OOD Case Study: Persistent Storage

32

The objective of this chapter is to illustrate the use of two persistent storage technologies—eXtensible Markup Language (XML) and Relational Database (Rdb)—by updating the case study designs.

32.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You understand how to develop data models, including entity relationship diagrams and fully-attributed logical data models. You also understand that translating a LDM to a physical data model needs to consider the limitations of the physical storage format.
- You understand the process of normalization and have applied normalization forms to a data design.
- You understand the following regarding eXtensible Markup Language (XML):
  - XML files represent data using a tree structure of tags. An XML tag will contain a data value and/or other tags, is given a name delimited with < and > (e..g, <lastname>), and typically has a matching end tag (e.g., </lastname>).
  - A DOM (document object model) represents an XML file as a tree structure of nodes. You navigate to any node in a DOM by starting at the root of the tree.
  - Both Java and Python provide support for using XML files and the DOM. First, an XML file is parsed to produce a DOM. The DOM contains Element nodes (one per tag) and Text nodes (one per data value).

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_32

- You understand the following regarding relational databases (Rdb):
  - A Rdb is a collection of one or more tables used to store data. Each table has rows representing data instances (e.g., a student) and columns representing data values (e.g., student has name Ahmad and an id of 123456). Relational databases are based on predicate logic and set theory.
  - SQL (structured query language) provides capabilities to create and manipulate databases. DDL (data definition language) statements are used to create a database, tables, indexes, and other types of database objects. DML (data manipulation language) statements are used to create, read, update, and delete data in tables.
  - Both Java and Python provide support for using Rdb. First, you connect to a relational database server and a specific database. You then execute SQL statements to manipulate the data in the database.

- You understand Model–View–Controller as a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.

32.2 OOD: ABA Persistent Storage Designs

The list of requirements found below has been modified from what was stated in Chap. 18. One requirement was changed to make the data modeling a little more interesting.

1. Allow for entry and (non-persistent) storage of people's names.
2. Store for each person a single phone number and a single email address. *Include the type of phone number and the type of email address for this person. The type of phone number may be either: home, cell, work, or other. The type of email address may be either: personal, work or other.*
3. Use a simple text-based user interface to obtain the contact data.

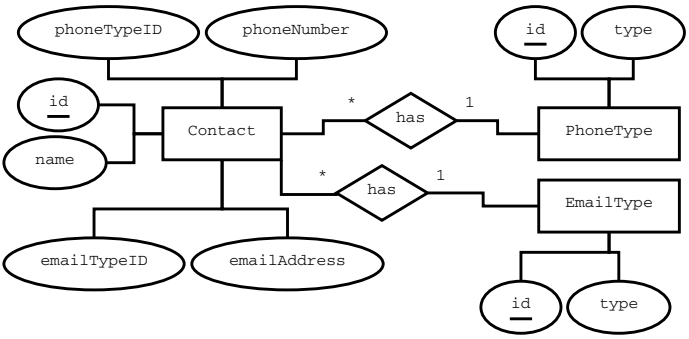

Fig. 32.1 ABA logical data model

4. Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Allow for display of contact data in the address book.

32.2.1 OOD: ABA Data Models

Figure 32.1 shows the logical data model based on the requirements listed above. Since each contact has one phone and one email, the Contact entity has attributes for phone number and email address. This includes identifying the type of phone number (e.g., cell, work) and type of email address (e.g., personal, work). The purpose of the PhoneType and EmailType entities is to keep track of each type of phone and email currently used by the ABA.

32.2.1.1 OOD: ABA XML Physical Data Model

The hierarchy chart in Fig. 32.2 shows the XML tag structure. One <AddressBook> tag will exist in the XML file and is the root node in the DOM tree. The XML file will contain zero or more <Contact> tags, each representing a unique contact name for the ABA. The XML file will also contain a fixed number of <EmailType> and <PhoneType> tags, representing the choices the user has for identifying the type of email address and phone number, respectively. Listing 32.1 shows three email types and four phone types that are part of the ABA. One contact is shown in this XML file. The first line in this XML file is the XML Declaration statement, used by software programs to identify the file as containing XML tags.

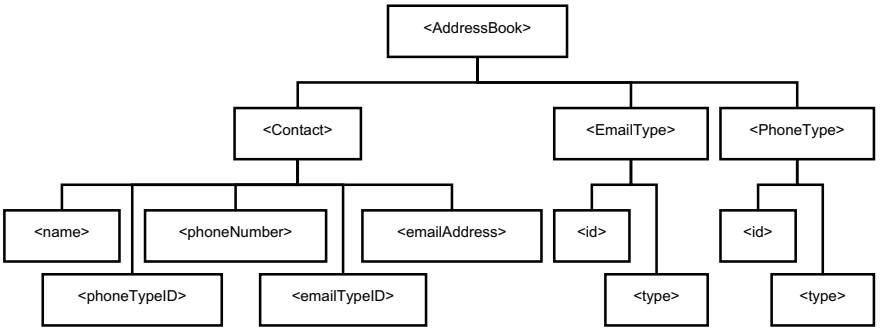

Fig. 32.2 ABA physical data model—XML

Listing 32.1 Sample ABA XML File

```
<?xml version="1.0" encoding="UTF–8" standalone="no"?> <AddressBook>
<EmailType><id>1</id><type>personal</type></EmailType>
<EmailType><id>2</id><type>work</type></EmailType>
<EmailType><id>3</id><type>other</type></EmailType>
<PhoneType><id>1</id><type>cell</type></PhoneType>
<PhoneType><id>2</id><type>home</type></PhoneType>
<PhoneType><id>3</id><type>work</type></PhoneType>
<PhoneType><id>4</id><type>other</type></PhoneType>
<Contact><name>Dave</name><phoneTypeID>1</phoneTypeID>
<phoneNumber>3155551234</phoneNumber><emailTypeID>1</emailTypeID>
<emailAddress>dave@david.com</emailAddress></Contact> </AddressBook>
```

32.2.1.2 OOD: ABA Rdb Physical Data Model

The relational database model in Fig. 32.3 shows the database table structure. The Contact table will contain zero or more rows (aka instances), each representing a unique contact name for the ABA. The database will also contain a fixed number of EmailType and PhoneType rows/instances, representing the choices the user has for identifying the type of email address and phone number, respectively. Listing 32.2 shows the SQL data definition language (DDL) statements to create the AddressBook database. The bottom of this listing includes DML statements to add three email types and four phone types to the respective tables.

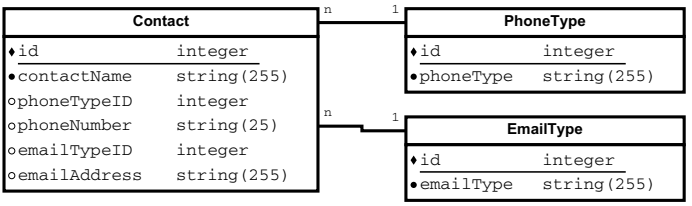

Fig. 32.3 ABA physical data model—Rdb

Listing 32.2 Sample ABA SQL DDL statements

```
drop database if exists AddressBook;
create database AddressBook;
use AddressBook;

create table PhoneType
  (id          INTEGER AUTO_INCREMENT PRIMARY KEY,
  phoneType    VARCHAR(255) NOT NULL);

create table EmailType
  (id          INTEGER AUTO_INCREMENT PRIMARY KEY,
  emailType    VARCHAR(255) NOT NULL);

create table Contact
  (id              INTEGER AUTO_INCREMENT PRIMARY KEY,
  contactName VARCHAR(255) NOT NULL UNIQUE,
  phoneTypeID INTEGER,
  phoneNumber VARCHAR(25),
  emailTypeID INTEGER,
  emailAddress     VARCHAR(255),
  FOREIGN KEY (phoneTypeID)
     REFERENCES PhoneType(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY (emailTypeID)
     REFERENCES EmailType(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

/* Indexes to improve performance */
create index Contact_Index_1
   ON Contact(contactName);
create index PhoneType_Index_1
   ON PhoneType(phoneType);
create index EmailType_Index_1
   ON EmailType(emailType);

/* Add phone types */
INSERT INTO PhoneType(phoneType) VALUES ("cell");
INSERT INTO PhoneType(phoneType) VALUES ("home");
INSERT INTO PhoneType(phoneType) VALUES ("work");
INSERT INTO PhoneType(phoneType) VALUES ("other");

/* Add email types */
INSERT INTO EmailType(emailType) VALUES ("personal");
INSERT INTO EmailType(emailType) VALUES ("work");
INSERT INTO EmailType(emailType) VALUES ("other");
```

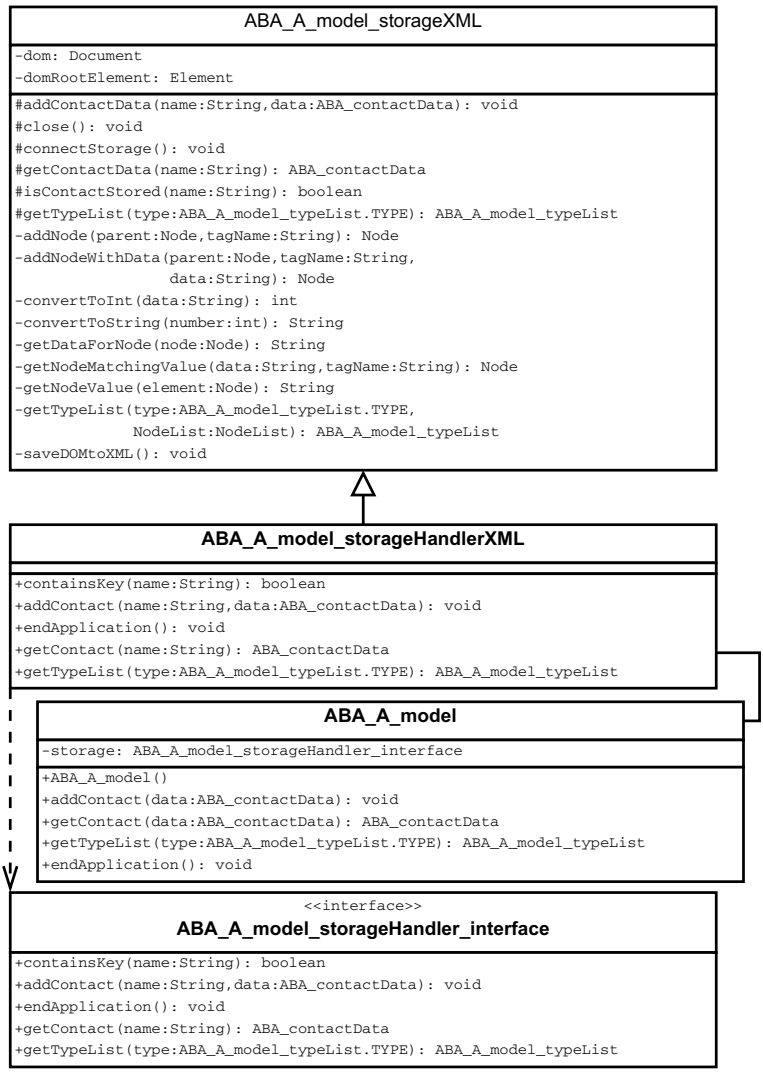

Fig. 32.4 Class diagram—ABA model component for XML

32.2.2 OOD: ABA Design—XML

The class diagram in Fig. 32.4 shows the classes in the model component responsible for the processing of the XML file and DOM. The ABA_A_model_storageXML class is abstract. It contains all of the XML and DOM specific logic for the ABA.XML file. The protected methods (shown with a pound sign "#" before each method name) are called from the ABA_A_model_storageHandlerXML class, which inherits the behavior of the storageXML abstract class and implements the ABA_A_model_storageHandler_interface.

The only attribute shown in the ABA_A_model class is of type ABA_A_model_storageHandler_interface, allowing the model to instantiate a storageHandler for XML or Rdb. Listing 32.3 shows how the model constructor will instantiate one of the storageHandler objects. The storageType parameter shown for the model constructor comes from the ABA_A.main method, which accepts a single parameter to identify whether XML or Rdb is used as the persistent data storage technology.

Listing 32.3 ABA model class constructor

```
private ABA_A_model_storageHandler_interface storage;

public ABA_A_model(String storageType)
{
  if (storageType.equals(ABA_A.ARG_RDB))
    storage = new ABA_A_model_storageHandlerRdb();
  else
    storage = new ABA_A_model_storageHandlerXML();
}
```

Listing 32.4 shows part of the main method, which calls the getFirstArgs method (not shown) to obtain the type of persistent storage to use. The main method then passes this to the constructor, which is responsible for constructing the model object.

Listing 32.4 ABA main method

```
public final static String ARG_RDB = "Rdb";
public final static
String ARG_XML = "XML";

public static void main(String[] args)
{
  String storageType = getFirstArg(args);
  ABA_A_controller aba = new ABA_A_controller(storageType);
  aba.go();
}
```

Below are two questions regarding the design of the model component.

1. Why does the design of the model component use an abstract class?
   An abstract class allows a developer to express behavior to be used by other classes, without having to explicitly construct an object instance (since an abstract class cannot have a constructor method). When a concrete class inherits from an abstract class, it allows the behavior expressed in the abstract class to be included through generalization. In this design, we've flipped the use of generalization (aka inheritance). That is, a superclass (i.e., what is being inherited) generally contains more general behavior while the subclass contains more specific behavior. In this design, the abstract class ABA_A_model_storageXML is the superclass. But it contains specific behavior related to processing an XML file and DOM. In contrast, the subclass ABA_A_model_storageHandlerXML contains general behavior as defined by the ABA_A_model_storageHandler_interface.

2. Why does the design of the model component use an interface?
   The use of this interface is critical to how the model constructor method creates an object instance representing the persistent data storage technology being used. As shown in Listing 32.3, the storage instance variable can refer any object instance whose concrete class implements the ABA_A_model_storageHandler_interface. This allows us to easily add another type of persistent storage (e.g., non-relational database) in some future version of the ABA.

32.2.3 OOD: ABA Design—Rdb

The class diagram in Fig. 32.5 shows the classes in the model component responsible for the processing of the relational database. The ABA_A_model_storageRdb class is abstract. It contains all of the SQL DML specific logic for the AddressBook database. The protected methods (shown with a pound sign "#" before each method name) are called from the ABA_A_model_storageHandlerRdb class, which inherits the behavior of the storageXML abstract class and implements the ABA_A_model_storageHandler_interface.

The only attribute shown in the ABA_A_model class is of type ABA_A_model_storageHandler_interface, allowing the model to instantiate a storageHandler for XML or Rdb. Listing 32.3 shows how the model constructor will instantiate one of the storageHandler objects. The storageType parameter shown for the model constructor comes from the ABA_A.main method, which accepts a single parameter to identify whether XML or Rdb is used as the persistent data storage technology.

Listing 32.4 shows part of the main method, which calls the getFirstArgs method (not shown) to obtain the type of persistent storage to use. The main method then passes this to the constructor, which is responsible for constructing the model object.

Refer to the two questions at the end of Sect. 32.2.2 for a discussion on the use of an abstract class and an interface in this design.

32.2.4 OOD: ABA Design—Summary

The above two sections illustrate one of the strengths of the MVC architectural framework. We are able to use different persistent storage technologies without making any changes to the controller and view components.

The two class diagrams in Figs. 32.4 and 32.5 shows a model component that can be easily extended to use a third type of persistent storage. For example, using a text file would result in creating two new classes: ABA_A_model_storageHandler TextFile which will implement the ABA_A_model_storageHandler_interface and ABA_A_model_storageTextFile which will be an abstract class containing behavior for using a text file as the ABA persistent storage technology. The only other change necessary would be to modify the ABA_A_model constructor method and the getFirstArgs method to add "TXT" as a valid parameter used when starting the ABA.

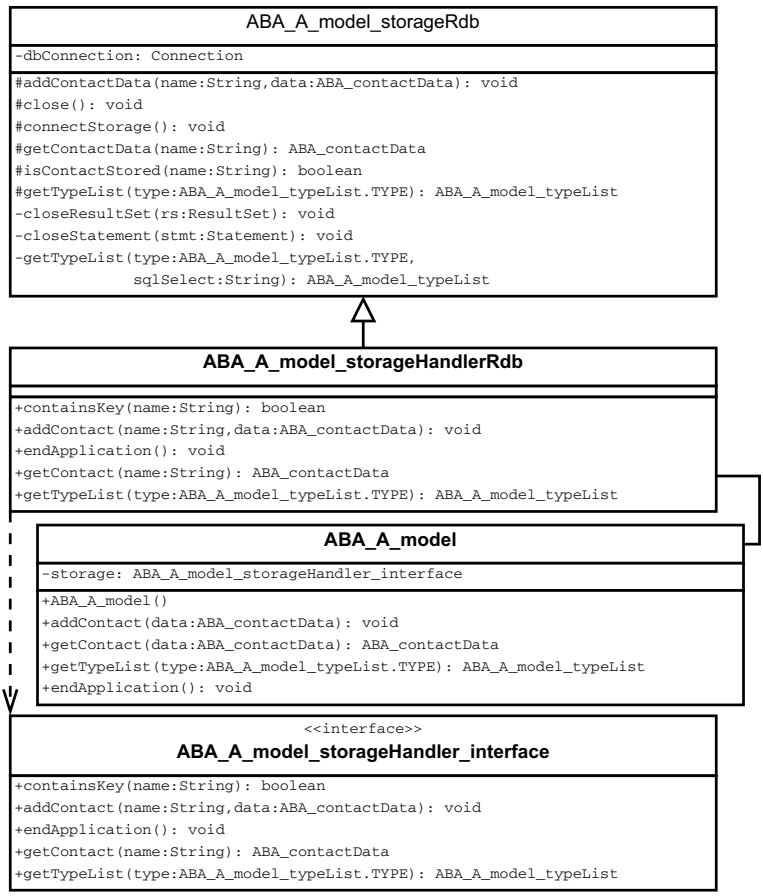

Fig. 32.5 Class diagram—ABA model component for Rdb

32.3 OOD Top-Down Design Perspective

We'll use the personal finances case study to reinforce data design choices as part of a top-down design approach.

32.3.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 12, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings

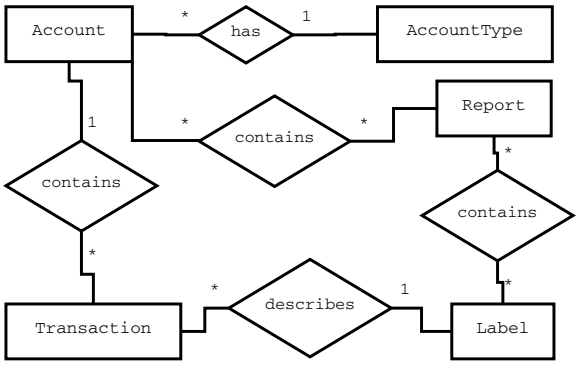

Fig. 32.6 Entity relationship diagram—PF

accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.

- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

32.3.2 OOD Personal Finances: Data Models

The ERD in Fig. 32.6 shows the entities and relationships, based on the requirements listed above. The relationship rules based on the cardinality are as follows. Note the two many-to-many relationships.

- An Account contains many Transactions; A Transaction is for one Account.
- An Account has one AccountType; An AccountType describes many Accounts.
- An Account is included on many Reports; A Report contains many Accounts.
- A Transaction is described by a Label; A Label describes many Transactions.
- A Label is included on many Reports; A Report contains many Labels.

The LDM in Fig. 32.7 shows the attributes associated with each data entity. Four of the entities are using an id field as the primary key. This would be implemented in a physical data model to ensure each id value is unique. The Account primary key is

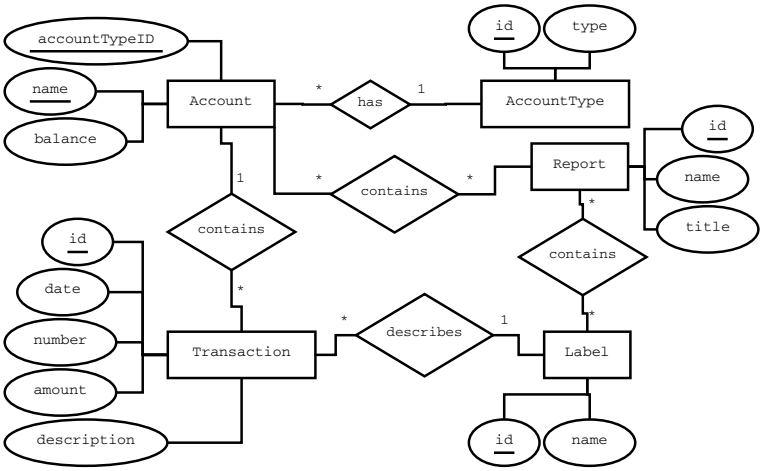

Fig. 32.7 Logical data model—PF

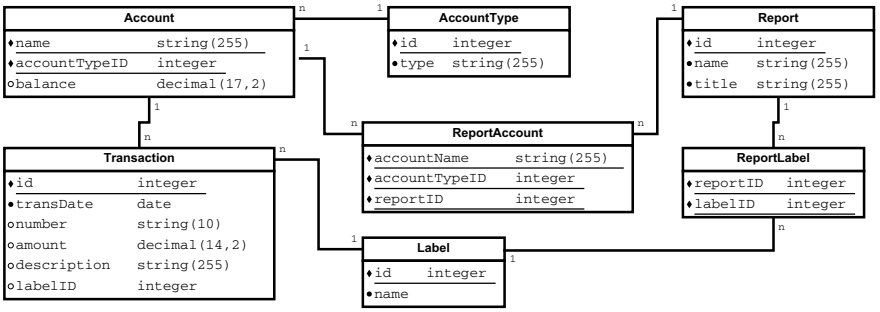

Fig. 32.8 Relational database design—PF

the combination of its name and id value associated with the account type assigned to the account.

The relational database model in Fig. 32.8 shows the translation of the LDM into a relational database. The two many-to-many relationships in the LDM have been replaced with a table, resulting in two additional one-to-many relationships in the physical data model. With a relational database, the id fields would be implemented using the AUTO_INCRMENT feature.

The hierarchy chart in Fig. 32.9 shows the structure of XML tags. With an XML file, the id fields would be implemented to allow the display of the personal finance data in the order they were created. Note the <MaxID> tag, which is used to keep track of the largest id value for each of the four entities having an id field.

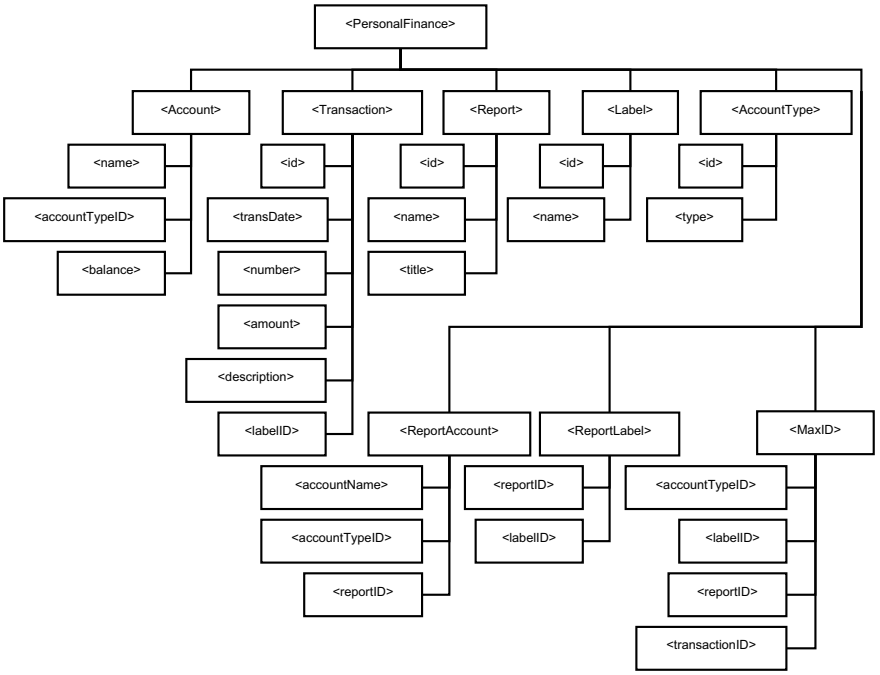

Fig. 32.9 XML design—PF

32.4 Post-conditions

The following should have been learned when completing this chapter.

- You understand how to develop data models, including entity relationship diagrams and fully-attributed logical data models. You also understand that translating a LDM to a physical data model needs to consider the limitations of the physical storage format.
- You understand the process of normalization and have applied normalization forms to a data design.
- You understand how to design XML files, and how to use Java to process an XML file.
- You understand how to design a Rdb, and how to use Java to process data stored in an Rdb.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data flow diagram are design models used in object-oriented solutions to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data flow diagram, UML communication diagram, and UML statechart are design models used to illustrate the behavior of your software design.
- You have created and/or modified models that describe an object-oriented software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Hands-on Exercises

1. Use an existing code solution you've developed, develop alternative data models to show ways in which the application data could be structured. Apply the normalization forms on your logical data models. How good or bad is your persistent data storage design?
2. Use your development of an application you started in Chap. 3 for this exercise. Modify your application to use persistent data storage. Evaluate your data models using the normalization forms.
3. Continue hands-on exercise 3 from Chap. 12 by developing a design for a persistent data storage technology. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 12 for details on each domain.

    - Airline reservation and seat assignment
    - Automated teller machine (ATM)
    - Bus transportation system
    - Course-class enrollment
    - Digital library
    - Inventory and distribution control
    - Online retail shopping cart
    - Personal calendar
    - Travel itinerary