Phase 5: Apex Programming and Automation

Introduction

This phase was designed to extend the declarative capabilities of Salesforce by implementing custom Apex code to enforce business rules, automate complex validations, and handle batch processing for scale. These enhancements were necessary to meet the rigorous operational requirements of the Student CRM system.

Development of Apex Utility Classes

To maintain clean, modular, and reusable code, I developed Apex utility classes. These classes encapsulate critical business logic such as enrollment validation, fee status checks, and student progress tracking.

- For example, the EnrollmentService class ensures no student can be actively enrolled in the same course twice, preventing data duplication and maintaining enrollment integrity.
- Centralizing logic in utility classes reduces code duplication and simplifies future enhancements.

  Apex Class code for Enrollement Service:

```
public class EnrollmentService {


    // Method to prevent overlapping enrollments for the same student
    public static void preventOverlappingEnrollments(List<Enrollment__c> newEnrollments,
Map<Id, Enrollment__c> oldMap) {
        Set<Id> studentIds = new Set<Id>();
        for (Enrollment__c e : newEnrollments) {
            studentIds.add(e.Student__c);
        }
```

```
        Map<Id, List<Enrollment__c>> existingEnrollments = new Map<Id,
    List<Enrollment__c>>();
        for (Enrollment__c e : [SELECT Id, Student__c, Course__c, Status__c FROM
    Enrollment__c WHERE Student__c IN :studentIds]) {
            if (!existingEnrollments.containsKey(e.Student__c)) {
                existingEnrollments.put(e.Student__c, new List<Enrollment__c>());
            }
            existingEnrollments.get(e.Student__c).add(e);
        }

        for (Enrollment__c newEnroll : newEnrollments) {
            if (existingEnrollments.containsKey(newEnroll.Student__c)) {
                for (Enrollment__c existingEnroll :
    existingEnrollments.get(newEnroll.Student__c)) {
                    if (existingEnroll.Course__c == newEnroll.Course__c &&
    existingEnroll.Status__c == 'Active' && newEnroll.Status__c == 'Active') {
                        newEnroll.addError('Student is already actively enrolled in this course.');
                    }
                }
            }
        }
    }
}
```

Implementation of Apex Triggers

Apex triggers were developed to intercept DML operations (insert/update) on key objects like Enrollment and Fee.

- These triggers call methods from the utility classes to validate business rules before changes are committed.
- The use of trigger handlers improves maintainability by separating logic from trigger definitions.
- For example, when a new grade record is inserted, triggers ensure that it falls within acceptable ranges and updates related GPA calculations.

  Apex Triggers for Enrollment:

```
trigger EnrollmentTrigger on Enrollment__c (before insert, before update) {
    if (Trigger.isBefore) {
        if (Trigger.isInsert || Trigger.isUpdate) {
            EnrollmentService.preventOverlappingEnrollments(Trigger.new, Trigger.oldMap);
        }
    }
}
```

Robust Test Classes

Testing is a critical component for Apex deployment. I wrote comprehensive test methods covering:

- Successful creation and update scenarios.
- Validation of exception throwing when business rules are violated.
- Testing with bulk data to simulate real-world use.
- Achieving high code coverage (>75%) required by Salesforce deployment policies.

These tests guarantee that the Apex code behaves correctly in all expected scenarios.

Apex Test Class for Enrollment Service and Trigger:

```
@isTest
public class EnrollmentServiceTest {
```

```
    static testMethod void testPreventOverlappingEnrollments() {
        // Create a student and courses
        Student__c student = new Student__c(Name='Test Student');
        insert student;

        Course__c course = new Course__c(Name='Mathematics');
        insert course;

        // Insert one active enrollment
        Enrollment__c enroll1 = new Enrollment__c(Student__c = student.Id, Course__c = course.Id, Status__c = 'Active');
        insert enroll1;

        // Try inserting second active enrollment in same course - should fail
        Enrollment__c enroll2 = new Enrollment__c(Student__c = student.Id, Course__c = course.Id, Status__c = 'Active');
        Test.startTest();
        try {
            insert enroll2;
            System.assert(false, 'Expected exception not thrown');
        } catch (DmlException e) {
            System.assert(e.getMessage().contains('Student is already actively enrolled'));
```

```
        }
        Test.stopTest();
    }
}
```

Batch Apex and Scheduling

Large data volumes necessitated asynchronous processing to maintain performance.

- Batch Apex classes were implemented to send periodic fee reminders to students and parents automatically.
- Scheduled Apex jobs ensure the batch executes daily without manual intervention.
- Error handling within batches captures and logs failures, maintaining system reliability.

Batch Apex Scheduling to send Fee Remainder Emails:

```
global class FeeReminderBatch implements Database.Batchable<SObject>,
Database.Stateful {
    global Database.QueryLocator start(Database.BatchableContext BC) {
        return Database.getQueryLocator([
            SELECT Id, Student__r.Email, Due_Date__c, Status__c
            FROM Fee__c
            WHERE Status__c = 'Pending' AND Due_Date__c = :Date.today().addDays(3)
        ]);
    }

    global void execute(Database.BatchableContext BC, List<Fee__c> scope) {
        List<Messaging.SingleEmailMessage> mails = new
List<Messaging.SingleEmailMessage>();
        for (Fee__c fee : scope) {
```

```
            if (fee.Student__r.Email != null) {
                Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
                mail.setToAddresses(new String[] { fee.Student__r.Email });
                mail.setSubject('Fee Payment Reminder');
                mail.setPlainTextBody('Dear Student, your fee is due on ' +
String.valueOf(fee.Due_Date__c) + '. Please pay it on time.');
                mails.add(mail);
            }
        }
        if (!mails.isEmpty()) {
            Messaging.sendEmail(mails);
        }
    }

    global void finish(Database.BatchableContext BC) {}
}
```

Scheduling the Batch Apex:

```
global class FeeReminderScheduler implements Schedulable {
    global void execute(SchedulableContext SC) {
        FeeReminderBatch batch = new FeeReminderBatch();
        Database.executeBatch(batch);
    }
}
```

Exception and Error Handling

User experience was enhanced by providing clear, actionable error messages through the use of `addError()` in triggers.

- This prevents faulty data entry while informing end users exactly what corrections are needed.
- Batch jobs and scheduled processes also include try-catch blocks to gracefully handle unexpected errors.

  Test Class for Batch:

```
@isTest
public class FeeReminderBatchTest {
    @isTest static void testBatch() {
        Student__c s = new Student__c(Name='Test Student', Email='test@example.com');
        insert s;
        Fee__c fee = new Fee__c(Student__c = s.Id, Due_Date__c = Date.today().addDays(3),
Status__c = 'Pending');
        insert fee;

        Test.startTest();
        FeeReminderBatch batch = new FeeReminderBatch();
        Database.executeBatch(batch);
        Test.stopTest();
    }
}
```

Outcomes and Benefits

The Apex code implementation substantially increased the Student CRM’s capabilities by:

- Automating complex validations that were previously manual or impossible declaratively.
- Reducing the administrative workload via automated fee reminders and status checks.
- Ensuring data quality and integrity across the system.
- Enabling scalable batch processing for growing student data.
- Improving user satisfaction through meaningful validation messages.