Wolf Inns Database Management System Design

Project Report 3

CSC 540 Database Management Concepts and Systems

Team 5: Xiaohui Ellis, Abhijeet Krishnan, Michael Moran, Chekad Sarami

Transactional Code

Two operations were selected to be implemented as transactions. The first is the check-in process which has to ensure that billing records are created, the stay record is created, dedicated staff are assigned to presidential suites, and a service record gets created to log the check-in process. The second is the checkout process which must ensure that the checkout date and time get updated, dedicated staff are released, and a service record is created logging the checkout process.

Below are outlines of the business logic for each transaction followed by the code for each transaction. If any step of the operation fails then the transaction is rolled back to the savepoint and autocommit is turned back on.

Check-in Program Logic

1. Parameters are passed into the method for the stays and billing_info records as well as the staff member who processed the check-in.
2. Turn off autocommit.
3. Create a savepoint.
4. Create billing_info record using parameters and return a BillingInfo object.
5. Create stays record using billingId from the BillingInfo record and parameters.
6. Retrieve the room record.
7. Check if the room is a presidential suite.
    7.1. Retrieve available catering staff assigned to the hotel.
    7.2. Retrieve available room service staff assigned to the hotel.
    7.3. Use the first staff record for each type and assign them to the presidential suite.
8. Add a service record to record the staff who served the customer for the check-in process.
9. Commit the transaction.
10. Turn on autocommit.

Check-in Transactional Code

```
public static boolean checkinStay(int hotelId, String roomNumber, int customerId, int numOfGuests, int
servicingStaffId, String responsiblePartySSN, String address, String city, String state, String
payMethodCode, String cardNumber) {

                Connection connection = null;
                PreparedStatement statement = null;
                Savepoint save1 = null;

                try {
```

```
                        connection = DatabaseConnection.getConnection();
                        connection.setAutoCommit(false);
                        save1 = connection.setSavepoint();

                        // Create the billing record
                        BillingInfo billingInfo =
MaintainBillingRecords.createBillingInfo(responsiblePartySSN, address, city, state, payMethodCode,
cardNumber, connection);
                        if (null == billingInfo) {throw new SQLException("Error seems to have occured.
Check the logs.");}

                        // Create the stays record
                        Stays stay = InformationProcessing.createStay(hotelId, roomNumber, customerId,
numOfGuests, billingInfo.getBillingId(), connection);
                        if (null == stay) {throw new SQLException("Error seems to have occured. Check the
logs.");}

                        // Retrieve room information to determine if it is a presidential suite
                        Rooms room = InformationProcessing.retrieveRoom(hotelId, roomNumber);
                        if (null == room) {throw new SQLException("Error seems to have occured. Check the
logs.");}

                        // Check for presidential suite
                        if (room.getCategoryCode().equals("PRES")) {
                                // Retrieve available catering staff
                                ArrayList<ServiceStaff> cateringStaff =
retrieveAvailableCateringStaff(hotelId);
                                if (null == cateringStaff && cateringStaff.size() > 0) {throw new
SQLException("Error seems to have occured. Check the logs.");}

                                // Retrieve available room service staff
                                ArrayList<ServiceStaff> roomServiceStaff =
retrieveAvailableRoomServiceStaff(hotelId);
                                if (null == roomServiceStaff && roomServiceStaff.size() > 0) {throw new
SQLException("Error seems to have occured. Check the logs.");}

                                int cateringStaffId = cateringStaff.get(0).getStaffId();
                                int roomServiceStaffId = roomServiceStaff.get(0).getStaffId();

                                // Assign dedicated staff to presidential suite
                                if (!assignDedicatedPresidentialSuiteStaff(hotelId, roomNumber,
cateringStaffId, roomServiceStaffId, connection)) {
                                        throw new SQLException("Error seems to have occured. Check the
logs.");
                                }
                        }

                        // Add a checkin service record
                        ServiceRecords serviceRecord =
MaintainingServiceRecords.createCheckinCheckoutRecord(stay.getStayId(), servicingStaffId, true,
connection);
                        if (null == serviceRecord) {throw new SQLException("Error seems to have occured.
Check the logs.");}

                        connection.commit();

                        return true;
```

```
		} catch (SQLException ex) {
			try { connection.rollback(save1); } catch (Exception e) {};
			return false;
		} finally {
			// Attempt to close all resources, ignore failures.
			try { connection.setAutoCommit(true); } catch (Exception ex) {};
			try { connection.close(); } catch (Exception ex) {};
		}
	}
```

Checkout Program Logic

1. Parameters are passed into the method for the stays as well as the staff member who processed the check-in.
2. Turn off autocommit.
3. Create a savepoint.
4. Add a service record to record the staff who served the customer for the checkout process.
5. Commit the service record to avoid locking the associated stay record.
6. Update the checkout date and time using the stay parameter. This will be needed to calculate room charge.
7. Retrieve the room record.
8. Check if the room is a presidential suite.
   8.1. Unassign dedicated staff from the presidential suite.
9. Generate the billing total and store it.
10. Use the billing total to update the totalCharges field on the billing_info record using the billingId from the stay parameter.
11. Commit the transaction.
12. Turn on autocommit.

Checkout Transactional Code

```
public static boolean checkoutStay(Stays stay, int servicingStaffId) {

		Connection connection = null;
		PreparedStatement statement = null;
		Savepoint save1 = null;


		try {
			connection = DatabaseConnection.getConnection();
			connection.setAutoCommit(false);
			save1 = connection.setSavepoint();

			// Add checkout service record
```

```
                        ServiceRecords serviceRecord =
MaintainingServiceRecords.createCheckinCheckoutRecord(stay.getStayId(), servicingStaffId, false,
connection);
                        if (null == serviceRecord) {throw new SQLException("Error seems to have occurred.
Check the logs.");}

                        connection.commit();

                        if(!HotelStayOperations.updateStayCheckoutDateTime(stay.getStayId(), connection))
{
                                throw new SQLException("Error seems to have occured. Check the logs.");
                        }

                        // Retrieve room information to determine if it is a presidential suite
                        Rooms room = InformationProcessing.retrieveRoom(stay.getHotelId(),
stay.getRoomNumber());
                        if (null == room) {throw new SQLException("Error seems to have occured. Check the
logs.");}

                        // Check for presidential suite
                        if (room.getCategoryCode().equals("PRES")) {
                                // Unassign dedicated staff for presidential suite
                                if (!unassignDedicatedPresidentialSuiteStaff(stay.getHotelId(),
stay.getRoomNumber(), connection)) {
                                        throw new SQLException("Error seems to have occured. Check the
logs.");
                                }
                        }

                        double totalCharges =
MaintainBillingRecords.generateBillingTotal(stay.getStayId());
                        if (totalCharges < 0) {throw new SQLException("Error seems to have occured. Check
the logs.");}

                        if (!MaintainBillingRecords.updateBillingInfoTotalCharges(stay.getBillingId(),
totalCharges, connection)) {
                                throw new SQLException("Error seems to have occured. Check the logs.");
                        }

                        connection.commit();

                        // Generate and print the itemized receipt
                        if(!generateItemizedReceipt(stay)) {throw new SQLException("Error seems to have
occured. Check the logs.");}

                        return true;

                } catch (SQLException ex) {
                        try { connection.rollback(save1); } catch (Exception e) {};
                        ex.printStackTrace();
                        return false;
                } finally {
                        // Attempt to close all resources, ignore failures.
                        try { connection.setAutoCommit(true); } catch (Exception ex) {};
                        try { connection.close(); } catch (Exception ex) {};
                }
        }
```

High-level Design Decisions

Our code is organized into the following files -

- `DatabaseConnection.java`: exposes methods to easily create connections to our database
- Files to expose APIs to implement the tasks and operations mentioned in the narrative
    - `InformationProcessing.java`
    - `HotelStayOperations.java`
    - `MaintainBillingRecords.java`
    - `MaintainingServiceRecords.java`
    - `Reporting.java`
- We implemented CRUD for tables where those operations are allowed.
- Classes for each of our database entities.
- APIs for executing SQL statements against a single table follow this paradigm:
    - Create - Will accept primitive types for each value. The method will return an object representing the record just created.
    - Retrieve - Single retrieves accept a key as parameters and return a single object representing the record. Retrieve all APIs return an ArrayList of objects for all records returned.
    - Update - Accepts an object representing the record with the updated values. Returns a boolean indicating success or failure.
    - Delete- Accepts an object representing the record to be deleted. Returns a boolean indicating success or failure.
- Classes which contain methods to implement the console frontend divided into four based on the four high level menu options.
    - Contain menu methods to handle display and interaction with sub-menus.
    - Contain menuOption methods to handle display and interaction with the various menu options.
- `TestRunner.java, TestJUnit.java`: JUnit tests for each of our APIs
- `WolfInns.java`: contains the main method to load the console frontend

Functional Roles

Michael Moran (mamoran) - Database Designer (prime), Software Engineer (prime)
Xiaohui Ellis (xzheng6) - Application Programmer (prime), Test Plan Engineer (backup)
Chekad Sarami (csarami) - Application Programmer (backup), Database Designer (backup)
Abhijeet Krishnan (akrish13) - Test Plan Engineer (prime), Software Engineer (backup)