of null where null means the value is unknown at that moment in time, and its value should not be used. This mismatch between the programming language and the relation models notion of null can easily lead to bugs in applications persisting objects to databases.

These and other ambiguities need further research to resolve.

5.3.1.2 Models in JML

JML models are mathematical abstractions used to specify abstract behavior independent of implementation. All Java core classes such as Float, Char etc. have an associated JML model class which can be used to verify Java program implementations with respect to the JML specification. JML model fields are variables that can only be used in behavior specifications and have no effect on the actual program implementation. JML model classes and fields are defined using the `model` keyword in the appropriate place, preceding the definition of the class or field, respectively.

5.3.1.3 Three-valued Logics

Three-valued logics is a term describing any of several multi-valued logic systems in which there are three truth values indicating true, false and some third value. As SQL Null means the value is “Unknown” at that time, comparison of real values with Null not only need to account for true and false, but also need to account for this additional third value state of `Unknown`. SQL’s three-valued logic is Kleene’s three-valued logic as the `Unknown` value in SQL is expected to be known at some time in the future [15].

5.3.1.4 The SqlNull Theory Encoded in JML

Listing 5.2 lists the JML model class `SqlNull` which defines the mathematical semantics (meaning) of SQL version 3's notion of Null. A JML static model field is defined for each of the basic types of Java types representing the SQL null value for that type. For example, NullChar is defined as the static model field to represent a null *character* in a database. For the boolean type, the rules governing SQL three-valued logic for each of the logical operators (AND, OR, NOT and EQUALS) are defined as JML axioms. JML axioms are predicates (boolean functions) that tools that reason about JML, like theorem provers, takes as a given axiomatic truth. For each of the other primitive Java types, only the three-valued logic semantics of value equality ("==") is required, as the other logical operators are not used by the other primitive types.

5.3.2 Tracking Object Persistence

To correctly reason about SQL Null we also need to take into account that it is not required to check whether the value is null or not until the object is persisted to the database. This is due to the database only checking whether it is required that the object be null or not when the database operation is performed. To achieve this a JML model field needs to be added to any persisted class to track whether the object has been persisted or not. Listing 5.3 is an example of a JML model field of type boolean called `persisted` which, if true, means the object has been persisted, or, if false, means the object has not yet been persisted.

5.3.3 Using Persistence Models in Invariants

Listing 5.4 is an example of how JML invariants can be used to specify that only after the *AuditLog* entity has been persisted, should the *auditLogId* attribute be checked to ensure it is not SQL null. The JML model field `persisted` will only be set to true after the object has been persisted. The *auditLogId* is of particular interest as it value is assigned by the database itself at time of insert, its value is therefore never known till the object has been persisted.

5.3.3.1 Reasoning in Data Access Objects

To ensure the JML model field `persisted` is set correctly, the data access object methods that call the JPA interface methods to persist or reads from the database must set and check the persisted field correctly. Listing 5.5 is an example of how the persisted field could be checked and set using JML in the *AuditDAOBean* class. The *auditEvent()* method is called to persist a *AuditLog* entity to the database. The *getAuditLog()* method is called to get a

Listing 5.2: JML Model Class SqlNull

```
/**
 * This JML model class defines the mathematical semantics (meaning) of
 * SQLv3's notion of NULL (aka UNKNOWN, or small omega).  The semantics
 * of UNKNOWN happen to be exactly the semantics of Kleene's three-valued
 * logic.
 *
 * @author Joe Kiniry
 * @author Alan Barrett
*/
/*@ public model class SqlNull {
        public static model boolean NullBoolean;
        public static model boolean UNKNOWN = NullBoolean;
        axiom UNKNOWN == NullBoolean;
        public static model char NullChar;
        public static model byte NullByte;
        public static model short NullShort;
        public static model int NullInt;
        public static model long NullLong;
        public static model float NullFloat;
        public static model double NullDouble;
        public static model Object nullObject;

        // Definition of semantics for booleans.  These are just
        // axiomatic encodings of the truth tables of Kleene's
        // 3-valued logic.

        // define the semantics of logical AND with respect to the
        // UNKNOWN (NULL) value
        axiom true & UNKNOWN <==> UNKNOWN;
        axiom false & UNKNOWN <==> false;
        axiom UNKNOWN & UNKNOWN <==> UNKNOWN;
        // define the semantics of logical OR ...
        axiom true | UNKNOWN <==> true;
        axiom false | UNKNOWN <==> UNKNOWN;
        axiom UNKNOWN | UNKNOWN <==> UNKNOWN;
        // define the semantics of logical NOT ...
        axiom ! UNKNOWN <==> UNKNOWN;
        // define the semantics of logical == ...
        axiom true == UNKNOWN <==> UNKNOWN;
        axiom false == UNKNOWN <==> UNKNOWN;
        axiom UNKNOWN == UNKNOWN <==> UNKNOWN;

        // Definition of semantics of == for other types
        axiom (NullChar == NullChar) <==> UNKNOWN;
        axiom (NullByte == NullByte) <==> UNKNOWN;
        axiom (NullShort == NullShort) <==> UNKNOWN;
        axiom (NullInt == NullInt) <==> UNKNOWN;
        axiom (NullLong == NullLong) <==> UNKNOWN;
        axiom (NullFloat == NullFloat) <==> UNKNOWN;
        axiom (NullDouble == NullDouble) <==> UNKNOWN;
        axiom (nullObject == nullObject) <==> UNKNOWN;
}
 */
```

Listing 5.3: JML Model boolean field persisted

```
public class AuditLog implements Serializable
{
        // The JML model field persisted represents whether the AuditLog
        // object has been persisted or not.
        //@ public model boolean persisted;
...
}
```

Listing 5.4: JML Invariant to test SQL nullability

```
public class AuditLog implements Serializable
{
        @Id
        //value generated by database when row inserted on table.
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        @Column(name = "AUDIT_LOG_ID", nullable = false)
        //@ invariant 0 < auditLogId;
        // Following JML invariant checks only after object has been
        // persisted, that the auditLogId is not a SQL null.
        //@ invariant persisted ==> (auditLogId != SqlNull.NullInt);
        private int auditLogId;
...
}
```

AuditLog entity from the database. The JML `ensures` keyword indicates that following predicate is a postcondition that must be satisfied when the method ends.

5.3.4 Leveraging Static Checking in DAOs

By running Listing 5.5 with a static checker such as ESC/Java2, runtime exceptions on the database are minimised as any scenarios where "Null" is assigned to a non-null column on a database are caught by the static checker.

5.3.5 Annotation and JML Generation

This subsection describes how a tool could be developed to automatically generate JML invariants for database nullness and column length from JPA annotations, and how JPA annotations can potentially be generated from JML invariants.

5.3.5.1 Generation of JML from JPA Annotations

As the JPA annotation `@Column` attributes of nullable and length can be translated into JML invariants a tool could be created which could generate automatically the JML invariants to check String length and SQL nullness.

Listing 5.5: JML persistent Field Setting in DAO

```
public class AuditDAOBean implements AuditDAO, AuditDAORemote{

        //@ ensures auditLog.persisted;
        public void auditEvent(String userId, String screenCode,
            char eventCode, GregorianCalendar gc,
            String auditMessage) throws Exception {
          AuditLog auditLog = new AuditLog(0, userId, screenCode,
            eventCode, gc, auditMessage);
          try {
            entityManager.persist(auditLog);
          }
...
        //@ ensures \result.persisted;
        public AuditLog getAuditLog(int auditLogId) throws Exception {
          AuditLog auditLog = null;
          try {
            auditLog = (AuditLog) entityManager.find(AuditLog.class,
               auditLogId);
          }
          return user;
        }
...
}
```

5.3.5.2 Generation of JPA Annotations from JML

The inverse is also true. If we know the entity is to be persisted as the class has been marked with the JPA annotation @Entity a tool could generate the JPA `@Column` annotation automatically specifying the length and null attributes of any of the attributes in the class by examining the JML invariants specified for the attributes.

5.3.5.3 Avoiding JML Generation

By enhancing JML models to understand JPA annotations, there is the possibility of avoiding having to translate JPA annotations to JML invariants and model fields. If JML models of the JPA annotations can be created, then static checking tools which understand JML can automatically check JPA persisted applications by utilising the persistent information stored in the JPA annotations. This would be particulary useful for developers who are not familiar with JML, as they could statically check their applications without having to add any JML specifications to the application.

5.3.6 Tools for Reasoning about Persistence

There are a number of tools that can be used to reason about Java persistence. This subsection will discuss the ESC/Java2 static checking tool.

5.3.6.1 ESC/Java2

The Extended Static Checker for Java version 2 (ESC/Java2) is an extended static checking tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal JML annotations [13]. As ESC/Java2 does not contain a three-valued model checker, the JML model class such as SqlNull in Listing 5.2 which defines the mathematical semantics (meaning) of SQL version 3's notion of NULL is required for the tool to be able to successfully reason about SQL nullness in Java programs.

5.4 Evaluation

Using JML to check the maximum length of strings is simple to achieve by adding a JML invariant. This simple addition of an invariant opens up the possibility via use of existing JML tools, to statically test, generates unit test cases, and verifies at runtime that the predicate is always true.

Using JML to statically reason about database nullness is not so simple to achieve. We have defined database nullness via a new JML model class "SqlNull", and also demonstrated how a JML model field "persisted" can be used to track whether an object has been persisted or not. Using the model class "SqlNull" and field "persisted", JML specifications can be added to any Java persistence program to reason about the nullness of entity attributes to be persisted.

Chapter 6

Conclusion

6.1 Conclusion

The case study demonstrated that JPA is an excellent standard persistence framework. A number of issues identified in the case study are scheduled to be resolved in version 2.0 of JPA which will further solidify the Java persistence standard. The optional use of JPA annotations was found to be much more intuitive than maintaining separate XML files with the object relational mappings for the persisted entities.

Having a standard persistence framework has enabled persistence tools to be developed that can greatly assist in the development and support of JPA applications.

This dissertation has taken the first steps to enabling reasoning about persistence applications, with particular emphasis on reasoning about database nullness and database column lengths for strings.

6.2 Further Work

The case study did not perform any performance testing of the converted RAP application. Further work should be done to measure any performance improvements or reductions due to the use of JPA.

Further work is needed to develop and test the schema comparison tool.

Merging the worlds of Java persistence and formal methods has the potential to deliver high quality tools to reason and verify Java persistence applications. There is though much more research and development to be done before the goal of reasoning about persistence can be fully achieved.

The next step would be to test the JML specification for verifying string column lengths against a number of classes with String attributes. The following step would be to utilise

the case study application to test the JML domain model class SqlNull with a number of entities to test database nullness.

Once the extended static checking tool ESC/Java2 supports Java version 1.5 the tool can be used to statically check the JML specifications for database nullness against the implemented case study application.

Bibliography

[1] The java modeling language (jml) home page, 2008. http://www.eecs.ucf.edu/~leavens/JML/.

[2] Marieke Huisman Bart Jacobs, Joachim van den Berg and Martijn van Berkum. Reasoning about java classes (preliminary report). *OOPSLA 1998, ACM*, 1998.

[3] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications, 2007.

[4] Grady Booch. *Object-oriented design with applications*. Addison-Wesley, 2001.

[5] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005.

[6] Dermot Cochran. Secure internet voting in ireland using the open source kiezen op afstand (koa) remote voting system. Master's thesis, School of Computer Science and Informatics, University College Dublin, 2006.

[7] Apache Software Foundation. Apache struts, 2008. http://struts.apache.org/.

[8] Genuitec. Myeclipse home page, 2008. http://www.myeclipseide.com/.

[9] JBoss. The hibernate persistence framework home page, 2008. http://www.hibernate.org/.

[10] JBoss. Hibernate validator home page, 2008. http://www.hibernate.org/412.html.

[11] Alan E. Morkan Joseph R. Kiniry and Barry Denby. Soundness and completeness warnings in esc/java2. *Fifth International Workshop on Specification and Verification of Component-Based Systems*, 2006.

[12] Mike Keith and Merrick Schincariol. *Pro EJB 3 Java Persistence API*. Apress, 2006.

[13] Joe Kiniry. Esc/java2, 2007. http://kind.ucd.ie/products/opensource/ESCJava2/.

[14] Joseph R. Kiniry and Daniel M. Zimmerman. Verification-centric software development with jml and esc/java2. *ETAPS'08*, 2008.

[15] Stephen Kleene. *Introduction to Metamathematics*. North-Holland, 1987.

[16] Gary T. Leavens and Yoonsik Cheon. Design by contract with jml, 2006. http://www.jmlspecs.org/jmldbc.pdf.

[17] Gary T. Leavens, K. Rustan, M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. Jml: notations and tools supporting detailed design in java. *OOPSLA '00 Companion, Minneapolis*, 2000.

[18] Avraham Leff and James T. Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International*, pages 118 – 127, 2001.

[19] Sun Microsystems. Core j2ee patterns - data access object, 2001. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html.

[20] Sun Microsystems. Jsr-75 a metadata facility for the java programming language, 2004. http://jcp.org/aboutJava/communityprocess/final/jsr175/index.html.

[21] Sun Microsystems. Jsr-220: Enterprise javabeans, version 3.0 java persistence api, 2006. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[22] Sun Microsystems. The java persistence api (jpa) home page, 2008. http://java.sun.com/javaee/technologies/persistence.jsp.

[23] Sun Microsystems. Javaserver pages technology, 2008. http://java.sun.com/products/jsp/.

[24] Sun Microsystems. Jsr-303: Bean validation, 2008. http://jcp.org/en/jsr/detail?id=303.

[25] Sun Microsystems. Jsr-317: Java persistence 2.0, 2008. http://jcp.org/en/jsr/detail?id=317.

[26] Sub project of the Web Tools Platform Project. Dali jpa tools, 2008. http://www.eclipse.org/webtools/dali/main.php.

Appendix A

Hibernate XML mapping files

This appendix presents the Hibernate XML configuration files used to map the AuditLog and Plant objects to the tables in the original case study application.

Listing A.1: Audit_Log.hbm.xml

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

<hibernate-mapping>
  <class name="com.dbaccess.bean.AuditLog" table="AUDIT_LOG" lazy="false">
    <id name="auditLogId" column="AUDIT_LOG_ID" type="int">
      generator class="sequence"/>
    </id>
    <property name="userId" column="USER_ID" type="string" unique="false"/>
    <property name="screenCode" column="SCREEN_CODE" type="string" unique="false"/>
    <property name="screenActionCode" column="SCREEN_ACTION_CODE" type="char" unique="false"/>
    <property name="auditTimestamp" column="DATETIME" type="calendar" unique="false"/>
    <property name="auditInfo" column="AUDIT_INFO" type="string" unique="false"/>
  </class>
</hibernate-mapping>
```

Listing A.2: Plant.hbm.xml

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

<hibernate-mapping>
  <class name="com.dbaccess.bean.Plant" table="PLANT">
    <id name="plantId" column="PLANT_CODE" type="string" unsaved-value="">
      <generator class="assigned"/>
    </id>
    <property name="plantName"  column="PLANT_NAME" type="string" unique="false"/>
    <property name="loginDuration" column="LOGIN_DURATION" type="int" unique="false"/>
    <property name="loginMessage" column="LOGIN_MESSAGE" type="string" unique="false"/>
    <property name="messageExpiryDate" column="MESSAGE_EXPIRY_DATE" type="calendar" unique="false"/>
    <property name="defaultLanguage" column="DEFAULT_LANGUAGE_CD" type="string" unique="false"/>
    <property name="plantOfficePrinterId" column="PLANT_OFFICE_PRINTER_ID" type="string" unique="false"/>
    <property name="damagePrinterId" column="DAMAGE_PRINTER_ID" type="string" unique="false"/>
    <property name="opticalArchiveFlag" column="OPTICAL_ARCHIVE_FLAG" type="char" unique="false"/>
    <property name="udrAutomaticPrintFlag" column="UDR_AUTOMATIC_PRINT_FLAG" type="char" unique="false"/>
    <property name="udrAutomaticEmailFlag" column="UDR_AUTOMATIC_EMAIL_FLAG" type="char" unique="false"/>
    <bag name="plantDocks" table="PLANT_DOCK" lazy="false" inverse="true" cascade="all,delete-orphan">
      <key><column name="PLANT_CODE"/></key>
      <one-to-many class="com.dbaccess.bean.PlantDock"/>
```

```
    </bag>
    <bag name="udrMailAddresses" table="UDR_EMAIL_ADDRESS"
     lazy="false" inverse="true" cascade="all,delete-orphan">
      <key><column name="PLANT_CODE"/></key>
      <one-to-many class="com.dbaccess.bean.UdrMailAddress"/>
    </bag>
    <bag name="tdrMailAddresses" table="DAMAGE_REPORT_EMAIL_ADDRESS"
     lazy="false" inverse="true" cascade="all,delete-orphan">
      <key><column name="PLANT_CODE"/></key>
      <one-to-many class="com.dbaccess.bean.TdrMailAddress"/>
    </bag>
  </class>

  <class name="com.dbaccess.bean.PlantDock" table="PLANT_DOCK">
    <composite-id name="pdid" class="com.dbaccess.bean.PlantDockId">
      <key-property name="plantId" type="string" column="PLANT_CODE"/>
      <key-property name="dockCd" type="string" column="DOCK_CD"/>
    </composite-id>
    <property name="dockPrinterId" column="DOCK_PRINTER_ID" type="string"/>
  </class>

  <class name="com.dbaccess.bean.UdrMailAddress" table="UDR_EMAIL_ADDRESS">
    <composite-id name="emid" class="com.dbaccess.bean.ReportEmailId">
      <key-property name="plantId" type="string" column="PLANT_CODE"/>
      <key-property name="emailAddress" type="string" column="EMAIL_ADDRESS_TXT"/>
    </composite-id>
    <property name="followUpCode" type="string" column="FOLLOW_UP_CD"/>
  </class>

  <class name="com.dbaccess.bean.TdrMailAddress" table="DAMAGE_RPT_EMAIL_ADDRESS">
    <composite-id name="dmid" class="com.dbaccess.bean.ReportEmailId">
      <key-property name="plantId" type="string" column="PLANT_CODE"/>
      <key-property name="emailAddress" type="string" column="EMAIL_ADDRESS_TXT"/>
    </composite-id>
  </class>
</hibernate-mapping>
```

Appendix B

RAP JPA Annotated JavaBeans

This appendix presents the AuditLog and Plant JavaBeans which have been annotated with the standard JPA annotation to map the objects to the tables in the original case study application. The JPA annotations have replaced the Hibernate XML configuration files in the original application.

Listing B.1: AuditLog.java

```
package com.alanb.rap.dbaccess.entity;

import java.io.Serializable;
import java.util.GregorianCalendar;
import javax.persistence.*;

/**
 * Entity Bean containing representing an Audit object.
 * @author Alan Barrett
 *
 */

@Entity @Table(name = "AUDIT_LOG")
public class AuditLog implements Serializable
{
        @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
        @Column(name = "AUDIT_LOG_ID", nullable = false)
        private int auditLogId;

        @Column(name = "USER_ID", length = 8, nullable = false)
        private String userId;
        @Column(name = "SCREEN_CODE", length = 20, nullable = false)
        private String screenCode;
        @Column(name = "SCREEN_ACTION_CODE", length = 1, nullable = false)
        private char screenActionCode;
        @Column(name = "DATETIME")
        @Temporal(TemporalType.TIMESTAMP)
        private GregorianCalendar auditTimestamp;
        @Column(name = "AUDIT_INFO", length = 80, nullable = false)
        private String auditInfo;

        public AuditLog() {
                super();
        }

        public AuditLog(  int auditLogId, String userId, String screenCode,
                char screenActionCode, GregorianCalendar auditTimestamp, String auditInfo) {
                super();
                this.setAuditLogId(auditLogId);
```

```
                this.setUserId(userId);
                this.setScreenCode(screenCode);
                this.setScreenActionCode(screenActionCode);
                this.setAuditTimestamp(auditTimestamp);
                this.setAuditInfo(auditInfo);
        }

        public int getAuditLogId() {
                eturn auditLogId;
        }
        public void setAuditLogId(int auditLogId) {
                this.auditLogId = auditLogId;
        }
        public GregorianCalendar getAuditTimestamp() {
                return auditTimestamp;
        }
        public void setAuditTimestamp(GregorianCalendar auditTimestamp) {
                this.auditTimestamp = auditTimestamp;
        }
        public char getScreenActionCode() {
                return screenActionCode;
        }
        public void setScreenActionCode(char screenActionCode) {
                this.screenActionCode = screenActionCode;
        }
        public String getScreenCode() {
                return screenCode;
        }
        public void setScreenCode(String screenCode) {
                this.screenCode = screenCode;
        }
        public String getUserId() {
                return userId;
        }
        public void setUserId(String userId) {
                this.userId = userId;
        }
        public String getAuditInfo() {
                return auditInfo;
        }
        public void setAuditInfo(String auditInfo) {
                this.auditInfo = auditInfo;
        }
}
```

Listing B.2: Plant.java

```
package com.alanb.rap.dbaccess.entity;

import java.io.Serializable;
import java.util.List;
import java.util.Calendar;
import javax.persistence.*;

/**
 * Entity Bean representing a Plant object.
 *
 * @author Alan Barrett
 * @review Joe Kiniry
 *
 */

@Entity @Table(name = "PLANT")

@NamedQuery(name = "Plant.getAllPlants",
        query = "Select␣p␣" + "FROM␣Plant␣p␣" + "ORDER␣BY␣p.plantId")

public class Plant implements Serializable {

        @Id
        @Column(name = "PLANT_CODE", length = 2, nullable = false)
```

```
        private String plantId;
        @Column(name = "PLANT_NAME", length = 20, nullable = false)
        private String plantName;
        @Column(name = "LOGIN_DURATION", nullable = false)
        private int loginDuration;
        @Column(name = "LOGIN_MESSAGE")
        private String loginMessage;
        @Column(name = "MESSAGE_EXPIRY_DATE")
        @Temporal(TemporalType.DATE)
        private Calendar messageExpiryDate;
        @Column(name = "DEFAULT_LANGUAGE_CD", length = 2, nullable = false)
        private String defaultLanguage;
        @Column(name = "PLANT_OFFICE_PRINTER_ID", length = 50, nullable = false)
        private String plantOfficePrinterId;
        @Column(name = "DAMAGE_PRINTER_ID", length = 50)
        private String damagePrinterId;
        @Column(name = "OPTICAL_ARCHIVE_FLAG", length = 1, nullable = false)
        private char opticalArchiveFlag;
        @Column(name = "UDR_AUTOMATIC_PRINT_FLAG", length = 1, nullable = false)
        private char udrAutomaticPrintFlag;
        @Column(name = "UDR_AUTOMATIC_EMAIL_FLAG", length = 1, nullable = false)
        private char udrAutomaticEmailFlag;

        // Need fetch type eager to ensure no lazy loading of sub objects.
        // Need to cascade all to ensure plant_dock entry deleted when plant deleted,
        // and also that when plant is created/updated any changes are cascaded.

        @OneToMany(fetch=FetchType.EAGER, mappedBy="plant", cascade=CascadeType.ALL)
        private List<PlantDock> plantDocks;

        @OneToMany(fetch=FetchType.EAGER, mappedBy="plant", cascade=CascadeType.ALL)
        private List<UdrMailAddress> udrMailAddresses;

        @OneToMany(fetch=FetchType.EAGER, mappedBy="plant", cascade=CascadeType.ALL)
        private List<TdrMailAddress> tdrMailAddresses;

        @ManyToMany(fetch=FetchType.EAGER, mappedBy="plants", cascade=CascadeType.ALL)
        private List<User> users;

        public Plant() {
        }

        public Plant( String plantId,String plantName,int loginDuration,String loginMessage,
                Calendar messageExpiryDate,     String defaultLanguage,String plantOfficePrinterId,
                String damagePrinterId,String tdrEmailAddress,char opticalArchiveFlag,
                char udrAutomaticPrintFlag,char udrAutomaticEmailFlag, List <PlantDock> plantDocks,
                List <UdrMailAddress> udrEMailAddresses, List <TdrMailAddress> tdrMailAddresses,
                List <User> users) {
                this.setPlantId(plantId);
                this.setPlantName(plantName);
                this.setLoginDuration(loginDuration);
                this.setLoginMessage(loginMessage);
                this.setMessageExpiryDate(messageExpiryDate);
                this.setDefaultLanguage(defaultLanguage);
                this.setPlantOfficePrinterId(plantOfficePrinterId);
                this.setDamagePrinterId(damagePrinterId);
                this.setOpticalArchiveFlag(opticalArchiveFlag);
                this.setUdrAutomaticPrintFlag(udrAutomaticPrintFlag);
                this.setUdrAutomaticEmailFlag(udrAutomaticEmailFlag);
                this.setPlantDocks(plantDocks);
                this.setUdrMailAddresses(udrEMailAddresses);
                this.setTdrMailAddresses(tdrMailAddresses);
                this.setUsers(users);
        }

        public String getDefaultLanguage() {
          return defaultLanguage;
        }
        public void setDefaultLanguage(String defaultLanguage) {
                this.defaultLanguage = defaultLanguage;
}
        public int getLoginDuration() {
```

```
        return loginDuration;
    }
    public void setLoginDuration(int loginDuration) {
        this.loginDuration = loginDuration;
    }
    public String getLoginMessage() {
        return loginMessage;
    }
    public void setLoginMessage(String loginMessage) {
        this.loginMessage = loginMessage;
    }
    public Calendar getMessageExpiryDate() {
        return messageExpiryDate;
    }
    public void setMessageExpiryDate(Calendar messageExpiryDate) {
        this.messageExpiryDate = messageExpiryDate;
    }
    public char getOpticalArchiveFlag() {
        return opticalArchiveFlag;
    }
    public void setOpticalArchiveFlag(char opticalArchiveFlag) {
        this.opticalArchiveFlag = opticalArchiveFlag;
    }
    public List <PlantDock> getPlantDocks() {
        return plantDocks;
    }
    public void setPlantDocks(List <PlantDock> plantDocks) {
        this.plantDocks = plantDocks;
    }
    public String getPlantId() {
        return plantId;
    }
    public void setPlantId(String plantId) {
        this.plantId = plantId;
    }
    public String getPlantName() {
        return plantName;
    }
    public void setPlantName(String plantName) {
        this.plantName = plantName;
    }
    public String getPlantOfficePrinterId() {
        return plantOfficePrinterId;
    }
    public void setPlantOfficePrinterId(String plantOfficePrinterId) {
        this.plantOfficePrinterId = plantOfficePrinterId;
    }
    public char getUdrAutomaticEmailFlag() {
        return udrAutomaticEmailFlag;
    }
    public void setUdrAutomaticEmailFlag(char udrAutomaticEmailFlag) {
        this.udrAutomaticEmailFlag = udrAutomaticEmailFlag;
    }
    public char getUdrAutomaticPrintFlag() {
        return udrAutomaticPrintFlag;
    }
    public void setUdrAutomaticPrintFlag(char udrAutomaticPrintFlag) {
        this.udrAutomaticPrintFlag = udrAutomaticPrintFlag;
    }
    public List <UdrMailAddress> getUdrMailAddresses() {
        return udrMailAddresses;
    }
    public void setUdrMailAddresses(List <UdrMailAddress> udrEMailAddresses) {
        this.udrMailAddresses = udrEMailAddresses;
    }
    public String getDamagePrinterId() {
        return damagePrinterId;
    }
    public void setDamagePrinterId(String damagePrinterId) {
        this.damagePrinterId = damagePrinterId;
    }
    public List <TdrMailAddress> getTdrMailAddresses() {
```

```
                return tdrMailAddresses;
        }
        public void setTdrMailAddresses(List <TdrMailAddress> tdrMailAddresses) {
                this.tdrMailAddresses = tdrMailAddresses;
        }
        public List <User> getUsers() {
                return users;
        }
        public void setUsers(List<User> users) {
                this.users = users;
        }
        public void addUser(User user) {
                this.getUsers().add(user);
        }
```

Listing B.3: PlantDock.java

```
package com.alanb.rap.dbaccess.entity;
import java.io.Serializable;
import javax.persistence.*;

/**
 * Entity Bean representing a Plant dock object.
 *
 * @author Alan Barrett
 *
 */

@Entity @Table(name = "PLANT_DOCK")
@IdClass(PlantDockId.class)
public class PlantDock implements Serializable {

        // insertable, updateable = false needed to ensure mapping is ignored.
        @Id @Column(name="PLANT_CODE", insertable=false, updatable=false, length = 2)
        private String plantId;
        @Id @Column(name="DOCK_CD", length = 8, nullable = false)
        private String dockCd;

        @Column(name = "DOCK_PRINTER_ID", length = 50, nullable = false)
        private String dockPrinterId;

        // needed to resolve duplicate insert using key columns, overrides earlier mapping for PLANT_CODE
        @ManyToOne
        @JoinColumn(name="PLANT_CODE")
        private Plant plant;

        public PlantDock() {
        }

        public PlantDock( String plantId,String dockCd,String dockPrinterId, Plant plant) {
                this.setPlantId(plantId);
                this.setDockCd(dockCd);
                this.setDockPrinterId(dockPrinterId);
                this.plant = plant;
        }

        public String getDockPrinterId() {
                return dockPrinterId;
        }
        public void setDockPrinterId(String dockPrinterId) {
                this.dockPrinterId = dockPrinterId;
        }
        public String getDockCd() {
                return dockCd;
        }
        public void setDockCd(String dockCd) {
                String padOut = "        "; /* 8 characters */
                if (dockCd.length() < 8) {
                        this.dockCd = dockCd + padOut.substring((dockCd.length()));
                }
                else, this.dockCd = dockCd;
```

```
        }
        public String getPlantId() {
                return plantId;
        }
        public void setPlantId(String plantId) {
                this.plantId = plantId;
        }
}
```

Listing B.4: PlantDockId.java

```
package com.alanb.rap.dbaccess.entity;
import java.io.Serializable;

/**
 * JavaBean representing a PlantDock ID object. This additional bean
 * is required for JPA as it needs a unique single object for a key, and
 * cannot cope with multiple key fields. This requirement is how it handles
 * multiple key fields. The JPA recommendation is for surrogate key fields,
 * which RAP uses where it can.
 *
 * @author Alan Barrett
 *
 */

public class PlantDockId implements Serializable{
        private String plantId;
        private String dockCd;

        public PlantDockId() {
        }

        public PlantDockId(String plantId, String dockCd) {
                this.setPlantId(plantId);
                this.setDockCd(dockCd);
        }

        public boolean equals(Object o) {
        return ((o instanceof PlantDockId) &&
                this.getPlantId().equals(((PlantDockId) o).getPlantId()) &&
                this.getDockCd().equals(((PlantDockId) o).getDockCd()));
        }

        public int hashCode() {
                return (this.getPlantId() + this.getDockCd()).hashCode();
        }

        public String getDockCd() {
                return dockCd;
        }
        public void setDockCd(String dockCd) {
                String padOut = "␣␣␣␣␣␣␣␣"; /* 8 characters */
                if (dockCd.length() < 8) {
                        this.dockCd = dockCd + padOut.substring((dockCd.length()));
                }
                else this.dockCd = dockCd;
        }
        public String getPlantId() {
                return plantId;
        }
        public void setPlantId(String plantId) {
                this.plantId = plantId;
        }
}
```

Appendix C

RAP JPA Data Access Objects

This appendix lists the Audit and Plant Data Access Objects (DAO) implemented in the refactored RAP application.

Listing C.1: AuditDAOBean.java

```
package com.alanb.rap.dbaccess.dao;
import java.util.GregorianCalendar;
import javax.persistence.*;
import javax.ejb.*;
import com.alanb.rap.dbaccess.entity.AuditLog;
import com.alanb.rap.exception.RAPDAOException;

/**
 * AuditData Access Object Class. Represents all the data access methods required for interaction between the
 * Audit Object and the associated database table.
 *
 * @author Alan Barrett
 *
 */
@Stateless
public class AuditDAOBean implements AuditDAO, AuditDAORemote{

        public AuditDAOBean() {
        }

        @PersistenceContext(unitName="jpa_rap2")
        private EntityManager em;
        /**
         * Add an entry to the Audit table. Catch and log any failures.
         *
         * @param userId userId
         * @param screenCode Code of the screen where the audit event took place
         * @param eventCode the audit event
         * @param gc current time
         * @param auditMessage the audit message for the audit table
         *
         */
        public void auditEvent( String userId, String screenCode, char eventCode,
          GregorianCalendar gc, String auditMessage) throws Exception {
                AuditLog al = new AuditLog(     0,userId,screenCode,eventCode,gc,auditMessage);
                try {
                  em.persist(al);
                }
                catch (PersistenceException pe) {
                  throw new RAPDAOException(pe, "error.Log.AuditError", screenCode);
                }
                catch (Exception e) {
                  throw e;
```

```
                }
                finally {
                }
        }
}
```

Listing C.2: PlantDAOBean.java

```
package com.alanb.rap.dbaccess.dao;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Vector;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceException;
import com.alanb.rap.dbaccess.entity.Plant;
import com.alanb.rap.dbaccess.entity.PlantDock;
import com.alanb.rap.dbaccess.entity.ReportEmailId;
import com.alanb.rap.dbaccess.entity.TdrMailAddress;
import com.alanb.rap.dbaccess.entity.UdrMailAddress;
import com.alanb.rap.dbaccess.entity.User;
import com.alanb.rap.exception.RAPDAOException;
/**
 * Plant Access Object Class. Represents all the data access methods required for interaction between the
 * Plant object and the associated database table.
 *
 * @author Alan Barrett
 *
 */

@Stateless
public class PlantDAOBean implements PlantDAO, PlantDAORemote {
        public PlantDAOBean() {
        }
        @PersistenceContext(unitName="jpa_rap2")
        private EntityManager em;
        @EJB
        UserDAO userDAOBean;
        /**
         * Add an entry to the Plant table. Catch and log any failures.
         *
         * @param newPlant new Plant object to be added
         *
         */
        public void addPlant(Plant newPlant) throws Exception {
                try {
                  em.persist(newPlant);
                  em.flush(); }
                catch (PersistenceException pe) {
                  throw new RAPDAOException(pe, "error.Log.PlantDAOaddPlant", newPlant.getPlantId()); }
                catch (Exception e){
                  throw e; }
                finally { }
        }

        /**
         * get a Plant object for the passed in plantID. Catch and log any failures.
         *
         * @param plantId : ID of plant to be retrieved.
         *
         */
        public Plant getPlant(String plantId) throws Exception {
                Plant thisPlant = null;
                try {
                  thisPlant = (Plant) em.find(Plant.class, plantId); }
                catch (PersistenceException pe) {
                  throw new RAPDAOException(pe, "error.Log.PlantDAOgetPlant", plantId); }
                catch (Exception e) {
```

```
            throw e; }
        finally {}
        return thisPlant;
    }

    /**
     * delete a Plant object for the passed in plantID. Catch and log any failures.
     *
     * @param plantId : ID of plant to be retrieved.
     *
     */
    public void deletePlant(String plantId) throws Exception {
        Plant thisPlant = null;
        try {
          thisPlant = em.find(Plant.class, plantId);
          em.remove(thisPlant);
          em.flush(); }
        catch (PersistenceException pe) {
          throw new RAPDAOException(pe, "error.Log.PlantDAOdeletePlant", plantId); }
        catch (Exception e) {
          throw e; }
        finally { }
    }

    /**
     * Return a complete list of all Plants. Catch and log any failures.
     *
     */
    public Vector<Plant> getAllPlants() throws Exception {
        Vector<Plant> allPlantsVector = new Vector<Plant>();
        try {
          List<Plant> listOfUserPlants = em.createNamedQuery("Plant.getAllPlants").getResultList();
          allPlantsVector = (Vector<Plant>) listOfUserPlants; }
        catch (PersistenceException pe) {
          throw new RAPDAOException(pe, "error.Log.PlantDAOgetAllPlants"); }
        catch (Exception e) {
          throw e; }
        finally { }
        return allPlantsVector;
    }

    /**
     * Delete a plantDock for the passed in plant and Dock. Catch and log any failures.
     *
     * @param plant plantCode
     * @param dock DockCode
     */
    public void deletePlantDock(Plant plant, String dock) throws Exception {
        PlantDock plantDock = null;
        try {
          Plant sessionPlant = (Plant) em.find(Plant.class, plant.getPlantId());
          Iterator<PlantDock> it = (Iterator<PlantDock>) sessionPlant.getPlantDocks().iterator();
          while (it.hasNext()) {
            plantDock = (PlantDock) it.next();
            if ( dock.equalsIgnoreCase(plantDock.getDockCd())) {
              em.remove(plantDock);
              it.remove();
            }
          }
          em.merge(sessionPlant);
          em.flush();
        }
        catch (PersistenceException pe) {
          throw new RAPDAOException(pe, "error.Log.PlantDAOdeletePlantDock", dock); }
        catch (Exception e) {
          throw e; }
        finally { }
    }

    /**
     * Add a plantDock for the passed in plant, Dock and printer. Catch and log any failures.
     *
```

```
     * @param plant plantCode
     * @param dock DockCode
     * @param printer Default plantDock Printer
     */
    public void addPlantDock(Plant plant, String dock, String printer) throws Exception {
            Plant thisPlant = plant;
            PlantDock plantDock = null;
            boolean dockAlreadyExists = false;
            try {
              Plant sessionPlant = (Plant) em.find(Plant.class, thisPlant.getPlantId());
              // check if the Dock already exists for the Plant
              List <PlantDock> aList = sessionPlant.getPlantDocks();
              Iterator <PlantDock> itPlantDocks =
                (Iterator <PlantDock>) sessionPlant.getPlantDocks().iterator();
              while (itPlantDocks.hasNext()) {
                PlantDock plantDocks = (PlantDock) itPlantDocks.next();
                if ( plantDocks.getDockCd().equalsIgnoreCase(dock)) {
                  // Dock already exists so update the Printer code.
                  dockAlreadyExists = true;
                  plantDocks.setDockPrinterId(printer);
                }
              }
              // if the dock doesn't already exist for the plant then add it
              if (!dockAlreadyExists) {
                plantDock = new PlantDock(plant.getPlantId(), dock, printer, sessionPlant);
                aList.add(plantDock);
              }
              em.merge(sessionPlant);
              em.flush();
            }
            catch (PersistenceException pe) {
              throw new RAPDAOException(pe, "error.Log.PlantDAOaddPlantDock", dock); }
            catch (Exception e) {
              throw e; }
            finally { }
    }

    /**
     * Add a UDR email for the passed in plant, udrEmail object and followUpCode.
     * Catch and log any failures.
     *
     * @param plant plantCode
     * @param udrEmail email address object
     * @param followUpCode Default plantDock Printer
     */
    public void addUdrEmail(Plant plant, String udrEmail, String followUpCode) throws Exception {
            Plant thisPlant = plant;
            try {
              Plant sessionPlant = (Plant) em.find(Plant.class, thisPlant.getPlantId());
              List <UdrMailAddress> udrMailList
                = (List <UdrMailAddress>) sessionPlant.getUdrMailAddresses();
              UdrMailAddress uMailAddress = new UdrMailAddress(
              new ReportEmailId(thisPlant.getPlantId(), udrEmail), followUpCode, sessionPlant);
              udrMailList.add(uMailAddress);
              em.flush();
            }
            catch (PersistenceException pe) {
              throw new RAPDAOException(pe, "error.Log.PlantDAOaddUdrEmail", udrEmail); }
            catch (Exception e) {
              throw e; }
            finally { }
    }

    /**
     * Delete a UDR email for the passed in plant and udrEmail address. Catch and log any failures.
     *
     * @param plant plantCode
     * @param udrEmail Unload Deviation Email
     */
    public void deleteUdrEmail(Plant plant, String udrEmail) throws Exception {
            Plant thisPlant = plant;
            try {
```

```
          Plant sessionPlant = (Plant) em.find(Plant.class, thisPlant.getPlantId());
          Iterator <UdrMailAddress> itUdrMailAddresses =
            (Iterator <UdrMailAddress>) sessionPlant.getUdrMailAddresses().iterator();
          while (itUdrMailAddresses.hasNext()) {
            UdrMailAddress udrMail = (UdrMailAddress) itUdrMailAddresses.next();
            if ( udrEmail.equalsIgnoreCase(udrMail.getEmid().getEmailAddress())) {
              itUdrMailAddresses.remove();
              //AB added to ensure UDR email address is removed from the database.
              em.remove(udrMail);
            }
          }
          em.flush();
        }
        catch (PersistenceException pe) {
          throw new RAPDAOException(pe, "error.Log.PlantDAOdeleteUdrEmail", udrEmail); }
        catch (Exception e) {
          throw e; }
        finally { }
}

/**
 * Add a UDR email for the passed in plant and tdrEmail address. Catch and log any failures.
 *
 * @param plant Plant Object
 * @param tdrEmail Damage Report Email address
 */

public void addTdrEmail(Plant plant, String tdrEmail) throws Exception {
        Plant thisPlant = plant;
        try {
          Plant sessionPlant = (Plant) em.find(Plant.class, thisPlant.getPlantId());
          List <TdrMailAddress> tdrEmailList =
            (List <TdrMailAddress>) sessionPlant.getTdrMailAddresses();
          TdrMailAddress tMailAddress = new TdrMailAddress(
          new ReportEmailId(thisPlant.getPlantId(), tdrEmail), sessionPlant);
          tdrEmailList.add(tMailAddress);
          em.merge(sessionPlant);
          em.flush();
        }
        catch (PersistenceException pe) {
          throw new RAPDAOException(pe, "error.Log.PlantDAOaddTdrEmail", tdrEmail); }
        catch (Exception e) {
          throw e; }
        finally  { }
}

/**
 * Delete a Damage email for the passed in plant and tdrEmail address. Catch and log any failures.
 *
 * @param plant Plant Object
 * @param tdrEmail Damage Report Email address
 */

public void deleteTdrEmail(Plant plant, String tdrEmail) throws Exception {
        Plant thisPlant = plant;
        try {
          Plant sessionPlant = (Plant) em.find(Plant.class, thisPlant.getPlantId());
          Iterator <TdrMailAddress> itTdrEamilAddresses =
            (Iterator <TdrMailAddress>) sessionPlant.getTdrMailAddresses().iterator();
          while (itTdrEamilAddresses.hasNext()) {
            TdrMailAddress tdrMail = (TdrMailAddress) itTdrEamilAddresses.next();
            if ( tdrEmail.equalsIgnoreCase(tdrMail.getDmid().getEmailAddress())) {
              itTdrEamilAddresses.remove();
              //AB added to ensure UDR email address is removed from the database.
              em.remove(tdrMail);
            }
          }
          em.flush();
        }
        catch (PersistenceException pe) {
          throw new RAPDAOException(pe, "error.Log.PlantDAOdeleteTdrEmail", tdrEmail); }
        catch (Exception e) {
```

```
                throw e; }
              finally  { }
        }

        /**
         * Update the plant main fields for the passed in plant Object. Create a new plant object and retrieve
         * the existing plant fields. Then perform a Hibernate "merge" thus updating the fields. Catch and log
         * any failures.
         *
         * @param plant Plant Object
         */

        public void updatePlantMainFields(Plant plant) throws Exception {
                Plant thisPlant = plant;
                try {
                  Plant sessionPlant = (Plant) em.find(Plant.class, thisPlant.getPlantId());
                  // if the default language of the plant has changed.
                  if (!(plant.getDefaultLanguage().equalsIgnoreCase(sessionPlant.getDefaultLanguage()))) {
                    // Update all users for that plant to have the new default language.
                    Iterator <User> itUsers = (Iterator <User>) plant.getUsers().iterator();
                    while (itUsers.hasNext()) {
                      User tempUserPlant = (User) itUsers.next();
                      tempUserPlant.setDefaultLanguage(plant.getDefaultLanguage());
                    }
                  }
                  em.merge(plant);
                  em.flush();
                }
                catch (PersistenceException pe) {
                  throw new RAPDAOException(pe, "error.Log.PlantDAOupdatePlantMainFields",
                     plant.getPlantId()); }
                catch (Exception e) {
                  throw e; }
                finally { }
        }
}
```