Phase 5 — Apex Programming (Developer)

Goal: Implement advanced custom logic beyond declarative automation: keep accurate member counts, compute fines, prevent invalid deletions, and run scheduled/batch jobs for overdue processing. This document is a step-by-step developer plan with patterns, ready-to-copy Apex snippets, testing guidance, deployment checklist, and operational tips.

Assumptions / Prerequisites

This plan assumes the following custom objects and fields exist. Update API names to match your org if they differ.

- **Objects**
- `Book__c` — fields: `Available_Quantity__c (Integer)`, `Fine_Per_Day__c (Decimal)`
- `Member__c` — fields: `Total_Issued_Books__c (Integer)`
- `Book_Transaction__c` — fields: `Book__c (Lookup)`, `Member__c (Lookup)`, `Status__c (Picklist: Issued, Returned, Cancelled)`, `Due_Date__c (Date)`, `Return_Date__c (Date)`, `Overdue_Flag__c (Checkbox)`
- `Fine__c` — fields: `Member__c (Lookup)`, `Transaction__c (Lookup)`, `Amount__c (Currency)`, `Is_Paid__c (Checkbox)`

- **Admin configuration**

- A Custom Metadata or Custom Setting record to store default fine rules (e.g. default per-day fine, grace period). This lets admins change rates without code changes.

High-level design & patterns

1. **Trigger Handler Pattern** — Always keep triggers thin and delegate logic to a handler/service class. This makes unit testing and bulkification easier.
2. **Service / Helper Classes** — Put reusable logic (calculateFine, checkBookAvailability) in `LibraryHelper` or `LibraryService` classes.
3. **Bulkification** — All trigger and batch code must be bulk-safe (no SOQL/DML in loops, use Maps and aggregate counts).
4. **Asynchronous Processing** — Use Batchable for heavy processing (overdue calculation), Queueable for small async work, and Schedulable for nightly orchestration.
5. **Error Handling & Auditing** — Log exceptions to a custom `Error_Log__c` object inside batch/async contexts for post-mortem.
6. **Config-driven** — Use Custom Metadata for fine rates and thresholds.

---

1) Trigger scaffold (single trigger + handler)

Trigger file: `BookTransactionTrigger.trigger`

```
trigger BookTransactionTrigger on Book_Transaction__c (
    after insert,
    after update,
    before delete
) {
    if (Trigger.isAfter) {
        if (Trigger.isInsert)
BookTransactionTriggerHandler.afterInsert(Trigger.new);
        if (Trigger.isUpdate)
BookTransactionTriggerHandler.afterUpdate(Trigger.new, Trigger.oldMap);
    }

    if (Trigger.isBefore && Trigger.isDelete) {
        BookTransactionTriggerHandler.beforeDelete(Trigger.old);
    }
}
```

Notes - Keep the trigger file minimal and delegate all work. - Use `Trigger.new`, `Trigger.old`, `Trigger.oldMap` appropriately.

---

2) Trigger handler: `BookTransactionTriggerHandler`

Purpose: update member counts, call fine calculation where appropriate, and enforce deletion rules.

```
public with sharing class BookTransactionTriggerHandler {

    public static void afterInsert(List<Book_Transaction__c> newList) {
        // 1) Update Member.Total_Issued_Books__c for newly issued
transactions
        Map<Id, Integer> incByMember = new Map<Id, Integer>();
        for (Book_Transaction__c t : newList) {
            if (t.Member__c == null) continue;
            if (t.Status__c == 'Issued') {
                Integer c = incByMember.get(t.Member__c);
                if (c == null) c = 0;
                incByMember.put(t.Member__c, c + 1);
            }
        }

        if (!incByMember.isEmpty()) {
            List<Member__c> membersToUpdate = [
                SELECT Id, Total_Issued_Books__c
                FROM Member__c
                WHERE Id IN :incByMember.keySet()
            ];
            for (Member__c m : membersToUpdate) {
                Integer current = (m.Total_Issued_Books__c == null) ? 0 :
Integer.valueOf(m.Total_Issued_Books__c);
```

```
                m.Total_Issued_Books__c = current + incByMember.get(m.Id);
            }
            update membersToUpdate;
        }

        // 2) Optionally: Check fine rules for transactions created as
already overdue (rare)
        List<Fine__c> finesToInsert = new List<Fine__c>();
        for (Book_Transaction__c t : newList) {
            if (t.Id == null) continue; // defensive
            // If transaction has Return_Date and is returned on insert,
compute fine
            if (t.Return_Date__c != null) {
                Decimal amount = LibraryHelper.calculateFine(t.Member__c,
t.Id);
                if (amount > 0) {
                    finesToInsert.add(new Fine__c(Member__c = t.Member__c,
Transaction__c = t.Id, Amount__c = amount));
                }
            }
        }
        if (!finesToInsert.isEmpty()) insert finesToInsert;
    }

    public static void afterUpdate(List<Book_Transaction__c> newList, Map<Id,
Book_Transaction__c> oldMap) {
        // Handle status transitions (e.g., Issued -> Returned) and update
counts & fines
        Map<Id, Integer> decByMember = new Map<Id, Integer>();
        List<Fine__c> finesToInsert = new List<Fine__c>();

        for (Book_Transaction__c t : newList) {
            Book_Transaction__c oldT = oldMap.get(t.Id);
            if (oldT == null) continue;

            // If changed from Issued -> Returned, decrement member issued
count and compute fine
            if (oldT.Status__c == 'Issued' && t.Status__c == 'Returned') {
                if (t.Member__c != null) {
                    Integer c = decByMember.get(t.Member__c);
                    if (c == null) c = 0;
                    decByMember.put(t.Member__c, c + 1);
                }

                Decimal amount = LibraryHelper.calculateFine(t.Member__c,
t.Id);
                if (amount > 0) {
                    finesToInsert.add(new Fine__c(Member__c = t.Member__c,
Transaction__c = t.Id, Amount__c = amount));
                }
            }
            // Handle other transitions if needed (Cancelled, Reissued etc.)
        }

        // Update members decremented
        if (!decByMember.isEmpty()) {
            List<Member__c> membersToUpdate = [SELECT Id,
Total_Issued_Books__c FROM Member__c WHERE Id IN :decByMember.keySet()];
            for (Member__c m : membersToUpdate) {
                Integer current = (m.Total_Issued_Books__c == null) ? 0 :
Integer.valueOf(m.Total_Issued_Books__c);
                m.Total_Issued_Books__c = Math.max(0, current -
decByMember.get(m.Id));
            }
            update membersToUpdate;
        }

        if (!finesToInsert.isEmpty()) insert finesToInsert;
    }

    public static void beforeDelete(List<Book_Transaction__c> oldList) {
        for (Book_Transaction__c t : oldList) {
            if (t.Status__c == 'Issued') {
                t.addError('Cannot delete a transaction for a book that is
currently issued. Return the book first.');
            }
        }
    }
}
```

Notes - Use `addError()` in `before delete` to prevent deletion in a user-friendly way (surface error on UI). - Protect against null Member__c. - Always bulkify — all operations above work with lists and maps.

---

3) `LibraryHelper` service class (core reusable logic)

Responsibilities: compute fines, check availability, read configuration.

```
public with sharing class LibraryHelper {

    // Example: returns calculated fine amount for a transaction (0 if none)
    public static Decimal calculateFine(Id memberId, Id transactionId) {
        if (transactionId == null) return 0;
        Book_Transaction__c tx = [
            SELECT Id, Due_Date__c, Return_Date__c, Book__r.Fine_Per_Day__c
            FROM Book_Transaction__c
            WHERE Id = :transactionId
            LIMIT 1
        ];

        if (tx == null || tx.Due_Date__c == null) return 0;

        Date endDate = (tx.Return_Date__c != null) ? tx.Return_Date__c :
Date.today();
        Integer daysOverdue = 0;
        if (endDate > tx.Due_Date__c) {
            daysOverdue = tx.Due_Date__c.daysBetween(endDate);
        }

        Decimal perDay = (tx.Book__r != null && tx.Book__r.Fine_Per_Day__c !=
null) ? tx.Book__r.Fine_Per_Day__c : 0;
        return perDay * daysOverdue;
    }

    public static Boolean checkBookAvailability(Id bookId) {
        if (bookId == null) return false;
        Book__c b = [SELECT Id, Available_Quantity__c FROM Book__c WHERE Id
= :bookId LIMIT 1];
        return (b != null && b.Available_Quantity__c != null &&
b.Available_Quantity__c > 0);
    }

    // Optional: method to read fine-per-day from Custom Metadata with
fallback
    public static Decimal getDefaultFinePerDay() {
        // Pseudocode: query custom metadata; fallback to a hardcoded default
        return 5; // currency units per day
    }
}
```

Notes - Consider reading per-day fine from Custom Metadata for admin control. - `calculateFine` should be deterministic and free of side effects (so it's safe to call from triggers, batch, tests).

---

4) Batch Apex — daily overdue scan & fine creation

Use case: run nightly to find overdue `Book_Transaction__c` records (Status = 'Issued' and Due_Date__c < today) and create `Fine__c` records (or update existing ones).

```
global class OverdueFineBatch implements Database.Batchable<sObject>,
Database.Stateful {

    global Database.QueryLocator start(Database.BatchableContext bc) {
        Date today = Date.today();
        String q = 'SELECT Id, Member__c, Book__c, Due_Date__c FROM
Book_Transaction__c WHERE Status__c = \'' + 'Issued' + '\' AND Due_Date__c
< :today';
```

```
        return Database.getQueryLocator([
            SELECT Id, Member__c, Book__c, Due_Date__c
            FROM Book_Transaction__c
            WHERE Status__c = 'Issued' AND Due_Date__c < :today
        ]);
    }

    global void execute(Database.BatchableContext bc,
List<Book_Transaction__c> scope) {
        List<Fine__c> fines = new List<Fine__c>();
        List<Book_Transaction__c> txToUpdate = new
List<Book_Transaction__c>();

        for (Book_Transaction__c tx : scope) {
            try {
                Decimal amount = LibraryHelper.calculateFine(tx.Member__c,
tx.Id);
                if (amount > 0) {
                    fines.add(new Fine__c(Member__c = tx.Member__c,
Transaction__c = tx.Id, Amount__c = amount));
                    tx.Overdue_Flag__c = true;
                    txToUpdate.add(tx);
                }
            } catch (Exception ex) {
                // Log error to a custom object for troubleshooting
                // Error_Log__c err = new Error_Log__c(...);
                // insert err; (keep in mind DML limits — consider collecting
and inserting outside loop)
            }
        }

        if (!fines.isEmpty()) insert fines;
        if (!txToUpdate.isEmpty()) update txToUpdate;
    }

    global void finish(Database.BatchableContext bc) {
        // Optionally send a summary email to admins or kick off another job
    }
}
```

Notes - Use `Database.getQueryLocator` for large datasets. - Keep scope size reasonable (default 200). Use Database.executeBatch(batch, 200) in Scheduler. - Use `Stateful` only if you need to hold state across execute calls.

---

5) Scheduled Apex — nightly orchestration

Class: `OverdueScheduler` — schedules the `OverdueFineBatch` nightly and optionally a notification job.

```
global class OverdueScheduler implements Schedulable {
    global void execute(SchedulableContext sc) {
        // kick off the batch
        Database.executeBatch(new OverdueFineBatch(), 200);

        // optionally: call a separate job to email members with overdue
items
    }
}

// Example: schedule via System.schedule('Nightly Overdue', '0 0 2 * * ?',
new OverdueScheduler());
```

Notes - Use a cron expression that runs in your org timezone (example above runs at 02:00 daily). - Schedule once in production after deployment (or create a setup UI for admins to toggle schedule).

---

6) Test classes & coverage strategy

Goals: achieve ≥75% coverage and verify business rules. Use `seeAllData=false` and create all test data within tests.

Tests to write:

1. **Trigger — after insert**
2. Create Member & Book; create a `Book_Transaction__c` with `Status__c = 'Issued'`; insert; assert `Member.Total_Issued_Books__c` incremented.
3. **Trigger — after update (Issued -> Returned)**
4. Create a record with Issued, then update to Returned with `Return_Date__c` in past. Assert `Total_Issued_Books__c` decremented and a `Fine__c` exists if overdue.
5. **Trigger — before delete**
6. Attempt to delete a `Book_Transaction__c` with `Status__c = 'Issued'` and assert `DmlException` thrown or error prevented.
7. **LibraryHelper.calculateFine**
8. Unit test multiple scenarios: returned late, returned on time, not returned yet (use Batch execution), ensure calculations match expected numbers.
9. **Batch**
10. Create overdue transactions; run batch inside `Test.startTest()` / `Test.stopTest()` and assert `Fine__c` records created and transactions marked overdue.
11. **Scheduler**
12. Use `System.schedule()` inside a test to execute the scheduled job and verify it runs (it runs within Test.stopTest()).

Sample test skeleton

```
@IsTest
private class BookTransactionTests {
    static testMethod void testAfterInsertUpdatesMemberCount() {
        Member__c m = new Member__c(Name='T1'); insert m;
        Book__c b = new Book__c(Name='B1', Available_Quantity__c = 1,
Fine_Per_Day__c = 10); insert b;

        Test.startTest();
        Book_Transaction__c tx = new Book_Transaction__c(Book__c = b.Id,
Member__c = m.Id, Status__c = 'Issued', Due_Date__c =
Date.today().addDays(7));
        insert tx;
        Test.stopTest();

        m = [SELECT Total_Issued_Books__c FROM Member__c WHERE Id = :m.Id];
        System.assertEquals(1, Integer.valueOf(m.Total_Issued_Books__c));
    }

    // Add tests for update -> returned, before delete, batch and scheduler
}
```

Tips for tests - Use `Test.startTest()` and `Test.stopTest()` to run asynchronous and scheduled code synchronously in tests. - Assert exact values and negative cases. - Cover error handling paths (e.g., missing fields) where possible.

---

7) Deployment checklist & best practices

1. **Develop in a sandbox** (dev/prod-like), not directly in production.
2. **Lint & Static analysis**: run PMD or similar to catch anti-patterns.
3. **Run all tests**: `sfdx force:apex:test:run` or via UI. All tests must pass and coverage should be ≥75%.
4. **Use Change Sets or SFDX** to deploy classes/triggers and to schedule the job in production.
5. **Post-deploy**: schedule `OverdueScheduler` in production if not scheduled via metadata.
6. **Monitor**: check Apex Jobs, Scheduled Jobs, and Setup -> Jobs for failures.

---

8) Observability, logging & error recovery

- **Error_Log__c**: create a lightweight custom object to capture failures from Batch/Queueable jobs (fields: `Context__c`, `Message__c`, `Stack_Trace__c`, `Record_Ids__c`).
- **Notifications**: send admin email on batch failure by catching exceptions inside `finish()` and calling `Messaging.sendEmail()`.
- **Retries**: if a batch fails due to transient reasons, consider re-queuing it from `finish()`.

---

9) Security & sharing

- Use `with sharing` for classes that should respect user sharing; use `without sharing` only when appropriate and documented.

- Enforce FLS in Apex when exposing fields to the UI or external APIs: use `Schema.sObjectType...isAccessible()/isCreateable()` checks or `Security.stripInaccessible()`.

---

10) Performance & Governor limit considerations

- Avoid SOQL and DML in loops.
- Combine DML statements into lists and perform single DML per object type where possible.
- Use `Database.getQueryLocator` for large queries.
- Use selective filters in queries and add indexes for large tables if needed.

---

11) Additional improvements / future work

- Add a dedicated `LoanPolicy__mdt` (Custom Metadata) to manage fine rates, grace period, and maximum fine cap.
- Add Platform Events for `TransactionReturnedEvent` so other systems can react asynchronously.
- Add an Apex REST endpoint to allow external systems to request availability checks or fine status.
- Add UI components to show member fine history and bulk payment processing.

---

12) Quick checklist for each user story

- [ ] Implement trigger + handler
- [ ] Implement `LibraryHelper` methods
- [ ] Implement `OverdueFineBatch` and `OverdueScheduler`
- [ ] Unit tests for each path (insert, update, delete, batch, scheduler)
- [ ] Create Custom Metadata for fine policy and use it in `LibraryHelper`
- [ ] Add error logging and admin notifications
- [ ] Deploy to production and schedule the job

If you want, I can: - split the above into individual Apex files ready for SFDX packaging, or - produce complete test classes for each unit (fully runnable, `seeAllData=false`), or - convert the schedule cron expression to a specific timezone/time for your org.

Tell me which of the above you'd like next and I will prepare the files (Apex classes + triggers + tests) formatted for copy/paste or SFDX.