Bhasha Quest - Design Rationale Document

CS3307A Deliverable 2

Student: Hardik Shrestha (251163014)

Date: November 10, 2025

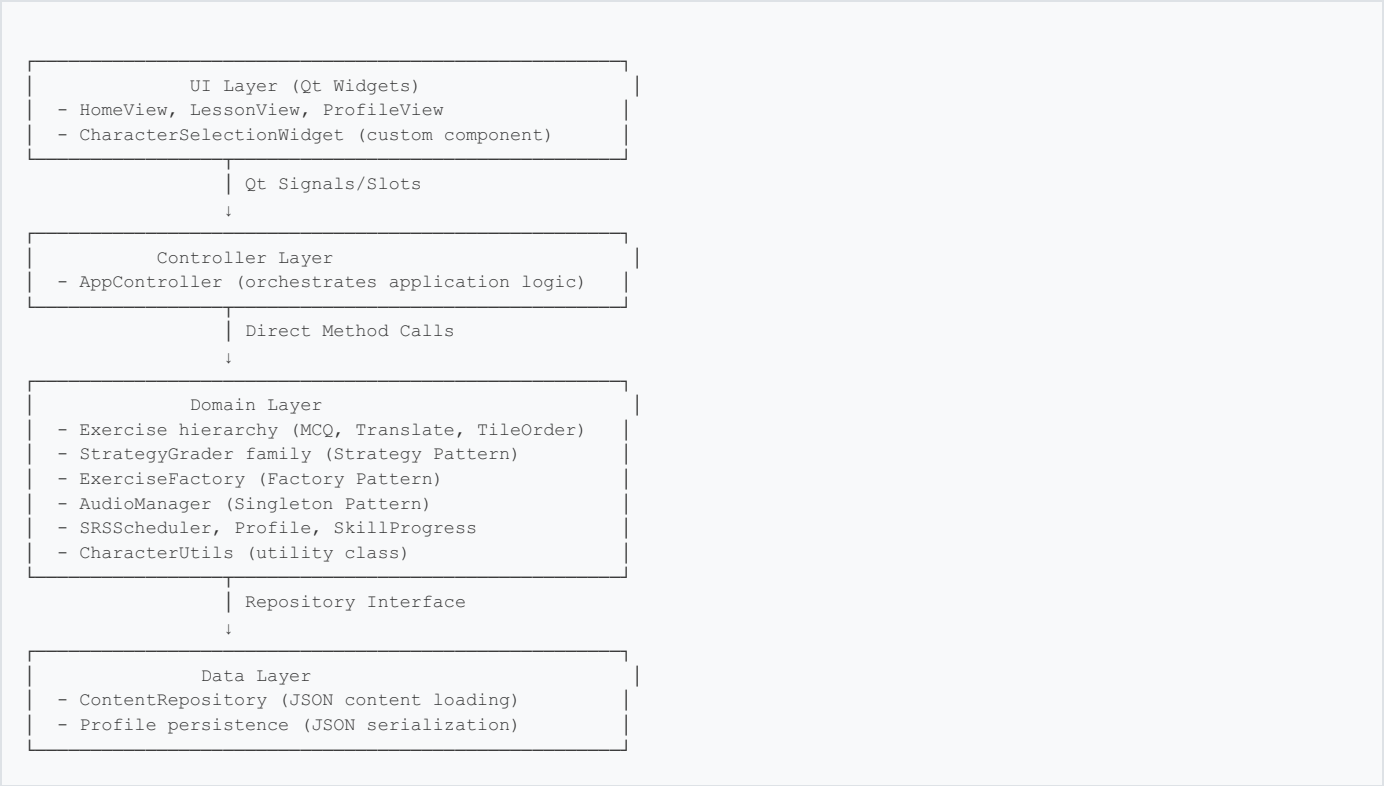
Project: Bhasha Quest - Language Learning Application for Indic Scripts

---

1. Architecture Overview

System Structure

Bhasha Quest employs a **layered MVC (Model-View-Controller) architecture** using Qt 6 and C++17, with strict adherence to the separation of concerns principle. The application is organized into four distinct architectural layers:



Architectural Principles

Strict Layer Separation:
The architecture enforces unidirectional dependency flow from UI → Controller → Domain → Data. The domain layer has zero knowledge of the UI implementation, ensuring business logic remains testable and reusable. UI components communicate exclusively with AppController through Qt's signal/slot mechanism, never directly accessing domain objects.

Qt Signal/Slot Mechanism for Loose Coupling:
The UI layer connects to AppController using Qt's signal/slot system, enabling loose coupling between presentation and control logic. When a user submits an answer, LessonView emits a signal that AppController receives through a slot, processes using domain logic, and emits result signals back to the UI. This decoupling allows UI modifications without touching controller code and vice versa.

Controller as Orchestrator:
AppController serves as the central orchestrator, managing lesson flow, exercise progression, and user profile

updates. It delegates business logic to domain classes (e.g., StrategyGrader for evaluation, SRSScheduler for review intervals) while maintaining session state. This separation ensures that complex workflows remain comprehensible and domain objects stay focused on single responsibilities.

Repository Pattern for Data Abstraction:
ContentRepository provides a clean abstraction over data storage, currently implemented with JSON file loading. The repository interface shields the domain layer from data source details, allowing future migration to SQLite or cloud storage without modifying domain classes. Profile serialization follows the same principle, using JSON as the persistence mechanism with clear boundaries between business objects and their serialized representations.

Key Architectural Benefits

1. **Testability:** Domain logic can be unit-tested in isolation without Qt dependencies
2. **Maintainability:** Changes to UI design don't ripple into business logic
3. **Extensibility:** New exercise types can be added by extending Exercise and creating corresponding graders
4. **Scalability:** Clear layer boundaries enable parallel development on different components

2. Design Patterns

2.1 Factory Pattern: ExerciseFactory

Pattern Type: Creational
Implementation Location: `src/core/domain/ExerciseFactory.h/cpp`

Why This Pattern Was Chosen

The application supports multiple exercise types (MCQExercise, TranslateExercise, TileOrderExercise) that share a common interface but have different creation requirements. Without the Factory pattern, the ContentRepository would need complex conditional logic to instantiate exercises based on type strings from JSON data. This would violate the Open/Closed Principle and create tight coupling between data loading and exercise instantiation.

How It Was Implemented

```
class ExerciseFactory {
public:
    static Exercise* createExercise(const QString& type, const QJsonObject& spec);

private:
    static Exercise* createMCQ(const QJsonObject& spec);
    static Exercise* createTranslate(const QJsonObject& spec);
    static Exercise* createTileOrder(const QJsonObject& spec);
};
```

The factory provides a single static `createExercise()` method that accepts an exercise type string ("MCQ", "Translate", "TileOrder") and a JSON specification. It delegates to private helper methods for type-specific construction, encapsulating the details of how each exercise type is initialized from JSON data.

Usage Example:

```
// In ContentRepository::loadExercises()
QString exerciseType = jsonObj["type"].toString();
QJsonObject spec = jsonObj["spec"].toObject();
Exercise* exercise = ExerciseFactory::createExercise(exerciseType, spec);
exerciseQueue.append(exercise);
```

Benefits Gained

1. **Encapsulation:** Object creation logic is centralized in one class rather than scattered across the codebase
2. **Extensibility:** Adding a new exercise type (e.g., ListenExercise) requires only modifying ExerciseFactory, not ContentRepository or AppController
3. **Maintainability:** Changes to exercise construction parameters are localized to factory methods
4. **Testability:** Factory can be tested independently with mock JSON specifications

2.2 Strategy Pattern: StrategyGrader Hierarchy

Pattern Type: Behavioral
Implementation Location: `src/core/domain/StrategyGrader.h`, `MCQGrader.cpp`, `TranslateGrader.cpp`, `TileOrderGrader.cpp`, `CharacterSelectionGrader.cpp`

Why This Pattern Was Chosen

Different exercise types require fundamentally different grading algorithms:

- **MCQExercise:** Simple index matching (user selected option vs. correct index)
- **TranslateExercise:** Fuzzy string matching with multiple acceptable answers
- **TileOrderExercise:** Sequence order verification
- **Character Selection Mode:** Character-by-character sequence matching

Embedding grading logic directly in Exercise classes would violate the Single Responsibility Principle and make testing difficult. The Strategy pattern allows grading algorithms to be selected and swapped at runtime based on exercise type.

How It Was Implemented

```
// Abstract strategy interface
class StrategyGrader {
public:
    virtual ~StrategyGrader() = default;
    virtual Result grade(const QString& userAnswer, const Exercise* exercise) = 0;
};

// Concrete strategies
class MCQGrader : public StrategyGrader {
    Result grade(const QString& userAnswer, const Exercise* exercise) override;
};

class TranslateGrader : public StrategyGrader {
    Result grade(const QString& userAnswer, const Exercise* exercise) override;
};
```

AppController dynamically selects the appropriate grader based on exercise type:

```
StrategyGrader* AppController::createGraderForExercise(Exercise* exercise) {
    if (dynamic_cast<MCQExercise*>(exercise)) {
        return new MCQGrader();
    } else if (dynamic_cast<TranslateExercise*>(exercise)) {
        if (exercise->usesCharacterSelection()) {
            return new CharacterSelectionGrader();
        }
        return new TranslateGrader();
    } else if (dynamic_cast<TileOrderExercise*>(exercise)) {
        return new TileOrderGrader();
    }
    return nullptr;
}
```

Benefits Gained

> 1. **Flexibility:** Grading algorithms can be modified or replaced without changing Exercise or AppController
> 2. **Extensibility:** New grading strategies can be added by implementing the StrategyGrader interface
> 3. **Testability:** Each grading algorithm can be unit-tested independently with mock exercises
> 4. **Runtime Selection:** The appropriate strategy is chosen dynamically based on exercise characteristics

2.3 Singleton Pattern: AudioManager

Pattern Type: Creational

Implementation Location: `src/core/domain/AudioManager.h/cpp`

Why This Pattern Was Chosen

Audio playback requires expensive Qt resources (QMediaPlayer, QAudioOutput) that should be initialized once and reused throughout the application lifecycle. Multiple AudioManager instances would cause:

- Resource conflicts (multiple players competing for audio device)
- Memory waste (duplicate initialization of media frameworks)
- Inconsistent audio state (multiple players playing simultaneously)

The Singleton pattern ensures exactly one instance manages all audio operations.

How It Was Implemented

```
class AudioManager {
public:
    static AudioManager& getInstance();

    // Delete copy/move constructors and assignment operators
    AudioManager(const AudioManager&) = delete;
    AudioManager& operator=(const AudioManager&) = delete;

    void playSuccess();
    void playError();
    void playAudio(const QString& filename);

private:
    AudioManager();  // Private constructor
    ~AudioManager();

    static AudioManager* instance;
    QMediaPlayer* player;
    QAudioOutput* audioOutput;
};
```

Usage Example:

```
// In AppController::submitAnswer()
if (result.correct) {
    AudioManager::getInstance().playSuccess();
} else {
    AudioManager::getInstance().playError();
}
```

Benefits Gained

1. **Resource Management:** Single QMediaPlayer instance is reused for all audio operations
2. **Global Access Point:** Any component can access audio functionality through getInstance()
3. **Lazy Initialization:** AudioManager is created only when first accessed
4. **Thread Safety:** Singleton ensures no race conditions during initialization (though current implementation is single-threaded)

3. Implementation Challenges

Challenge 1: Qt 6 Installation and Learning Curve

Problem:
As a developer new to Qt, I faced a steep learning curve understanding Qt's build system, signal/slot mechanism, and widget hierarchy. The initial setup required downloading Qt 6.10 (3+ GB), configuring Qt Creator, and resolving path dependencies.

Resolution:
I approached this systematically by:

1. Following Qt's official "Getting Started" documentation for basic project structure
2. Creating a minimal "Hello World" Qt Widgets application to verify installation
3. Incrementally adding complexity (signals/slots, layouts, custom widgets)
4. Using Qt Creator's integrated documentation (F1 key) to understand class APIs
5. Studying the Qt Widgets Gallery example for UI best practices

Lessons Learned:
Qt's meta-object compiler (MOC) requires Q_OBJECT macro for signal/slot functionality. Missing this macro caused cryptic linker errors. I learned to always include Q_OBJECT in classes that define signals or slots.

Challenge 2: qmake vs. CMake Build System Confusion

Problem:
The project specification mentioned Qt Creator but didn't clarify whether to use qmake (.pro files) or CMake (CMakeLists.txt). I initially created both build files, which caused conflicts when Qt Creator tried to determine the project's build system. This resulted in build errors and missing dependencies.

Resolution:
After researching Qt's build systems, I decided to standardize on **qmake** for this project because:

- qmake is Qt's native build system with simpler syntax for Qt-specific features
- .pro files are more concise for projects heavily using Qt modules (Widgets, Multimedia)
- Qt Creator has excellent integration with qmake projects

I removed CMakeLists.txt and maintained only BhashaQuestV3.pro, ensuring all source files, headers, and forms were properly listed.

Lessons Learned:
Build systems must be mutually exclusive. Attempting to support both qmake and CMake simultaneously creates conflicts. For Qt-centric projects, qmake reduces boilerplate, while CMake is preferable for cross-platform projects with non-Qt dependencies.

Challenge 3: Indic Script Input Solution - CharacterSelectionWidget Innovation

Problem:
The original design assumed users would type Devanagari or Kannada characters using system keyboard input methods. However, this approach has significant usability issues:

1. Users may not have Indic keyboard layouts installed

2. Learning a new input method adds friction to the language learning experience
3. Testing on the TA's machine is unpredictable if the required fonts/input methods are missing

Resolution:
I developed a custom **CharacterSelectionWidget** UI component that displays a character bank as clickable buttons. Users construct answers by tapping characters in sequence rather than typing. This innovation:

- Eliminates dependency on system input methods
- Provides a Duolingo-like tactile experience
- Guarantees Indic script rendering (character bank is generated from exercise data)
- Simplifies testing (no keyboard configuration required)

Supporting classes:

- **CharacterUtils:** Generates character banks by extracting unique characters from the correct answer and adding random distractors
- **CharacterSelectionGrader:** Strategy implementation that grades character-by-character sequences

Lessons Learned:
Sometimes the best solution is to bypass complex system dependencies entirely. Rather than fighting with input method configuration, building a custom UI component provided better UX and eliminates deployment concerns.

Challenge 4: Qt Signal/Slot Wiring Complexity

Problem:
The application's architecture requires numerous signal/slot connections between UI components and AppController. Initially, I manually connected each signal using Qt's `connect()` function in the constructor, leading to verbose, error-prone code with over 20 connect statements in LessonView alone.

Resolution:
I adopted a **systematic naming convention** where signals and slots use descriptive names that clearly indicate their purpose:

- Signal: `answerSubmitted(QString)` → Slot: `onAnswerSubmitted(QString)`
- Signal: `lessonCompleted(int, int)` → Slot: `onLessonCompleted(int, int)`

I organized connections into logical groups (lesson lifecycle, answer submission, progress updates) with comments explaining the data flow. Qt Creator's "Go to Slot" feature auto-generates slot declarations, reducing boilerplate.

Lessons Learned:
Clear naming conventions and systematic organization of signal/slot connections dramatically improves code readability. Grouping related connections together makes it easier to trace event flows through the application.

4. Changes from Deliverable 1

Addition 1: CharacterSelectionWidget (Beyond Requirements)

Motivation:
The original proposal planned for a simple text input field for translation exercises. During implementation, I realized this approach would create deployment and usability issues for Indic scripts.

Implementation:
Created a custom Qt widget that displays a shuffled character bank as clickable buttons. Users tap characters in sequence to build their answer. This provides:

- Visual feedback (selected characters highlighted)
- Undo/Clear operations
- Guaranteed Indic script rendering
- Duolingo-like user experience

Impact:
This addition enhances the application beyond the base requirements, demonstrating proactive problem-solving and attention to user experience.

Addition 2: CharacterUtils Helper Class

Motivation:
Generating character banks requires non-trivial string processing:

- Extracting unique Unicode characters from answers
- Shuffling characters randomly
- Adding distractor characters from language-specific sets

Implementation:
Created a static utility class with methods for:

- `extractUniqueCharacters(QString)`: Parse Unicode strings into individual characters
- `generateCharacterBank(QString, QStringList)`: Create shuffled n+3 character banks
- `getRandomDistractors(QString, QStringList, int)`: Select random characters not in the answer

Impact:
Encapsulates character processing logic in a reusable, testable class. This follows the Single Responsibility Principle and keeps TranslateExercise focused on exercise semantics rather than string manipulation.

Addition 3: Enhanced UI with Animations and Modern Styling

Motivation:
The initial prototype had basic, unstyled Qt widgets. To create an engaging learning experience, I enhanced the UI with:

- QPropertyAnimation for smooth transitions (fade effects, slide-ins)
- Custom stylesheets for buttons, labels, and layouts
- QGraphicsEffect for shadows and visual polish

Implementation:
Applied Qt stylesheets (CSS-like syntax) to customize widget appearance:

```
submitButton->setStyleSheet(
    "QPushButton {"
    "  background-color: #4CAF50;"
    "  color: white;"
    "  border-radius: 8px;"
    "  padding: 12px 24px;"
    "  font-size: 16px;"
    "}"
    "QPushButton:hover {"
    "  background-color: #45a049;"
    "}"
);
```

Used QPropertyAnimation for feedback effects:

```
QPropertyAnimation* fadeOut = new QPropertyAnimation(feedbackLabel, "opacity");
fadeOut->setDuration(500);
fadeOut->setStartValue(1.0);
fadeOut->setEndValue(0.0);
fadeOut->start();
```

Impact:
Creates a polished, modern learning experience that rivals commercial applications. Demonstrates mastery of Qt's graphics and animation frameworks beyond basic widget usage.

Addition 4: Time Travel Dev Tools for Testing

Motivation:
Testing spaced repetition scheduling (SRSScheduler) is difficult when review intervals span days or weeks. Manual testing by waiting for real time is impractical during development.

Implementation:
Added debug functions to SRSScheduler that simulate time passage:

```
#ifdef QT_DEBUG
void SRSScheduler::advanceTimeTo(const QDateTime& futureTime) {
    simulatedCurrentTime = futureTime;
}
#endif
```

This allows testing scenarios like "What happens if a user returns after 3 days?" without waiting 3 real days.

Impact:
Dramatically speeds up development and testing. Ensures SRS algorithm correctness without relying on long integration tests. These functions are conditionally compiled (only in debug builds) to avoid deployment concerns.

5. Next Steps for Deliverable 3

Feature 1: ListenExercise with Audio Playback

Description:
Implement a new exercise type where users hear a word/phrase and select the correct translation from options.

Technical Approach:

- Create ListenExercise class extending Exercise base
- Integrate with AudioManager to play exercise audio on demand
- Implement ListenGrader strategy for evaluating user selections
- Update ExerciseFactory to handle "Listen" type
- Add UI controls (play button, repeat button) to LessonView

Estimated Complexity: Moderate (leverages existing AudioManager singleton)

Feature 2: SpeakExercise with Recording + Stub ASR

Description:
Implement a speaking exercise where users record their pronunciation and receive feedback. Due to time constraints, use a **stub Automatic Speech Recognition (ASR)** evaluator rather than integrating a real speech-to-text API.

Technical Approach:

- Create SpeakExercise class extending Exercise base
- Implement SpeakGrader with StubSpeechEvaluator interface:
    - Stub implementation returns random scores for development
    - Interface design allows future integration with real ASR (Google Cloud Speech, Whisper)
- Add recording UI (microphone button, waveform visualization)
- Use Qt Multimedia's audio recording capabilities (QAudioRecorder)

Estimated Complexity: High (involves audio capture, which is more complex than playback)

Feature 3: Observer Pattern for Real-Time Updates

Description:
Implement the Observer pattern to enable real-time UI updates when profile or progress changes without explicit refresh calls.

Technical Approach:

- Create Observable interface for Profile, SkillProgress classes
- Implement Observer interface for UI components
- Profile notifies observers when XP, streak, or mastery levels change
- UI components react to notifications and update displays automatically

Estimated Complexity: Low (Qt's signal/slot mechanism already provides observer-like functionality; this is primarily refactoring existing code to use a more explicit pattern)

Feature 4: GoogleTest/GMock Unit Tests

Description:
Add comprehensive unit tests for core domain classes using GoogleTest framework and Google Mock for

dependency isolation.

Test Coverage Plan:

- **SRSScheduler:** Test review interval calculations, due date logic
- **ExerciseFactory:** Verify correct exercise types are created from JSON specs
- **StrategyGrader Implementations:** Test all grading algorithms with edge cases
- **Profile:** Test XP accumulation, streak calculation, serialization
- **ContentRepository:** Mock file I/O, test JSON parsing error handling

Example Test Structure:

```
TEST(SRSSchedulerTest, CalculatesNextReviewInterval) {
    SRSScheduler scheduler;
    QDateTime startTime = QDateTime::currentDateTime();

    // First review after 1 day for "Easy"
    QDateTime nextReview = scheduler.getNextReview(startTime, Difficulty::Easy);
    EXPECT_EQ(nextReview, startTime.addDays(1));
}

TEST(ExerciseFactoryTest, CreatesMCQFromValidSpec) {
    QJsonObject spec = createMockMCQSpec();
    Exercise* ex = ExerciseFactory::createExercise("MCQ", spec);

    ASSERT_NE(ex, nullptr);
    EXPECT_EQ(ex->getType(), "MCQ");
    delete ex;
}
```

Estimated Complexity: Moderate (GoogleTest integration with qmake requires configuration, but test writing is straightforward)

Feature 5: SQLite Persistence Option

Description:
Add SQLite as an alternative to JSON serialization for profile and progress data, allowing faster queries and better data integrity.

Technical Approach:

- Use Qt SQL module (QSqlDatabase, QSqlQuery)
- Create ProfileRepository interface with two implementations:
    - JSONProfileRepository (current implementation)
    - SQLiteProfileRepository (new implementation)
- Use Repository pattern to allow switching storage backends via configuration
- Design schema for profiles and skill_progress tables

Estimated Complexity: Moderate (SQL schema design and query optimization require careful consideration)

Reflection

> This project has reinforced key lessons in software architecture:
>
> 1. **Design Patterns Aren't Abstract Theory:** The Factory, Strategy, and Singleton patterns solved concrete problems in this application. Understanding when and why to apply patterns is more

valuable than memorizing their UML diagrams.

2. **Architecture Matters from Day One:** Starting with clean layer separation meant that adding CharacterSelectionWidget didn't require refactoring. Poor initial architecture compounds technical debt exponentially.
3. **User Experience Drives Technical Decisions:** The CharacterSelectionWidget innovation arose from considering the end-user's actual environment (missing keyboard layouts, font issues). Technical elegance is meaningless if users struggle to complete basic tasks.
4. **Test-Driven Development Would Have Helped:** In hindsight, writing tests before implementing SRSScheduler and StrategyGrader would have caught edge cases earlier. Deliverable 3 will prioritize testing from the start.
5. **Framework Documentation Is Essential:** Qt's documentation, while comprehensive, requires time investment. Reading API docs upfront saves hours of trial-and-error debugging later.

Looking ahead to Deliverable 3, the foundation is solid. The clean architecture and well-defined patterns make adding new exercise types, implementing observers, and integrating testing frameworks straightforward extensions rather than invasive changes.

End of Document