សាកលវិទ្យាល័យ នរតុន
NORTON UNIVERSITY

Department of Computer Study

🎓 Complete Expert System Integration Guide

Khmer-English Dictionary with AI Intelligence

By: SEK SOCHEAT

Mobile: 017 879 967

📋 Table of Contents

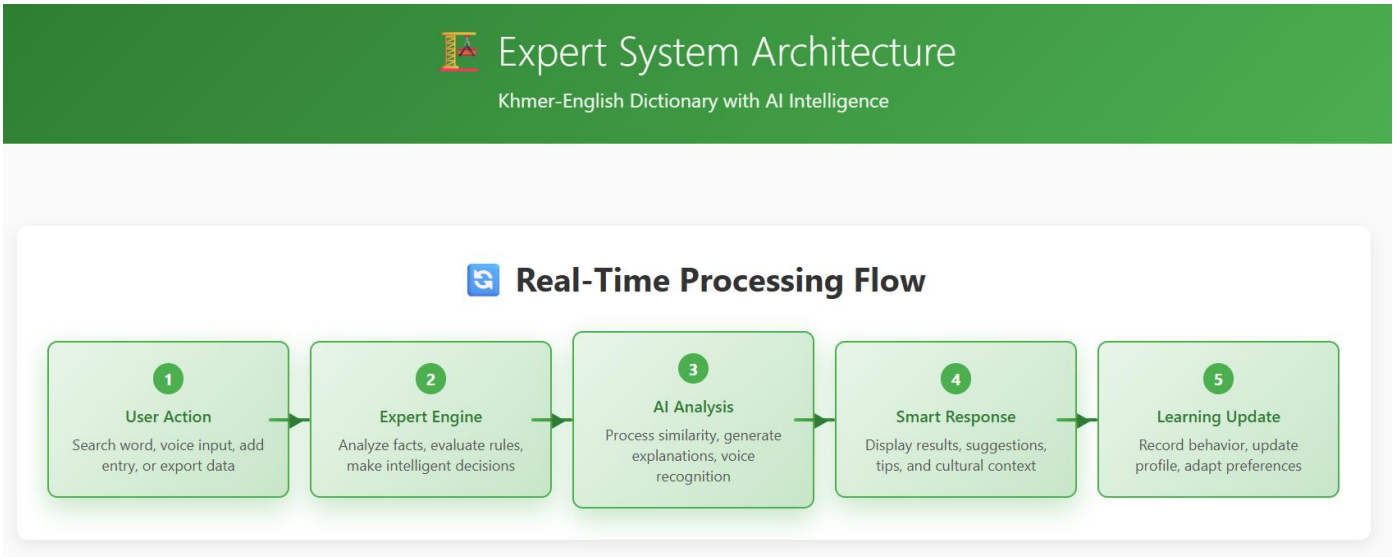
🔍 System Overview

What You're Building

Transform your basic dictionary into an **intelligent learning system** that:

- **Learns from user behavior** and adapts suggestions
- **Provides AI-powered recommendations** based on search patterns
- **Offers voice search** for hands-free operation
- **Generates smart explanations** about grammar and culture
- **Exports comprehensive data** in multiple formats

Architecture Flow



Core Intelligence Components

Component	Purpose	Input	Output
Rule Engine	Make decisions based on conditions	User facts, search data	Smart suggestions
User Profiler	Learn and remember user behavior	Search history, preferences	Personalized recommendations
Word Embeddings	Find similar words using AI	Search terms	Similar word suggestions
AI Explainer	Generate educational content	Word data, grammar rules	Detailed explanations
Data Exporter	Save learning progress	All system data	Multiple file formats

---

📦 Required Dependencies

Essential Packages

Git Bash

```
# Core Requirements (Must Have)
pip install PyQt6          # UI framework
pip install sqlite3        # Database (usually included)
pip install json           # Data handling (usually included)

# AI Features (Recommended)
pip install SpeechRecognition # Voice search
pip install pyaudio        # Audio input

# Export Enhancement (Optional)
pip install pandas         # Better data handling
```

Dependency Verification

```
def check_dependencies():
    """Verify all packages are working"""
    requirements = {
        'PyQt6': 'UI framework for application',
        'sqlite3': 'Database for storing words',
        'json': 'Data format for user profiles',
        'speech_recognition': 'Voice search functionality',
        'csv': 'Export data to spreadsheets'
    }

    for package, description in requirements.items():
        try:
            __import__(package)
```

```
    print(f"☑ {package}: {description}")
except ImportError:
    print(f"✗ {package}: Missing - {description}")
```

🧠 Expert System Components

Component 1: Rule-Based Decision Engine

```
class ExpertSystemEngine:
    """The AI brain that makes smart decisions"""

    def __init__(self):
        self.rules = []        # List of decision rules
        self.facts = {}        # Current situation data
        self.initialize_rules() # Load smart rules

    def add_fact(self, key, value):
        """Tell the system what's happening"""
        self.facts[key] = value

    def infer(self):
        """Make smart suggestions based on facts"""
        suggestions = []
        for rule in self.rules:
            if rule.evaluate(self.facts):
                suggestions.extend(rule.fire(self.facts))
        return suggestions
```

Example Rule:

```
# If user searches for noun but nothing found → suggest related objects
Rule(
    name="noun_help",
    conditions=["word_type == 'noun'", "not_found == True"],
    actions=["Try searching for objects, places, or things",
```

```
            "Check spelling of physical items"]
)
```

Component 2: User Behavior Learning System

```
class UserProfileManager:
    """Remembers and learns from user actions"""

    def __init__(self):
        self.profile = {
            "difficulty_level": "beginner",     # User skill level
            "search_history": [],               # What they searched
            "word_preferences": {},             # What they like
            "favorite_word_types": {},          # Noun, verb, etc.
            "learning_progress": {}             # How they improve
        }

    def record_search(self, term, search_type, found):
        """Remember what user searched for"""
        # Store search and learn patterns

    def get_recommendations(self):
        """Suggest what user should learn next"""
        # Analyze patterns and suggest improvements
```

Component 3: Smart Word Similarity Finder

```
class SimpleWordEmbeddings:
    """Finds similar words using AI math"""

    def add_word(self, word, context):
```

```
        """Add word to AI knowledge"""
        features = self._extract_features(word, context)
        self.embeddings[word] = features

    def find_similar(self, word, n=5):
        """Find words that are similar"""
        # Use math to find similar words
        # Returns: [("computer", 0.85), ("computing", 0.78)]
```

Component 4: AI Grammar Teacher

```
class AIExplanationGenerator:
    """Generates helpful explanations about words"""

    def generate_explanation(self, word_data):
        """Create detailed explanation for a word"""
        # Analyzes: grammar rules, cultural context, usage tips
        # Returns: HTML explanation with examples
```

Component 5: Comprehensive Data Exporter

```
class DataExporter:
    """Saves all data in multiple formats"""

    def export_data(self, formats, data_types, options, filename):
        """Export learning data"""
        # Formats: CSV, JSON, TXT, HTML
        # Data: Dictionary, Analytics, Profile, History
        # Returns: List of created files
```

🔧 Step-by-Step Integration

Step 1: Prepare Your Code Structure

```
# Add these imports to your existing file
import json
import csv
from datetime import datetime
from collections import defaultdict, Counter
from pathlib import Path
from PyQt6.QtWidgets import QFileDialog

# Add voice search imports (optional)
try:
    import speech_recognition as sr
    SPEECH_RECOGNITION_AVAILABLE = True
except ImportError:
    SPEECH_RECOGNITION_AVAILABLE = False
```

Step 2: Add Expert System Classes

```
# Add all expert system classes to your main file:
# - Rule (decision making)
# - ExpertSystemEngine (AI brain)
# - UserProfileManager (learning system)
# - SimpleWordEmbeddings (word similarity)
# - AIExplanationGenerator (grammar teacher)
# - VoiceSearchThread (voice input)
# - DataExporter (file export)
# - ExportDialog (export options)
```

Step 3: Enhance Your Database

```
# Replace your DictionaryDatabase with EnhancedDictionaryDatabase
class EnhancedDictionaryDatabase(DictionaryDatabase):
    def __init__(self):
        super().__init__()
        self.embeddings = SimpleWordEmbeddings()  # AI word finder
        self._build_embeddings()  # Learn from existing words

    def create_word(self, english, khmer, word_type, definition, example,
                    difficulty="beginner", cultural_tags="", grammar_notes=""):
        """Enhanced word creation with AI features"""
        # Original creation + AI learning
        word_id = super().create_word(...)
        self.embeddings.add_word(english, definition)
        return word_id

    def find_similar_words(self, word, n=5):
        """NEW: Find similar words using AI"""
        return self.embeddings.find_similar(word, n)

    def get_smart_suggestions(self, search_term, search_type):
        """NEW: Get AI suggestions for failed searches"""
        similar_words = self.find_similar_words(search_term, 3)
        # Return actual word data for similar words
```

Step 4: Upgrade Your Translator Widget

```
class TranslatorWidget(QWidget):
    def __init__(self, db, font_manager, expert_engine, user_profile, ai_generator):
        super().__init__()
        # Add expert system components
```

```
        self.expert_engine = expert_engine
        self.user_profile = user_profile
        self.ai_generator = ai_generator
        self.init_ui()

    def init_ui(self):
        # Original UI code...

        # ADD: Voice button between search input and search button
        self.voice_button = QPushButton("🎤")
        self.voice_button.clicked.connect(self.start_voice_search)

        # Layout: [Input] [🎤] [Search]
        controls_layout.addWidget(self.search_input)
        controls_layout.addWidget(self.voice_button)  # NEW
        controls_layout.addWidget(self.search_button)

    def search_word(self):
        """Enhanced search with AI features"""
        search_term = self.search_input.text().strip()
        results = self.db.read_word(search_term, search_type)
        found = len(results) > 0

        # NEW: Record user behavior
        self.user_profile.record_search(search_term, search_type, found)

        # NEW: Add facts to expert system
        self.expert_engine.add_fact("search_term", search_term)
        self.expert_engine.add_fact("not_found", not found)
```

```
        # NEW: Get AI suggestions if not found
        if not found:
            suggestions = self.db.get_smart_suggestions(search_term, search_type)
            self.display_no_results_with_suggestions(search_term, suggestions)

        # NEW: Get expert recommendations
        expert_advice = self.expert_engine.infer()
        self.display_expert_suggestions(expert_advice)
```

Step 5: Add Voice Search Functionality

```
def start_voice_search(self):
    """Start voice recognition"""
    if not SPEECH_RECOGNITION_AVAILABLE:
        # Show installation message
        return

    self.voice_thread = VoiceSearchThread()
    self.voice_thread.speech_recognized.connect(self.on_voice_search_completed)
    self.voice_thread.start()

def on_voice_search_completed(self, recognized_text):
    """Handle completed voice search"""
    self.search_input.setText(recognized_text)
    self.search_word()  # Automatically search
```

Step 6: Enhance Dictionary Manager

```
class DictionaryManagerWidget(QWidget):
    def __init__(self, db, font_manager, ai_generator):
```

```
        self.ai_generator = ai_generator  # NEW: AI helper
        # Original init code...

    def view_selected_word(self):
        """Enhanced word details with AI explanation"""
        word_data = self.get_selected_word()
        if word_data:
            # NEW: Show AI-enhanced dialog
            dialog = WordDetailsDialog(word_data, self.font_manager,
                                       self.ai_generator, self)
            dialog.exec()

    def ai_assist(self):
        """NEW: AI assistance for filling forms"""
        english_word = self.english_input.text().strip()
        if english_word:
            # Provide smart suggestions for word type, difficulty, etc.
```

Step 7: Add Export Functionality

```
class StatisticsWidget(QWidget):
    def export_data(self):
        """NEW: Comprehensive data export"""
        # Show export options dialog
        dialog = ExportDialog(self.font_manager, self)
        if dialog.exec() == QDialog.DialogCode.Accepted:
            formats, data_types, options = dialog.get_export_settings()

            # Choose save location
            filename, _ = QFileDialog.getSaveFileName(self, "Export Data")
```

```
        # Export data
        exporter = DataExporter(self.db, self.user_profile, self.font_manager)
        results = exporter.export_data(formats, data_types, options, filename)

        # Show results
        self.show_export_results(results)
```

Step 8: Update Main Application

```
class KhmerEnglishDictionaryApp(QMainWindow):
    def __init__(self):
        super().__init__()

        # NEW: Initialize expert system components
        self.db = EnhancedDictionaryDatabase()          # Smart database
        self.expert_engine = ExpertSystemEngine()       # AI brain
        self.user_profile = UserProfileManager()        # Learning system
        self.ai_generator = AIExplanationGenerator()    # Grammar teacher

        # Create enhanced widgets with AI features
        self.translator_tab = TranslatorWidget(
            self.db, self.font_manager, self.expert_engine,
            self.user_profile, self.ai_generator
        )

        self.manager_tab = DictionaryManagerWidget(
            self.db, self.font_manager, self.ai_generator
        )
```

```
self.stats_tab = StatisticsWidget(
    self.db, self.font_manager, self.user_profile
)
```

🎯 Key Features Explanation

Feature 1: Intelligent Search Suggestions 🔍

What it does: When exact word not found, suggests similar words using AI

Implementation:

```
# In search_word() method
if not results:
    similar_words = self.db.find_similar_words(search_term, 3)
    suggestions = []
    for word, similarity in similar_words:
        if similarity > 0.3:  # 30% similarity threshold
            word_data = self.db.read_word(word, "english")
            suggestions.extend(word_data)
    self.display_suggestions(suggestions)
```

User Experience:

- User types "computr" (misspelled)
- System shows: "Did you mean: computer, computing, compute?"
- User clicks suggestion and finds the word

Feature 2: Adaptive User Learning 📈

What it does: Learns user preferences and adapts difficulty level

Implementation:

```
# Track every search
self.user_profile.record_search(term, search_type, found)

# Track word interactions
self.user_profile.record_word_interaction(word_type, difficulty)

# Get personalized recommendations
```

recommendations = self.user_profile.get_recommendations()

User Experience:

- User searches many "noun" words → System suggests noun-related features
- User searches complex words → System adapts to "intermediate" level
- System remembers user's language preference (English/Khmer focus)

Feature 3: Expert Rule System 🧠

What it does: Provides context-aware help based on user situation

Implementation:

```
# Add facts about current situation
self.expert_engine.add_fact("word_type", "adjective")
self.expert_engine.add_fact("not_found", True)
self.expert_engine.add_fact("user_level", "beginner")

# Get smart advice
suggestions = self.expert_engine.infer()
# Returns: ["Adjectives in Khmer follow nouns", "Try simple descriptive words"]
```

User Experience:

- User searches for adjective but fails → System explains adjective grammar
- Beginner user gets simpler suggestions than advanced user
- System adapts advice based on search patterns

Feature 4: Voice Search Integration 🎤

What it does: Convert speech to text for hands-free searching

Implementation:

```
# Voice button triggers speech recognition
def start_voice_search(self):
    self.voice_thread = VoiceSearchThread()
    self.voice_thread.speech_recognized.connect(self.on_voice_completed)
```

```
    self.voice_thread.start()


def on_voice_completed(self, text):
    self.search_input.setText(text)
    self.search_word()  # Auto-search
```

User Experience:

- Click 🎤 button
- Speak: "hello"
- System automatically searches for "hello"
- Shows results without typing

Feature 5: AI Grammar Explanations 📚

What it does: Generates detailed explanations about word usage

Implementation:

```
def generate_explanation(self, word_data):
    explanation = f"<h3>Expert Analysis: {english} → {khmer}</h3>"

    # Add grammar rules
    if word_type in self.grammar_rules:
        explanation += f"<h4>Grammar:</h4>{grammar_info}"

    # Add cultural context
    cultural_notes = self._get_cultural_notes(english, khmer)
    explanation += f"<h4>Culture:</h4>{cultural_notes}"

    # Add usage tips
    tips = self._generate_usage_tips(word_type)
    explanation += f"<h4>Tips:</h4>{tips}"
```

return explanation

User Experience:

- Click "View Details" on any word
- See comprehensive explanation with:
    - Grammar rules for that word type
    - Cultural context and appropriate usage
    - Practical tips for remembering the word

Feature 6: Comprehensive Data Export 📊

What it does: Export all learning data in multiple formats

Implementation:

```
# Export options
formats = ["csv", "json", "txt", "html"]
data_types = ["dictionary", "analytics", "profile", "history"]

# Export process
exporter = DataExporter(db, user_profile, font_manager)
results = exporter.export_data(formats, data_types, options, filename)

# Generated files:
# - dictionary_20241221.csv (all words)
# - analytics_20241221.json (learning stats)
# - profile_20241221.txt (user preferences)
```

User Experience:

- Choose what to export (words, progress, preferences)
- Choose format (spreadsheet, web page, text file)
- Get multiple files with all learning data
- Use files for backup, analysis, or sharing

🧪 Testing and Validation

Test 1: Expert System Intelligence

```
def test_expert_system():
    """Test if AI brain makes smart decisions"""

    # Create expert engine
    engine = ExpertSystemEngine()

    # Add facts about user situation
    engine.add_fact("word_type", "noun")
    engine.add_fact("not_found", True)
    engine.add_fact("search_count", 5)
    engine.add_fact("user_level", "beginner")

    # Get AI suggestions
    suggestions = engine.infer()

    # Validate intelligence
    assert len(suggestions) > 0, "Expert system should give suggestions"
    assert any("noun" in s.lower() for s in suggestions), "Should mention nouns"

    print(f"☑ Expert system generated {len(suggestions)} smart suggestions")
    return True
```

Test 2: User Learning System

```
def test_user_learning():
    """Test if system learns from user behavior"""
```

```
    # Create user profile
    profile = UserProfileManager()

    # Simulate user behavior
    profile.record_search("hello", "english", True)
    profile.record_search("computer", "english", True)
    profile.record_search("difficult_word", "english", False)
    profile.record_word_interaction("noun", "medium")

    # Check learning
    user_facts = profile.get_user_facts()
    recommendations = profile.get_recommendations()

    # Validate learning
    assert user_facts["search_count"] >= 3, "Should count searches"
    assert len(recommendations) >= 0, "Should generate recommendations"

    print(f"☑ User learning: {user_facts['search_count']} searches tracked")
    return True
```

Test 3: Smart Database Features

```
def test_smart_database():
    """Test AI-enhanced database functions"""

    # Create enhanced database
    db = EnhancedDictionaryDatabase()

    # Test smart word creation
    word_id = db.create_word(
```

```
        english_word="testing",
        khmer_word="ការសាកល្បង",
        word_type="noun",
        definition="Process of checking something works",
        example="Testing is important for quality",
        difficulty="intermediate",
        cultural_tags="technology,education",
        grammar_notes="Modern technical term"
    )

    # Test AI similarity search
    similar_words = db.find_similar_words("testing", 3)
    smart_suggestions = db.get_smart_suggestions("testin", "english")

    # Validate AI features
    assert word_id is not None, "Should create enhanced word"
    assert len(similar_words) >= 0, "Should find similar words"
    assert len(smart_suggestions) >= 0, "Should provide suggestions"

    print(f"☑ Smart database: {len(similar_words)} similar words found")
    return True
```

Test 4: Voice Search System

```
def test_voice_search():
    """Test voice recognition setup"""

    # Check if voice available
    if not SPEECH_RECOGNITION_AVAILABLE:
```

```
        print("⚠️ Voice search not available (install SpeechRecognition)")
        return True  # Not required feature

    # Test voice thread creation
    voice_thread = VoiceSearchThread()

    # Check initialization
    has_recognizer = hasattr(voice_thread, 'recognizer')
    has_microphone = hasattr(voice_thread, 'microphone')
    has_signals = hasattr(voice_thread, 'speech_recognized')

    # Validate voice setup
    assert has_recognizer, "Should have speech recognizer"
    assert has_microphone, "Should have microphone access"
    assert has_signals, "Should have communication signals"

    print("☑ Voice search system properly initialized")
    return True
```

Test 5: AI Explanations

```
def test_ai_explanations():
    """Test AI grammar and culture teacher"""

    # Create AI explanation generator
    ai_generator = AIExplanationGenerator()

    # Test with sample word data
    sample_word = (
```

```
    1, "hello", "ស្វស្តី", "greeting",
    "Common greeting", "Hello friend",
    "beginner", 10, "social,polite", "Casual greeting"
)

# Generate explanation
explanation = ai_generator.generate_explanation(sample_word)

# Check explanation quality
required_sections = ["Expert Analysis", "Grammar", "Usage Tips"]
sections_found = sum(1 for section in required_sections if section in explanation)

# Validate AI teaching
assert len(explanation) > 100, "Should generate substantial explanation"
assert sections_found >= 2, "Should include multiple teaching sections"

print(f"☑ AI explanation: {len(explanation)} chars, {sections_found} sections")
return True
```

Test 6: Data Export System

```
def test_data_export():
    """Test comprehensive data export"""

    # Create export system
    db = EnhancedDictionaryDatabase()
    user_profile = UserProfileManager()
    font_manager = FontManager()
    exporter = DataExporter(db, user_profile, font_manager)
```

```
    # Add test data
    user_profile.record_search("export_test", "english", True)

    # Test export
    results = exporter.export_data(
        formats=["csv", "json"],
        data_types=["dictionary", "analytics"],
        options={"include_timestamps": True},
        base_filename="test_export"
    )

    # Check export success
    successful_exports = sum(1 for _, success in results if success)

    # Validate export
    assert len(results) > 0, "Should attempt exports"
    assert successful_exports > 0, "Should successfully export some data"

    print(f"☑ Data export: {successful_exports}/{len(results)} files created")
    return True
```

Test 7: Complete Integration

```
def test_complete_integration():
    """Test all components working together"""

    print("=== COMPLETE INTEGRATION TEST ===")

    # Initialize all components
```

```
db = EnhancedDictionaryDatabase()
expert_engine = ExpertSystemEngine()
user_profile = UserProfileManager()
ai_generator = AIExplanationGenerator()

# Test data flow: Add word → Search → Learn → Suggest

# 1. Add enhanced word
word_id = db.create_word("integration", "ការរួមបញ្ចូល", "noun",
                         "Combining parts", "System integration")

# 2. Search for word
results = db.read_word("integration", "english")
found = len(results) > 0

# 3. Record user behavior
user_profile.record_search("integration", "english", found)
user_profile.record_word_interaction("noun", "medium")

# 4. Get expert suggestions
expert_engine.add_fact("search_term", "integration")
expert_engine.add_fact("word_type", "noun")
user_facts = user_profile.get_user_facts()
for key, value in user_facts.items():
    expert_engine.add_fact(key, value)
suggestions = expert_engine.infer()

# 5. Generate AI explanation
```

```
    explanation = ai_generator.generate_explanation(results[0]) if results else ""

    # 6. Test export
    exporter = DataExporter(db, user_profile, FontManager())
    export_results = exporter.export_data(["json"], ["dictionary"], {}, "integration_test")

    # Validate complete workflow
    tests = [
        word_id is not None,          # Database working
        found,                        # Search working
        len(suggestions) > 0,         # Expert system working
        len(explanation) > 0,         # AI explanation working
        any(success for _, success in export_results),  # Export working
        len(user_facts) > 0           # User profiling working
    ]

    success_rate = sum(tests) / len(tests)

    if success_rate >= 0.8:
        print(f"☑ INTEGRATION SUCCESSFUL: {success_rate:.1%}")
        return True
    else:
        print(f"❌ INTEGRATION ISSUES: {success_rate:.1%}")
        return False
```

Master Test Runner

```
def run_all_tests():
    """Run complete test suite"""
```

```
tests = [
    ("Expert System", test_expert_system),
    ("User Learning", test_user_learning),
    ("Smart Database", test_smart_database),
    ("Voice Search", test_voice_search),
    ("AI Explanations", test_ai_explanations),
    ("Data Export", test_data_export),
    ("Complete Integration", test_complete_integration)
]

results = []
print("🧪 RUNNING EXPERT SYSTEM TEST SUITE")
print("=" * 50)

for test_name, test_function in tests:
    print(f"\n🔬 Testing: {test_name}")
    try:
        success = test_function()
        results.append((test_name, success))
        print(f"{'☑ PASS' if success else '❌ FAIL'}: {test_name}")
    except Exception as e:
        results.append((test_name, False))
        print(f"💥 ERROR: {test_name} - {e}")

# Final report
passed = sum(1 for _, success in results if success)
total = len(results)
success_rate = passed / total
```

```
    print("\n" + "=" * 50)
    print(f"📊 FINAL RESULTS: {passed}/{total} tests passed ({success_rate:.1%})")

    if success_rate >= 0.8:
        print("🎉 EXPERT SYSTEM READY FOR DEPLOYMENT!")
    else:
        print("⚠️ EXPERT SYSTEM NEEDS MORE WORK")

    return success_rate >= 0.8

# Run the complete test suite
if __name__ == "__main__":
    run_all_tests()
```

📋 Student Workflow Checklist

📊 Complete Integration Timeline

Phase	Task	Description	Time Est.	Dependencies	Success Criteria	Status
🔧 Phase 1: Setup					**3 hours**	
1.1	Install Dependencies	Get all packages working	45 min	Python 3.8+	All imports successful	☐
1.2	Study Expert Components	Understand each AI class	60 min	Code reading	Can explain each component	☐
1.3	Analyze Test Algorithms	Read through all test functions	45 min	Understanding logic	Know what each test checks	☐
1.4	Prepare Code Structure	Organize files and imports	30 min	File management	Clean code structure	☐
🚀 Phase 2: Core Integration					**4 hours**	
2.1	Add Expert Classes	Copy all AI system classes	30 min	Code organization	Classes compile without errors	☐
2.2	Enhance Database	Replace with EnhancedDictionaryDatabase	45 min	Database knowledge	Smart database features work	☐
2.3	Upgrade Translator	Add AI features to search widget	60 min	UI modification	Expert suggestions appear	☐
2.4	Add Voice Search	Integrate speech recognition	45 min	Threading, audio	Voice button responds	☐
2.5	Enhance Manager	Add AI assistance to CRUD operations	30 min	Form handling	AI assist button works	☐
2.6	Add Export System	Integrate comprehensive data export	60 min	File I/O	Export dialog opens	☐

2.7	Update Main App	Connect all expert components	30 min	Integration	Application launches	☐
🧪 **Phase 3: Testing**					**2.5 hours**	
3.1	Test Expert Engine	Run expert system algorithm	15 min	Algorithm 1 complete	Expert suggestions generated	☐
3.2	Test User Learning	Run user profiling algorithm	15 min	Algorithm 2 complete	User behavior tracked	☐
3.3	Test Smart Database	Run enhanced database algorithm	20 min	Algorithm 3 complete	AI database features work	☐
3.4	Test Voice Search	Run voice recognition algorithm	15 min	Algorithm 4 complete	Voice system initializes	☐
3.5	Test AI Explanations	Run explanation generator algorithm	15 min	Algorithm 5 complete	AI generates explanations	☐
3.6	Test Data Export	Run export system algorithm	20 min	Algorithm 6 complete	Files export successfully	☐
3.7	Test Complete Integration	Run full system algorithm	30 min	All algorithms pass	80%+ success rate	☐
3.8	Debug and Fix	Resolve any test failures	45 min	Test results	All critical tests pass	☐
📈 **Phase 4: Validation**					**2 hours**	
4.1	Performance Testing	Ensure smooth operation	30 min	Working application	No crashes or slowdowns	☐
4.2	User Experience Test	Have others try the application	45 min	Complete features	Users can use AI features	☐
4.3	Feature Validation	Verify all expert features work	30 min	Feature checklist	All AI features functional	☐
4.4	Documentation	Write user guide for AI features	15 min	Understanding	Clear usage instructions	☐

📈 Progress Tracking Dashboard

Metric	Target	Current Progress	Status
Dependencies Installed	100%	___%	☐
Expert Classes Added	7/7	___/7	☐
UI Components Enhanced	3/3	___/3	☐
Test Algorithms Passing	80%+	___%	☐
AI Features Working	6/6	___/6	☐
User Experience Quality	Good	___	☐

🎯 Daily Achievement Goals

Day	Focus Area	Key Milestones	Validation Method
Day 1	Setup + Basic Integration	Dependencies installed, expert classes added	Run basic import tests
Day 2	Core AI Features	Database enhanced, search upgraded	Run algorithms 1-3
Day 3	Advanced Features	Voice search, export system added	Run algorithms 4-6
Day 4	Testing + Polish	All tests passing, user experience refined	Run complete test suite

🔍 Quality Checkpoints

Checkpoint	Validation Questions	Pass Criteria
After Phase 1	Can you import all expert classes without errors?	All imports successful
After Phase 2	Does the application launch with AI features visible?	Application runs, AI buttons appear

After Phase 3	Do test algorithms show 80%+ success rate?	Most tests passing
After Phase 4	Can a new user benefit from AI features?	Positive user feedback

⚠️ Common Issues and Solutions

Issue	Symptoms	Solution	Prevention
Import Errors	ModuleNotFoundError	Install missing packages	Run dependency check first
Voice Search Fails	Button disabled, no recognition	Install SpeechRecognition + pyaudio	Check optional dependencies
Expert System Silent	No suggestions appear	Check rule initialization	Run expert system test
Export Crashes	Error during file creation	Check file permissions	Test with simple export first
Database Errors	Enhanced features don't work	Verify database migration	Compare with original database

🎉 Success Indicators

Your expert system integration is successful when:

- ☑ **Expert suggestions appear** when searching for words
- ☑ **User behavior is tracked** and preferences remembered
- ☑ **Voice search responds** to button clicks (even if no microphone)
- ☑ **AI explanations generate** detailed grammar and cultural information
- ☑ **Data export creates files** in multiple formats
- ☑ **Application feels smarter** than the original dictionary
- ☑ **80%+ of test algorithms pass** without critical errors
- ☑ **New users can discover and use** AI features intuitively

🎓 Summary

This guide transforms your basic Khmer-English dictionary into an **intelligent learning system** that:

1. **Learns from user behavior** and provides personalized suggestions
2. **Uses AI to find similar words** when exact matches fail
3. **Generates smart explanations** about grammar and cultural context
4. **Supports voice search** for hands-free operation
5. **Exports comprehensive data** for analysis and backup
6. **Adapts difficulty level** based on user progress

Integration involves:

- Adding 7 expert system classes to your code
- Enhancing 3 main UI widgets with AI features
- Running 7 test algorithms to verify functionality
- Following a 4-phase workflow over 4 days

Success means:

- Your dictionary becomes a smart tutor that helps users learn
- Users get better suggestions and explanations
- The system remembers and adapts to user preferences
- Advanced features like voice search and data export work seamlessly

Complete Coding:

```
import sys
import sqlite3
import json
import re
import math
from datetime import datetime, timedelta
from collections import defaultdict, Counter
from typing import List, Dict, Tuple, Optional
import pickle
import os
import threading
import time
import csv
from pathlib import Path

from PyQt6.QtWidgets import (QApplication, QMainWindow, QVBoxLayout, QHBoxLayout,
                             QWidget, QLabel, QLineEdit, QPushButton, QTextEdit,
                             QTableView, QMessageBox, QTabWidget, QGroupBox,
                             QFormLayout, QComboBox, QHeaderView, QFrame,
                             QScrollArea, QSplitter, QAbstractItemView, QDialog,
                             QDialogButtonBox, QTextBrowser, QProgressBar,
                             QSlider, QCheckBox, QSpinBox, QFileDialog)
from PyQt6.QtCore import Qt, pyqtSignal, QTimer, QAbstractTableModel, QModelIndex, QVariant, QThread, pyqtSlot
from PyQt6.QtGui import QFont, QFontDatabase, QAction

# Try to import speech recognition
try:
    import speech_recognition as sr
    SPEECH_RECOGNITION_AVAILABLE = True
except ImportError:
    SPEECH_RECOGNITION_AVAILABLE = False
    print("Speech recognition not available. Install with: pip install SpeechRecognition pyaudio")

# Export System Components
class ExportDialog(QDialog):
    """Dialog for choosing export options"""
```

```
37
38      def __init__(self, font_manager, parent=None):
39          super().__init__(parent)
40          self.font_manager = font_manager
41          self.export_format = "CSV"
42          self.export_data_type = "Dictionary"
43          self.include_analytics = True
44          self.init_ui()
45
46      def init_ui(self):
47          self.setWindowTitle("Export Data Options")
48          self.setModal(True)
49          self.resize(600, 350)
50          self.font_manager.apply_font(self)
51
52          layout = QVBoxLayout()
53
54          # Export format selection
55          format_group = QGroupBox("Export Format")
56          self.font_manager.apply_font(format_group, bold=True)
57          format_layout = QVBoxLayout()
58
59          self.format_csv = QCheckBox("CSV (Comma Separated Values)")
60          self.format_json = QCheckBox("JSON (JavaScript Object Notation)")
61          self.format_txt = QCheckBox("TXT (Plain Text)")
62          self.format_html = QCheckBox("HTML (Web Page)")
63
64          self.font_manager.apply_font(self.format_csv)
65          self.font_manager.apply_font(self.format_json)
66          self.font_manager.apply_font(self.format_txt)
67          self.font_manager.apply_font(self.format_html)
68
69          # Set default
70          self.format_csv.setChecked(True)
71
```

```
72          format_layout.addWidget(self.format_csv)
73          format_layout.addWidget(self.format_json)
74          format_layout.addWidget(self.format_txt)
75          format_layout.addWidget(self.format_html)
76          format_group.setLayout(format_layout)
77
78          # Data type selection
79          data_group = QGroupBox("Data to Export")
80          self.font_manager.apply_font(data_group, bold=True)
81          data_layout = QVBoxLayout()
82
83          self.data_dictionary = QCheckBox("Dictionary Entries")
84          self.data_analytics = QCheckBox("Learning Analytics")
85          self.data_profile = QCheckBox("User Profile")
86          self.data_history = QCheckBox("Search History")
87
88          self.font_manager.apply_font(self.data_dictionary)
89          self.font_manager.apply_font(self.data_analytics)
90          self.font_manager.apply_font(self.data_profile)
91          self.font_manager.apply_font(self.data_history)
92
93          # Set defaults
94          self.data_dictionary.setChecked(True)
95          self.data_analytics.setChecked(True)
96
97          data_layout.addWidget(self.data_dictionary)
98          data_layout.addWidget(self.data_analytics)
99          data_layout.addWidget(self.data_profile)
100         data_layout.addWidget(self.data_history)
101         data_group.setLayout(data_layout)
102
```

```
        # Options
        options_group = QGroupBox("Export Options")
        self.font_manager.apply_font(options_group, bold=True)
        options_layout = QVBoxLayout()

        self.include_timestamps = QCheckBox("Include Timestamps")
        self.include_metadata = QCheckBox("Include Metadata")
        self.compress_output = QCheckBox("Compress Large Files")

        self.font_manager.apply_font(self.include_timestamps)
        self.font_manager.apply_font(self.include_metadata)
        self.font_manager.apply_font(self.compress_output)

        self.include_timestamps.setChecked(True)
        self.include_metadata.setChecked(True)

        options_layout.addWidget(self.include_timestamps)
        options_layout.addWidget(self.include_metadata)
        options_layout.addWidget(self.compress_output)
        options_group.setLayout(options_layout)

        # Buttons
        button_box = QDialogButtonBox(
            QDialogButtonBox.StandardButton.Ok | QDialogButtonBox.StandardButton.Cancel
        )
        self.font_manager.apply_font(button_box)
        button_box.accepted.connect(self.accept)
        button_box.rejected.connect(self.reject)

        layout.addWidget(format_group)
        layout.addWidget(data_group)
        layout.addWidget(options_group)
        layout.addWidget(button_box)

```

```
        self.setLayout(layout)

    def get_export_settings(self):
        """Get selected export settings"""
        formats = []
        if self.format_csv.isChecked():
            formats.append("csv")
        if self.format_json.isChecked():
            formats.append("json")
        if self.format_txt.isChecked():
            formats.append("txt")
        if self.format_html.isChecked():
            formats.append("html")

        data_types = []
        if self.data_dictionary.isChecked():
            data_types.append("dictionary")
        if self.data_analytics.isChecked():
            data_types.append("analytics")
        if self.data_profile.isChecked():
            data_types.append("profile")
        if self.data_history.isChecked():
            data_types.append("history")

        options = {
            "include_timestamps": self.include_timestamps.isChecked(),
            "include_metadata": self.include_metadata.isChecked(),
            "compress_output": self.compress_output.isChecked()
        }

        return formats, data_types, options

```

```python
class DataExporter:
    """Handle data export in various formats"""

    def __init__(self, db, user_profile, font_manager):
        self.db = db
        self.user_profile = user_profile
        self.font_manager = font_manager

    def export_data(self, formats, data_types, options, base_filename):
        """Export data in specified formats"""
        results = []
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

        try:
            for fmt in formats:
                for data_type in data_types:
                    filename = f"{base_filename}_{data_type}_{timestamp}.{fmt}"

                    if fmt == "csv":
                        success = self._export_csv(data_type, filename, options)
                    elif fmt == "json":
                        success = self._export_json(data_type, filename, options)
                    elif fmt == "txt":
                        success = self._export_txt(data_type, filename, options)
                    elif fmt == "html":
                        success = self._export_html(data_type, filename, options)
                    else:
                        success = False

                    results.append((filename, success))

            return results

        except Exception as e:
            print(f"Export error: {e}")
            return [(f"Export failed: {str(e)}", False)]
```

```
205
206     def _export_csv(self, data_type, filename, options):
207         """Export data as CSV"""
208         try:
209             with open(filename, 'w', newline='', encoding='utf-8') as csv_file:
210                 if data_type == "dictionary":
211                     writer = csv.writer(csv_file)
212
213                     # Header
214                     headers = ["ID", "English", "Khmer", "Type", "Definition", "Example"]
215                     if options.get("include_timestamps"):
216                         headers.extend(["Created", "Updated"])
217                     if options.get("include_metadata"):
218                         headers.extend(["Difficulty", "Frequency", "Cultural_Tags", "Grammar_Notes"])
219
220                     writer.writerow(headers)
221
222                     # Data
223                     words = self.db.read_all_words()
224                     for word in words:
225                         row = list(word[:6])  # Basic fields
226                         if options.get("include_timestamps") and len(word) > 10:
227                             row.extend([word[10], word[11]])  # created_at, updated_at
228                         if options.get("include_metadata") and len(word) > 6:
229                             row.extend([
230                                 word[6] if len(word) > 6 else "",  # difficulty
231                                 word[7] if len(word) > 7 else "",  # frequency
232                                 word[8] if len(word) > 8 else "",  # cultural_tags
233                                 word[9] if len(word) > 9 else ""   # grammar_notes
234                             ])
235                         writer.writerow(row)
236
237                 elif data_type == "analytics":
238                     writer = csv.writer(csv_file)
239                     writer.writerow(["Metric", "Value", "Description"])
```

```
                words = self.db.read_all_words()
                user_facts = self.user_profile.get_user_facts()

                analytics_data = [
                    ("Total_Words", len(words), "Total dictionary entries"),
                    ("User_Level", user_facts.get("user_level", "unknown"), "Current difficulty level"),
                    ("Search_Count", user_facts.get("search_count", 0), "Total searches performed"),
                    ("Session_Time", user_facts.get("session_time", 0), "Current session time in seconds"),
                    ("Avg_Word_Length", user_facts.get("avg_word_length", 0), "Average search term length")
                ]

                for metric, value, desc in analytics_data:
                    writer.writerow([metric, value, desc])

            elif data_type == "profile":
                writer = csv.writer(csv_file)
                writer.writerow(["Setting", "Value"])

                profile_data = [
                    ("Difficulty_Level", self.user_profile.profile.get("difficulty_level", "beginner")),
                    ("Language_Preference", self.user_profile.profile.get("language_preference", "english")),
                    ("Total_Searches", self.user_profile.profile.get("search_count", 0)),
                    ("Last_Active", self.user_profile.profile.get("last_active", "unknown"))
                ]

                for setting, value in profile_data:
                    writer.writerow([setting, value])

            elif data_type == "history":
                writer = csv.writer(csv_file)
                writer.writerow(["Term", "Type", "Found", "Timestamp"])

```

```
                history = self.user_profile.profile.get("search_history", [])
                if history:
                    for search in history[-100:]:  # Last 100 searches
                        writer.writerow([
                            search.get("term", ""),
                            search.get("type", ""),
                            search.get("found", False),
                            search.get("timestamp", "")
                        ])
                else:
                    writer.writerow(["No search history available", "", "", ""])

            return True
        except Exception as e:
            print(f"CSV export error: {e}")
            return False

    def _export_json(self, data_type, filename, options):
        """Export data as JSON"""
        try:
            data = {}

            if data_type == "dictionary":
                words = self.db.read_all_words()
                data["dictionary"] = []

                for word in words:
                    word_dict = {
                        "id": word[0],
                        "english": word[1],
                        "khmer": word[2],
                        "type": word[3],
                        "definition": word[4],
                        "example": word[5]
                    }

                    if options.get("include_metadata") and len(word) > 6:
                        word_dict.update({
                            "difficulty": word[6] if len(word) > 6 else "",
                            "frequency": word[7] if len(word) > 7 else "",
                            "cultural_tags": word[8] if len(word) > 8 else "",
                            "grammar_notes": word[9] if len(word) > 9 else ""
                        })
```

```
            if options.get("include_timestamps") and len(word) > 10:
                word_dict.update({
                    "created_at": word[10] if len(word) > 10 else "",
                    "updated_at": word[11] if len(word) > 11 else ""
                })

            data["dictionary"].append(word_dict)

        elif data_type == "analytics":
            words = self.db.read_all_words()
            user_facts = self.user_profile.get_user_facts()

            data["analytics"] = {
                "total_words": len(words),
                "user_stats": user_facts,
                "generated_at": datetime.now().isoformat() if options.get("include_timestamps") else None
            }

        elif data_type == "profile":
            data["user_profile"] = dict(self.user_profile.profile)
            # Convert defaultdicts to regular dicts for JSON serialization
            if "word_preferences" in data["user_profile"]:
                data["user_profile"]["word_preferences"] = dict(data["user_profile"]["word_preferences"])
            if "favorite_word_types" in data["user_profile"]:
                data["user_profile"]["favorite_word_types"] = dict(data["user_profile"]["favorite_word_types"])

        elif data_type == "history":
            data["search_history"] = self.user_profile.profile.get("search_history", [])
            if not data["search_history"]:
                data["search_history"] = [{"note": "No search history available"}]

        if options.get("include_metadata"):
            data["export_metadata"] = {
                "export_time": datetime.now().isoformat(),
                "export_version": "2.1",
                "data_type": data_type
            }

        with open(filename, 'w', encoding='utf-8') as json_file:
            json.dump(data, json_file, indent=2, ensure_ascii=False)

        return True
```

```
359         except Exception as e:
360             print(f"JSON export error: {e}")
361             return False
362 
363     def _export_txt(self, data_type, filename, options):
364         """Export data as plain text"""
365         try:
366             with open(filename, 'w', encoding='utf-8') as text_file:
367                 text_file.write(f"Khmer-English Dictionary Export\n")
368                 text_file.write(f"Data Type: {data_type.title()}\n")
369                 if options.get("include_timestamps"):
370                     text_file.write(f"Exported: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")
371                 text_file.write("=" * 50 + "\n\n")
372 
373                 if data_type == "dictionary":
374                     words = self.db.read_all_words()
375                     for word in words:
376                         text_file.write(f"ID: {word[0]}\n")
377                         text_file.write(f"English: {word[1]}\n")
378                         text_file.write(f"Khmer: {word[2]}\n")
379                         text_file.write(f"Type: {word[3]}\n")
380                         if word[4]:
381                             text_file.write(f"Definition: {word[4]}\n")
382                         if word[5]:
383                             text_file.write(f"Example: {word[5]}\n")
384                         text_file.write("-" * 30 + "\n")
385 
386                 elif data_type == "analytics":
387                     words = self.db.read_all_words()
388                     user_facts = self.user_profile.get_user_facts()
389 
390                     text_file.write(f"Total Dictionary Entries: {len(words)}\n")
391                     text_file.write(f"User Level: {user_facts.get('user_level', 'unknown')}\n")
392                     text_file.write(f"Total Searches: {user_facts.get('search_count', 0)}\n")
393                     text_file.write(f"Session Time: {user_facts.get('session_time', 0)} seconds\n")
394 
395                 elif data_type == "profile":
396                     profile = self.user_profile.profile
397                     for key, value in profile.items():
398                         if key not in ["search_history", "word_preferences", "favorite_word_types"]:
399                             text_file.write(f"{key.title()}: {value}\n")
400 
```

```
            elif data_type == "history":
                history = self.user_profile.profile.get("search_history", [])
                if history:
                    for search in history[-50:]:  # Last 50 searches
                        text_file.write(f"Term: {search.get('term', '')}\n")
                        text_file.write(f"Type: {search.get('type', '')}\n")
                        text_file.write(f"Found: {search.get('found', False)}\n")
                        text_file.write(f"Time: {search.get('timestamp', '')}\n")
                        text_file.write("-" * 20 + "\n")
                else:
                    text_file.write("No search history available.\n")

            return True
        except Exception as e:
            print(f"TXT export error: {e}")
            return False

    def _export_html(self, data_type, filename, options):
        """Export data as HTML"""
        try:
            with open(filename, 'w', encoding='utf-8') as html_file:
                html_file.write("<!DOCTYPE html>\n<html>\n<head>\n")
                html_file.write("<meta charset='UTF-8'>\n")
                html_file.write(f"<title>Khmer-English Dictionary - {data_type.title()}</title>\n")
                html_file.write("<style>\n")
                html_file.write("body { font-family: Arial, sans-serif; margin: 20px; }\n")
                html_file.write("table { border-collapse: collapse; width: 100%; }\n")
                html_file.write("th, td { border: 1px solid #ddd; padding: 8px; text-align: left; }\n")
                html_file.write("th { background-color: #f2f2f2; }\n")
                html_file.write("h1 { color: #2E7D32; }\n")
                html_file.write("</style>\n</head>\n<body>\n")

                html_file.write(f"<h1>Khmer-English Dictionary Export</h1>\n")
                html_file.write(f"<h2>Data Type: {data_type.title()}</h2>\n")
                if options.get("include_timestamps"):
                    html_file.write(f"<p>Exported: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}</p>\n")

                if data_type == "dictionary":
                    words = self.db.read_all_words()
                    html_file.write("<table>\n<tr>\n")
                    html_file.write("<th>English</th><th>Khmer</th><th>Type</th><th>Definition</th><th>Example</th>\n")
                    html_file.write("</tr>\n")
```

```
                for word in words:
                    html_file.write("<tr>\n")
                    html_file.write(f"<td>{word[1]}</td>")
                    html_file.write(f"<td>{word[2]}</td>")
                    html_file.write(f"<td>{word[3]}</td>")
                    html_file.write(f"<td>{word[4] or ''}</td>")
                    html_file.write(f"<td>{word[5] or ''}</td>")
                    html_file.write("</tr>\n")

                html_file.write("</table>\n")

            elif data_type == "analytics":
                words = self.db.read_all_words()
                user_facts = self.user_profile.get_user_facts()

                html_file.write("<h3>Statistics Summary</h3>\n")
                html_file.write(f"<p><strong>Total Words:</strong> {len(words)}</p>\n")
                html_file.write(f"<p><strong>User Level:</strong> {user_facts.get('user_level', 'unknown')}</p>\n")
                html_file.write(f"<p><strong>Total Searches:</strong> {user_facts.get('search_count', 0)}</p>\n")

            html_file.write("</body>\n</html>\n")

        return True
    except Exception as e:
        print(f"HTML export error: {e}")
        return False

# Voice Search Components
class VoiceSearchThread(QThread):
    """Thread for handling voice recognition without blocking UI"""
    speech_recognized = pyqtSignal(str)
    speech_error = pyqtSignal(str)
    recording_started = pyqtSignal()
    recording_stopped = pyqtSignal()

    def __init__(self):
        super().__init__()
        self.recognizer = None
        self.microphone = None
        self.is_listening = False

```

```
        if SPEECH_RECOGNITION_AVAILABLE:
            try:
                self.recognizer = sr.Recognizer()
                self.microphone = sr.Microphone()
                # Adjust for ambient noise
                with self.microphone as source:
                    self.recognizer.adjust_for_ambient_noise(source, duration=0.5)
            except Exception as e:
                print(f"Voice recognition setup error: {e}")

    def run(self):
        """Run voice recognition in background thread"""
        if not SPEECH_RECOGNITION_AVAILABLE or not self.recognizer or not self.microphone:
            self.speech_error.emit("Speech recognition not available")
            return

        try:
            self.recording_started.emit()
            self.is_listening = True

            with self.microphone as source:
                # Listen for audio with timeout
                audio = self.recognizer.listen(source, timeout=5, phrase_time_limit=10)

            self.recording_stopped.emit()
            self.is_listening = False

            # Try different recognition services
            try:
                # Try Google first (requires internet)
                text = self.recognizer.recognize_google(audio)
                self.speech_recognized.emit(text)
            except sr.RequestError:
                try:
                    # Fallback to offline recognition (if available)
                    text = self.recognizer.recognize_sphinx(audio)
                    self.speech_recognized.emit(text)
                except:
                    self.speech_error.emit("Could not recognize speech. Check internet connection.")
            except sr.UnknownValueError:
                self.speech_error.emit("Could not understand audio. Please speak clearly.")
```

```
527             except sr.WaitTimeoutError:
528                 self.recording_stopped.emit()
529                 self.speech_error.emit("Listening timeout. No speech detected.")
530             except Exception as e:
531                 self.recording_stopped.emit()
532                 self.speech_error.emit(f"Voice recognition error: {str(e)}")
533 
534     def stop_listening(self):
535         """Stop the listening process"""
536         self.is_listening = False
537         self.quit()
538 
539 # Expert System Components
540     """Thread for handling voice recognition without blocking UI"""
541     speech_recognized = pyqtSignal(str)
542     speech_error = pyqtSignal(str)
543     recording_started = pyqtSignal()
544     recording_stopped = pyqtSignal()
545 
546     def __init__(self):
547         super().__init__()
548         self.recognizer = None
549         self.microphone = None
550         self.is_listening = False
551 
552         if SPEECH_RECOGNITION_AVAILABLE:
553             try:
554                 self.recognizer = sr.Recognizer()
555                 self.microphone = sr.Microphone()
556                 # Adjust for ambient noise
557                 with self.microphone as source:
558                     self.recognizer.adjust_for_ambient_noise(source, duration=0.5)
559             except Exception as e:
560                 print(f"Voice recognition setup error: {e}")
561 
562     def run(self):
563         """Run voice recognition in background thread"""
564         if not SPEECH_RECOGNITION_AVAILABLE or not self.recognizer or not self.microphone:
565             self.speech_error.emit("Speech recognition not available")
566             return
567 
```

```
        try:
            self.recording_started.emit()
            self.is_listening = True

            with self.microphone as source:
                # Listen for audio with timeout
                audio = self.recognizer.listen(source, timeout=5, phrase_time_limit=10)

            self.recording_stopped.emit()
            self.is_listening = False

            # Try different recognition services
            try:
                # Try Google first (requires internet)
                text = self.recognizer.recognize_google(audio)
                self.speech_recognized.emit(text)
            except sr.RequestError:
                try:
                    # Fallback to offline recognition (if available)
                    text = self.recognizer.recognize_sphinx(audio)
                    self.speech_recognized.emit(text)
                except:
                    self.speech_error.emit("Could not recognize speech. Check internet connection.")
            except sr.UnknownValueError:
                self.speech_error.emit("Could not understand audio. Please speak clearly.")

        except sr.WaitTimeoutError:
            self.recording_stopped.emit()
            self.speech_error.emit("Listening timeout. No speech detected.")
        except Exception as e:
            self.recording_stopped.emit()
            self.speech_error.emit(f"Voice recognition error: {str(e)}")

    def stop_listening(self):
        """Stop the listening process"""
        self.is_listening = False
        self.quit()

```

```python
# Expert System Components
class Rule:
    """Individual rule in the expert system"""
    def __init__(self, name: str, conditions: List[str], actions: List[str], confidence: float = 1.0):
        self.name = name
        self.conditions = conditions
        self.actions = actions
        self.confidence = confidence
        self.fired_count = 0

    def evaluate(self, facts: Dict) -> bool:
        """Evaluate if all conditions are met"""
        for condition in self.conditions:
            if not self._evaluate_condition(condition, facts):
                return False
        return True

    def _evaluate_condition(self, condition: str, facts: Dict) -> bool:
        """Evaluate a single condition"""
        try:
            # Simple condition evaluation
            # Format: "word_type == 'noun'" or "search_count > 5"
            for key, value in facts.items():
                condition = condition.replace(key, f"'{value}'" if isinstance(value, str) else str(value))
            return eval(condition)
        except:
            return False

    def fire(self, facts: Dict) -> List[str]:
        """Execute rule actions"""
        self.fired_count += 1
        suggestions = []
        for action in self.actions:
            suggestion = self._execute_action(action, facts)
            if suggestion:
                suggestions.append(suggestion)
        return suggestions

    def _execute_action(self, action: str, facts: Dict) -> str:
        """Execute a single action"""
        # Replace placeholders in actions
        for key, value in facts.items():
            action = action.replace(f"{{{key}}}", str(value))
        return action
```

```
650
651 class ExpertSystemEngine:
652     """Core expert system reasoning engine"""
653     def __init__(self):
654         self.rules = []
655         self.facts = {}
656         self.inference_chain = []
657         self.initialize_rules()
658
659     def initialize_rules(self):
660         """Initialize the rule base"""
661         # Word type suggestion rules
662         self.add_rule(Rule(
663             "noun_suggestions",
664             ["word_type == 'noun'", "not_found == True"],
665             ["Try searching for related objects, places, or things",
666             "Consider checking spelling or trying synonyms"]
667         ))
668
669         self.add_rule(Rule(
670             "verb_suggestions",
671             ["word_type == 'verb'", "not_found == True"],
672             ["Try searching for action words or infinitive forms",
673             "Check for irregular verb forms"]
674         ))
675
676         # User behavior rules
677         self.add_rule(Rule(
678             "frequent_searcher",
679             ["search_count > 10"],
680             ["You're an active learner! Try exploring word categories",
681             "Consider using the advanced features in Dictionary Manager"]
682         ))
683
684         self.add_rule(Rule(
685             "khmer_learner",
686             ["khmer_searches > english_searches", "session_time > 300"],
687             ["Focus on Khmer script recognition",
688             "Try using example sentences for better context"]
689         ))
690
```

```
691         # Difficulty adaptation rules
692         self.add_rule(Rule(
693             "beginner_user",
694             ["avg_word_length < 5", "search_count < 5"],
695             ["Start with basic vocabulary",
696             "Use simple example sentences"]
697         ))
698 
699         self.add_rule(Rule(
700             "advanced_user",
701             ["avg_word_length > 8", "search_count > 20"],
702             ["Explore complex grammar patterns",
703             "Try advanced word types and idioms"]
704         ))
705 
706         # Context-based rules
707         self.add_rule(Rule(
708             "grammar_help",
709             ["word_type == 'adjective'", "user_level == 'beginner'"],
710             ["Adjectives in Khmer often follow nouns",
711             "Practice with simple descriptive phrases"]
712         ))
713 
714         self.add_rule(Rule(
715             "cultural_context",
716             ["contains_honorific == True"],
717             ["This word shows respect in Khmer culture",
718             "Usage depends on social context and age"]
719         ))
720 
721     def add_rule(self, rule: Rule):
722         """Add a new rule to the system"""
723         self.rules.append(rule)
724 
725     def add_fact(self, key: str, value):
726         """Add a fact to the knowledge base"""
727         self.facts[key] = value
728 
729     def infer(self) -> List[str]:
730         """Run inference engine and return suggestions"""
731         suggestions = []
732         self.inference_chain = []
```

```
733
734         for rule in self.rules:
735             if rule.evaluate(self.facts):
736                 rule_suggestions = rule.fire(self.facts)
737                 suggestions.extend(rule_suggestions)
738                 self.inference_chain.append(f"Fired rule: {rule.name}")
739
740         return suggestions
741
742     def get_explanation(self) -> str:
743         """Get explanation of inference process"""
744         return "\n".join(self.inference_chain)
745
746 class SimpleWordEmbeddings:
747     """Lightweight word embeddings for similarity search"""
748     def __init__(self):
749         self.embeddings = {}
750         self.vocabulary = set()
751         self.similarity_cache = {}
752
753     def add_word(self, word: str, context: str = ""):
754         """Add word to vocabulary with simple feature extraction"""
755         word = word.lower().strip()
756         self.vocabulary.add(word)
757
758         # Simple feature vector based on character n-grams and length
759         features = self._extract_features(word, context)
760         self.embeddings[word] = features
761
762     def _extract_features(self, word: str, context: str = "") -> List[float]:
763         """Extract simple features for word similarity"""
764         features = []
765
766         # Length feature
767         features.append(len(word) / 20.0)  # Normalized length
768
769         # Character frequency features
770         char_counts = Counter(word.lower())
771         common_chars = 'aeioutrnslh'
772         for char in common_chars:
773             features.append(char_counts.get(char, 0) / len(word))
774
```

```
        # N-gram features (bigrams)
        bigrams = [word[i:i+2] for i in range(len(word)-1)]
        common_bigrams = ['th', 'er', 'on', 'an', 're', 'he', 'in', 'ed', 'nd', 'ha']
        for bigram in common_bigrams:
            features.append(1.0 if bigram in bigrams else 0.0)

        # Context features
        if context:
            features.append(1.0 if 'example' in context.lower() else 0.0)
            features.append(1.0 if any(word in context.lower() for word in ['formal', 'respect', 'polite']) else 0.0)

        return features

    def find_similar(self, word: str, n: int = 5) -> List[Tuple[str, float]]:
        """Find n most similar words"""
        word = word.lower().strip()
        if word not in self.embeddings:
            return []

        similarities = []
        target_features = self.embeddings[word]

        for other_word, other_features in self.embeddings.items():
            if other_word != word:
                similarity = self._cosine_similarity(target_features, other_features)
                similarities.append((other_word, similarity))

        similarities.sort(key=lambda x: x[1], reverse=True)
        return similarities[:n]

    def _cosine_similarity(self, vec1: List[float], vec2: List[float]) -> float:
        """Calculate cosine similarity between two vectors"""
        if len(vec1) != len(vec2):
            return 0.0

        dot_product = sum(a * b for a, b in zip(vec1, vec2))
        magnitude1 = math.sqrt(sum(a * a for a in vec1))
        magnitude2 = math.sqrt(sum(a * a for a in vec2))

        if magnitude1 == 0 or magnitude2 == 0:
            return 0.0

```

```
817         return dot_product / (magnitude1 * magnitude2)
818
819 class UserProfileManager:
820     """Track and model user behavior"""
821     def __init__(self, profile_file: str = "user_profile.json"):
822         self.profile_file = profile_file
823         self.profile = self._load_profile()
824         self.session_start = datetime.now()
825
826         # Ensure defaultdicts are properly set up
827         if not isinstance(self.profile.get("word_preferences"), defaultdict):
828             self.profile["word_preferences"] = defaultdict(int, self.profile.get("word_preferences", {}))
829         if not isinstance(self.profile.get("favorite_word_types"), defaultdict):
830             self.profile["favorite_word_types"] = defaultdict(int, self.profile.get("favorite_word_types", {}))
831
832     def _load_profile(self) -> Dict:
833         """Load user profile from file"""
834         default_profile = {
835             "search_history": [],
836             "word_preferences": defaultdict(int),
837             "language_preference": "english",
838             "difficulty_level": "beginner",
839             "search_count": 0,
840             "session_time": 0,
841             "favorite_word_types": defaultdict(int),
842             "error_patterns": [],
843             "learning_progress": {"beginner": 0, "intermediate": 0, "advanced": 0},
844             "last_active": datetime.now().isoformat()
845         }
846
847         try:
848             if os.path.exists(self.profile_file):
849                 with open(self.profile_file, 'r', encoding='utf-8') as f:
850                     loaded = json.load(f)
851                     # Merge with defaults to handle new fields
852                     for key, value in default_profile.items():
853                         if key not in loaded:
854                             loaded[key] = value
855
```

```
                # Convert regular dicts back to defaultdicts
                if 'word_preferences' in loaded and not isinstance(loaded['word_preferences'], defaultdict):
                    loaded['word_preferences'] = defaultdict(int, loaded['word_preferences'])
                if 'favorite_word_types' in loaded and not isinstance(loaded['favorite_word_types'], defaultdict):
                    loaded['favorite_word_types'] = defaultdict(int, loaded['favorite_word_types'])

                return loaded
        except Exception as e:
            print(f"Error loading profile: {e}")

        return default_profile

    def save_profile(self):
        """Save user profile to file"""
        try:
            # Update session time
            self.profile["session_time"] = (datetime.now() - self.session_start).seconds
            self.profile["last_active"] = datetime.now().isoformat()

            # Convert defaultdict to regular dict for JSON serialization
            profile_to_save = dict(self.profile)
            for key, value in profile_to_save.items():
                if isinstance(value, defaultdict):
                    profile_to_save[key] = dict(value)

            with open(self.profile_file, 'w', encoding='utf-8') as f:
                json.dump(profile_to_save, f, indent=2, ensure_ascii=False)
        except Exception as e:
            print(f"Error saving profile: {e}")

    def record_search(self, term: str, search_type: str, found: bool):
        """Record a search action"""
        search_record = {
            "term": term,
            "type": search_type,
            "found": found,
            "timestamp": datetime.now().isoformat()
        }

        self.profile["search_history"].append(search_record)
        self.profile["search count"] += 1
```

```
897
898         # Keep only last 1000 searches
899         if len(self.profile["search_history"]) > 1000:
900             self.profile["search_history"] = self.profile["search_history"][-1000:]
901
902         # Update preferences - ensure it's a defaultdict
903         if not isinstance(self.profile["word_preferences"], defaultdict):
904             self.profile["word_preferences"] = defaultdict(int, self.profile["word_preferences"])
905
906         if found:
907             self.profile["word_preferences"][term] += 1
908
909     def record_word_interaction(self, word_type: str, difficulty: str = "medium"):
910         """Record interaction with specific word type"""
911         # Ensure it's a defaultdict
912         if not isinstance(self.profile["favorite_word_types"], defaultdict):
913             self.profile["favorite_word_types"] = defaultdict(int, self.profile["favorite_word_types"])
914
915         self.profile["favorite_word_types"][word_type] += 1
916
917         # Update difficulty level based on usage patterns
918         if difficulty == "hard" and self.profile["difficulty_level"] == "beginner":
919             self.profile["learning_progress"]["intermediate"] += 1
920             if self.profile["learning_progress"]["intermediate"] > 20:
921                 self.profile["difficulty_level"] = "intermediate"
922
923     def get_recommendations(self) -> List[str]:
924         """Get personalized recommendations"""
925         recommendations = []
926
927         # Analyze search patterns
928         recent_searches = self.profile["search_history"][-20:] if self.profile["search_history"] else []
929
930         if len(recent_searches) > 5:
931             # Find most searched word types
932             type_counts = defaultdict(int)
933             for search in recent_searches:
934                 # Simple word type detection (would be enhanced with actual word type data)
935                 if len(search["term"]) > 8:
936                     type_counts["complex"] += 1
937                 elif search["type"] == "khmer":
938                     type_counts["khmer_focus"] += 1
939
```

```
            if type_counts["complex"] > 3:
                recommendations.append("You're exploring advanced vocabulary! Try the word type filters.")

            if type_counts["khmer_focus"] > 3:
                recommendations.append("Great Khmer practice! Consider learning about Khmer grammar patterns.")

        # Difficulty-based recommendations
        if self.profile["difficulty_level"] == "beginner" and self.profile["search_count"] > 10:
            recommendations.append("Ready for intermediate words? Try exploring different word types.")

        return recommendations

    def get_user_facts(self) -> Dict:
        """Get user facts for expert system"""
        session_time = (datetime.now() - self.session_start).seconds
        recent_searches = self.profile["search_history"][-10:] if self.profile["search_history"] else []

        khmer_searches = sum(1 for s in recent_searches if s["type"] == "khmer")
        english_searches = sum(1 for s in recent_searches if s["type"] == "english")

        avg_word_length = 0
        if recent_searches:
            avg_word_length = sum(len(s["term"]) for s in recent_searches) / len(recent_searches)

        return {
            "search_count": self.profile["search_count"],
            "session_time": session_time,
            "khmer_searches": khmer_searches,
            "english_searches": english_searches,
            "avg_word_length": avg_word_length,
            "user_level": self.profile["difficulty_level"],
            "favorite_types": list(self.profile["favorite_word_types"].keys())
        }

class AIExplanationGenerator:
    """Generate contextual explanations and grammar help"""
    def __init__(self):
        self.grammar_rules = self._load_grammar_rules()
        self.cultural_context = self._load_cultural_context()

```

```
    def _load_grammar_rules(self) -> Dict:
        """Load Khmer grammar rules"""
        return {
            "noun": {
                "structure": "Khmer nouns don't change form for singular/plural",
                "usage": "Often combined with classifiers when counting",
                "example": "សៀវភៅ (book) + មួយ (one) + ក្បាល (classifier)"
            },
            "verb": {
                "structure": "Verbs don't conjugate for tense like English",
                "usage": "Time indicators show when action occurs",
                "example": "ហូប (eat) + ហើយ (already) = ate"
            },
            "adjective": {
                "structure": "Usually follows the noun it describes",
                "usage": "Can be intensified with particles",
                "example": "ផ្ទះ (house) + ធំ (big) = big house"
            },
            "greeting": {
                "structure": "Varies by time of day and formality",
                "usage": "Shows respect and social awareness",
                "example": "សួស្តី for casual, ជំរាបសួរ for formal"
            }
        }

    def _load_cultural_context(self) -> Dict:
        """Load cultural context information"""
        return {
            "honorifics": "Khmer uses different vocabulary levels to show respect",
            "family_terms": "Family relationships are very specific in Khmer",
            "religious_terms": "Buddhism influences many expressions",
            "formal_speech": "Used with elders, authorities, and strangers",
            "age_hierarchy": "Language changes based on relative age"
        }

    def generate_explanation(self, word_data: Tuple, context: str = "") -> str:
        """Generate detailed explanation for a word"""
        if not word_data:
            return "No explanation available."

        word_id, english, khmer, word_type, definition, example = word_data[:6]

        explanation = f"<h3>Expert Analysis: {english} → {khmer}</h3>"
```

```
1023
1024        # Grammar explanation
1025        if word_type in self.grammar_rules:
1026            rule = self.grammar_rules[word_type]
1027            explanation += f"""
1028            <h4>Grammar Structure:</h4>
1029            <p><strong>Rule:</strong> {rule['structure']}</p>
1030            <p><strong>Usage:</strong> {rule['usage']}</p>
1031            <p><strong>Pattern:</strong> {rule['example']}</p>
1032            """
1033
1034        # Word analysis
1035        explanation += f"""
1036        <h4>Word Analysis:</h4>
1037        <ul>
1038            <li><strong>Type:</strong> {word_type.title()}</li>
1039            <li><strong>Length:</strong> {len(khmer)} Khmer characters</li>
1040            <li><strong>Complexity:</strong> {'Advanced' if len(english) > 8 else 'Basic'}</li>
1041        </ul>
1042        """
1043
1044        # Cultural context
1045        cultural_notes = self._get_cultural_notes(english, khmer, word_type)
1046        if cultural_notes:
1047            explanation += f"""
1048            <h4>Cultural Context:</h4>
1049            <p>{cultural_notes}</p>
1050            """
1051
1052        # Usage tips
1053        tips = self._generate_usage_tips(english, khmer, word_type)
1054        explanation += f"""
1055        <h4>Usage Tips:</h4>
1056        <ul>
1057        """
1058        for tip in tips:
1059            explanation += f"<li>{tip}</li>"
1060        explanation += "</ul>"
1061
```

```
        # Related concepts
        explanation += f"""
        <h4>Learning Connections:</h4>
        <p>This word connects to: {word_type} vocabulary, Khmer script practice,
        {'formal speech patterns' if len(english) > 6 else 'basic conversation'}</p>
        """

        return explanation

    def _get_cultural_notes(self, english: str, khmer: str, word_type: str) -> str:
        """Get cultural context for the word"""
        cultural_keywords = {
            "family": ["mother", "father", "uncle", "aunt", "grandparent"],
            "respect": ["please", "thank", "sorry", "excuse"],
            "religion": ["temple", "monk", "pray", "festival"],
            "food": ["rice", "fish", "soup", "curry"]
        }

        for category, keywords in cultural_keywords.items():
            if any(keyword in english.lower() for keyword in keywords):
                if category == "family":
                    return "Family terms in Khmer are very specific about relationships and age."
                elif category == "respect":
                    return "This word is important for polite communication in Khmer culture."
                elif category == "religion":
                    return "This term reflects Buddhist influence in Cambodian culture."
                elif category == "food":
                    return "Food vocabulary is central to Cambodian social interactions."

        return ""

    def _generate_usage_tips(self, english: str, khmer: str, word_type: str) -> List[str]:
        """Generate practical usage tips"""
        tips = []

        if word_type == "noun":
            tips.append("Remember: Khmer nouns don't change for plural forms")
            tips.append("Use classifiers when counting specific quantities")
        elif word_type == "verb":
            tips.append("Add time markers (ហើយ, នឹង) to indicate tense")
            tips.append("Verb order: Subject + Verb + Object")
        elif word_type == "adjective":
```

```
            tips.append("Place after the noun: [noun] + [adjective]")
            tips.append("Can intensify with 'ណាស់' (very)")
        elif word_type == "greeting":
            tips.append("Choose formality level based on the situation")
            tips.append("Time of day affects which greeting to use")

        # Length-based tips
        if len(khmer) > 6:
            tips.append("Complex word - practice writing the script slowly")
        else:
            tips.append("Short word - good for memorization practice")

        # Frequency tips
        if english.lower() in ["hello", "thank", "please", "water", "food"]:
            tips.append("High-frequency word - essential for daily conversation")

        return tips

# Enhanced Database with Expert System Features
class DictionaryDatabase:
    """Enhanced database with expert system features"""
    def __init__(self, db_path="expert_khmer_dictionary.db"):
        self.db_path = db_path
        self.embeddings = SimpleWordEmbeddings()
        self.init_database()
        self._build_embeddings()

    def init_database(self):
        """Initialize database with enhanced schema"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Main dictionary table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS dictionary (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                english_word TEXT NOT NULL UNIQUE,
                khmer_word TEXT NOT NULL,
                word_type TEXT DEFAULT 'noun',
                definition TEXT,
                example_sentence TEXT,
                difficulty_level TEXT DEFAULT 'beginner',
```

```
            frequency_score INTEGER DEFAULT 1,
            cultural_tags TEXT,
            grammar_notes TEXT,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
    ''')

    # User interactions table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS user_interactions (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            word_id INTEGER,
            interaction_type TEXT,
            timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            user_rating INTEGER,
            FOREIGN KEY (word_id) REFERENCES dictionary (id)
        )
    ''')

    # Check if we need to populate with sample data
    cursor.execute("SELECT COUNT(*) FROM dictionary")
    if cursor.fetchone()[0] == 0:
        self._insert_enhanced_sample_data(cursor)

    conn.commit()
    conn.close()

def _insert_enhanced_sample_data(self, cursor):
    """Insert enhanced sample data with expert system features"""
    enhanced_sample_data = [
        ("hello", "ស្តីី", "greeting", "A common greeting", "Hello, how are you?", "beginner", 10, "social,polite", "Used for casual greetings"),
        ("goodbye", "លាហើយ", "greeting", "Farewell expression", "Goodbye, see you tomorrow!", "beginner", 8, "social,departure", "Casual farewell"),
        ("thank you", "អរគុណ", "expression", "Express gratitude", "Thank you for your help", "beginner", 9, "polite,respect", "Shows appreciation"),
        ("please", "សូម", "adverb", "Polite request", "Please help me", "beginner", 9, "polite,formal", "Makes requests polite"),
        ("yes", "ចាទ/ចាស", "response", "Affirmative response", "Yes, I agree", "beginner", 10, "response,agreement", "ចាទ for males, ចាស for females"),
        ("no", "ទេ", "response", "Negative response", "No, I don't want", "beginner", 8, "response,disagreement", "Simple negation"),
        ("water", "ទឹក", "noun", "Essential liquid", "I need water", "beginner", 9, "daily,essential", "Basic necessity"),
        ("food", "អាហារ", "noun", "Nutrition substance", "The food is delicious", "beginner", 8, "daily,culture", "Central to culture"),
        ("house", "ផ្ទះ", "noun", "Living building", "This is my house", "beginner", 7, "home,family", "Place of residence"),
        ("school", "សាលា", "noun", "Education institution", "I go to school every day", "beginner", 6, "education,children", "Learning place"),
        ("book", "សៀវភៅ", "noun", "Written work", "I'm reading a book", "intermediate", 5, "education,knowledge", "Knowledge container"),
```

```
            ("student", "សិស្ស", "noun", "Learning person", "She is a good student", "intermediate", 6, "education,youth", "Knowledge seeker"),
            ("teacher", "គ្រូ", "noun", "Education provider", "The teacher explains well", "intermediate", 5, "education,respect", "Highly respected profession"),
            ("mother", "ម្តាយ", "noun", "Female parent", "I love my mother", "beginner", 8, "family,respect", "Very important family role"),
            ("father", "ឪពុក", "noun", "Male parent", "My father works hard", "beginner", 8, "family,respect", "Family head traditionally"),
            ("beautiful", "ស្អាត", "adjective", "Pleasing appearance", "She is beautiful", "intermediate", 6, "description,appearance", "Common compliment"),
            ("big", "ធំ", "adjective", "Large size", "This is a big house", "beginner", 7, "size,description", "Basic size descriptor"),
            ("small", "តូច", "adjective", "Little size", "I have a small car", "beginner", 7, "size,description", "Opposite of big"),
            ("eat", "ញុំ", "verb", "Consume food", "Let's eat together", "beginner", 9, "daily,action", "Essential daily action"),
            ("go", "ទៅ", "verb", "Move to place", "I go to work", "beginner", 9, "movement,action", "Basic movement verb")
        ]

        cursor.executemany('''
            INSERT INTO dictionary (english_word, khmer_word, word_type, definition,
                                    example_sentence, difficulty_level, frequency_score,
                                    cultural_tags, grammar_notes)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        ''', enhanced_sample_data)

    def _build_embeddings(self):
        """Build word embeddings for similarity search"""
        words = self.read_all_words()
        for word in words:
            english, khmer, definition = word[1], word[2], word[4] or ""
            context = f"{definition} {word[8] or ''} {word[9] or ''}"  # Include cultural tags and grammar notes
            self.embeddings.add_word(english, context)
            self.embeddings.add_word(khmer, context)

    def find_similar_words(self, word: str, n: int = 5) -> List[Tuple[str, float]]:
        """Find similar words using embeddings"""
        return self.embeddings.find_similar(word, n)

    def get_smart_suggestions(self, search_term: str, search_type: str) -> List[Tuple]:
        """Get AI-powered suggestions for failed searches"""
        # First try similarity search
        similar_words = self.find_similar_words(search_term, 3)
        suggestions = []

        for similar_word, similarity in similar_words:
            if similarity > 0.3:  # Threshold for similarity
                results = self.read_word(similar_word, "english")
                if not results:
                    results = self.read_word(similar_word, "khmer")
                suggestions.extend(results)
```

```
1231
1232        return suggestions[:5]  # Return top 5 suggestions
1233
1234    # Include all methods from original DictionaryDatabase
1235    def create_word(self, english_word, khmer_word, word_type="noun", definition="", example="",
1236                    difficulty="beginner", cultural_tags="", grammar_notes=""):
1237        """Enhanced CREATE operation"""
1238        try:
1239            conn = sqlite3.connect(self.db_path)
1240            cursor = conn.cursor()
1241            cursor.execute('''
1242                INSERT INTO dictionary (english_word, khmer_word, word_type, definition,
1243                                        example_sentence, difficulty_level, cultural_tags, grammar_notes)
1244                VALUES (?, ?, ?, ?, ?, ?, ?, ?)
1245            ''', (english_word.lower().strip(), khmer_word.strip(), word_type,
1246                  definition, example, difficulty, cultural_tags, grammar_notes))
1247            conn.commit()
1248            word_id = cursor.lastrowid
1249            conn.close()
1250
1251            # Update embeddings
1252            context = f"{definition} {cultural_tags} {grammar_notes}"
1253            self.embeddings.add_word(english_word, context)
1254            self.embeddings.add_word(khmer_word, context)
1255
1256            return word_id
1257        except sqlite3.IntegrityError:
1258            raise ValueError(f"Word '{english_word}' already exists in dictionary")
1259        except Exception as e:
1260            raise ValueError(f"Database error: {str(e)}")
1261
1262    def read_word(self, search_term, search_type="english"):
1263        """Enhanced READ operation with AI suggestions"""
1264        conn = sqlite3.connect(self.db_path)
1265        cursor = conn.cursor()
1266
1267        try:
1268            if search_type == "english":
1269                cursor.execute('''
1270                    SELECT * FROM dictionary
1271                    WHERE english_word LIKE ? OR english_word = ?
1272                    ORDER BY frequency_score DESC, english_word
1273                ''', (f"%{search term.lower()}%", search term.lower()))
```

```
            else:
                cursor.execute('''
                    SELECT * FROM dictionary
                    WHERE khmer_word LIKE ?
                    ORDER BY frequency_score DESC, khmer_word
                ''', (f"%{search_term}%",))

            results = cursor.fetchall()
            conn.close()
            return results
        except Exception as e:
            conn.close()
            return []

    def read_all_words(self):
        """Enhanced READ ALL operation"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        try:
            cursor.execute("SELECT * FROM dictionary ORDER BY frequency_score DESC, english_word")
            results = cursor.fetchall()
            conn.close()
            return results
        except Exception as e:
            conn.close()
            return []

    def update_word(self, word_id, english_word=None, khmer_word=None, word_type=None,
                    definition=None, example=None, difficulty=None, cultural_tags=None, grammar_notes=None):
        """Enhanced UPDATE operation"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        try:
            updates = []
            params = []

            if english_word is not None:
                updates.append("english_word = ?")
                params.append(english_word.lower().strip())
            if khmer_word is not None:
                updates.append("khmer_word = ?")
```

```
            params.append(khmer_word.strip())
        if word_type is not None:
            updates.append("word_type = ?")
            params.append(word_type)
        if definition is not None:
            updates.append("definition = ?")
            params.append(definition)
        if example is not None:
            updates.append("example_sentence = ?")
            params.append(example)
        if difficulty is not None:
            updates.append("difficulty_level = ?")
            params.append(difficulty)
        if cultural_tags is not None:
            updates.append("cultural_tags = ?")
            params.append(cultural_tags)
        if grammar_notes is not None:
            updates.append("grammar_notes = ?")
            params.append(grammar_notes)

        updates.append("updated_at = CURRENT_TIMESTAMP")
        params.append(word_id)

        if updates:
            query = f"UPDATE dictionary SET {', '.join(updates)} WHERE id = ?"
            cursor.execute(query, params)
            conn.commit()

        conn.close()
    except Exception as e:
        conn.close()
        raise ValueError(f"Update error: {str(e)}")

def delete_word(self, word_id):
    """Enhanced DELETE operation"""
    conn = sqlite3.connect(self.db_path)
    cursor = conn.cursor()
    try:
        cursor.execute("DELETE FROM dictionary WHERE id = ?", (word_id,))
        conn.commit()
        deleted_count = cursor.rowcount
        conn.close()
        return deleted_count > 0
```

```
1359            except Exception as e:
1360                conn.close()
1361                raise ValueError(f"Delete error: {str(e)}")
1362
1363        def get_random_words(self, limit=5):
1364            """Get random words with enhanced data"""
1365            conn = sqlite3.connect(self.db_path)
1366            cursor = conn.cursor()
1367            try:
1368                cursor.execute("SELECT * FROM dictionary ORDER BY RANDOM() LIMIT ?", (limit,))
1369                results = cursor.fetchall()
1370                conn.close()
1371                return results
1372            except Exception as e:
1373                conn.close()
1374                return []
1375
1376    # Keep standard UI classes with expert system integration
1377    class FontManager:
1378        """Manage Khmer OS Siemreap font for the application - Single font size 11"""
1379
1380        def __init__(self):
1381            self.khmer_font = None
1382            self.font_size = 11  # Single font size for entire application
1383            self.init_fonts()
1384
1385        def init_fonts(self):
1386            """Initialize and load Khmer OS Siemreap font"""
1387            preferred_fonts = [
1388                "Khmer OS Siemreap"
1389            ]
1390
1391            try:
1392                available_families = QFontDatabase().families()
1393            except:
1394                available_families = []
1395
1396            for font_name in preferred_fonts:
1397                if font_name in available_families:
1398                    self.khmer_font = QFont(font_name, self.font_size)
1399                    break
1400
```

```
1401        if self.khmer_font is None:
1402            self.khmer_font = QFont("Khmer OS Siemreap", self.font_size)
1403
1404        # Configure font properties for optimal Khmer rendering
1405        self.khmer_font.setStyleHint(QFont.StyleHint.System)
1406        self.khmer_font.setWeight(QFont.Weight.Normal)
1407        self.khmer_font.setStyleStrategy(QFont.StyleStrategy.PreferAntialias | QFont.StyleStrategy.PreferQuality)
1408        self.khmer_font.setHintingPreference(QFont.HintingPreference.PreferFullHinting)
1409
1410    def get_font(self, size=None, bold=False):
1411        """Get the standard font with specified size and weight"""
1412        if size is None:
1413            size = self.font_size
1414        font = QFont(self.khmer_font)
1415        font.setPointSize(size)
1416        if bold:
1417            font.setWeight(QFont.Weight.Bold)
1418        return font
1419
1420    def get_font_family(self):
1421        """Get the font family name"""
1422        return self.khmer_font.family()
1423
1424    def apply_font(self, widget, size=None, bold=False):
1425        """Apply font to any widget recursively"""
1426        if size is None:
1427            size = self.font_size
1428        try:
1429            font = self.get_font(size, bold)
1430            widget.setFont(font)
1431            # Apply to all child widgets recursively
1432            for child in widget.findChildren(QWidget):
1433                if hasattr(child, 'setFont'):
1434                    child.setFont(font)
1435        except Exception as e:
1436            print(f"Font application error: {e}")
1437
1438    def create_message_box(self, parent, icon, title, text, buttons=None):
1439        """Create a message box with proper Khmer font"""
1440        msg_box = QMessageBox(parent)
1441        msg_box.setIcon(icon)
1442        msg_box.setWindowTitle(title)
1443        msg_box.setText(text)
```

```
1444
1445            # Apply font to message box and all children
1446            self.apply_font(msg_box, self.font_size)
1447
1448            if buttons:
1449                msg_box.setStandardButtons(buttons)
1450
1451            return msg_box
1452
1453    class DictionaryTableModel(QAbstractTableModel):
1454        """Enhanced table model with expert system features"""
1455
1456        def __init__(self, data=None):
1457            super().__init__()
1458            self.headers = ["ID", "English", "Khmer", "Type", "Definition", "Example"]
1459            self._data = data or []
1460
1461        def rowCount(self, parent=QModelIndex()):
1462            return len(self._data)
1463
1464        def columnCount(self, parent=QModelIndex()):
1465            return len(self.headers)
1466
1467        def data(self, index, role=Qt.ItemDataRole.DisplayRole):
1468            if not index.isValid():
1469                return QVariant()
1470
1471            if role == Qt.ItemDataRole.DisplayRole:
1472                row = index.row()
1473                col = index.column()
1474                if row < len(self._data):
1475                    data_item = self._data[row]
1476                    if col < len(self.headers):
1477                        # Map columns to basic data fields (keep standard view)
1478                        field_mapping = [0, 1, 2, 3, 4, 5]  # id, english, khmer, type, def, example
1479                        if col < len(field_mapping) and field_mapping[col] < len(data_item):
1480                            value = data_item[field_mapping[col]]
1481                            return str(value) if value is not None else ""
1482
1483            return QVariant()
1484
```

```
    def headerData(self, section, orientation, role=Qt.ItemDataRole.DisplayRole):
        if role == Qt.ItemDataRole.DisplayRole and orientation == Qt.Orientation.Horizontal:
            if section < len(self.headers):
                return self.headers[section]
        return QVariant()

    def update_data(self, new_data):
        """Update the model with new data"""
        self.beginResetModel()
        self._data = new_data or []
        self.endResetModel()

    def get_row_data(self, row):
        """Get complete data for a specific row"""
        if 0 <= row < len(self._data):
            return self._data[row]
        return None

    def add_row(self, row_data):
        """Add a new row to the model"""
        self.beginInsertRows(QModelIndex(), len(self._data), len(self._data))
        self._data.append(row_data)
        self.endInsertRows()

    def remove_row(self, row):
        """Remove a row from the model"""
        if 0 <= row < len(self._data):
            self.beginRemoveRows(QModelIndex(), row, row)
            del self._data[row]
            self.endRemoveRows()
            return True
        return False

class WordDetailsDialog(QDialog):
    """Enhanced dialog with AI explanations but standard UI"""

    def __init__(self, word_data, font_manager, ai_generator, parent=None):
        super().__init__(parent)
        self.font_manager = font_manager
        self.word_data = word_data
        self.ai_generator = ai_generator
        self.init_ui()

```

```
    def init_ui(self):
        self.setWindowTitle("Word Details with AI Analysis")
        self.setModal(True)
        self.resize(700, 550)

        # Apply font to dialog
        self.font_manager.apply_font(self)

        layout = QVBoxLayout()

        if self.word_data:
            # Create scrollable content area
            scroll = QScrollArea()
            scroll_widget = QWidget()
            scroll_layout = QVBoxLayout()

            word_id, english, khmer, word_type = self.word_data[0], self.word_data[1], self.word_data[2], self.word_data[3]
            definition, example = self.word_data[4], self.word_data[5]

            # Basic info
            basic_info = QTextBrowser()
            self.font_manager.apply_font(basic_info)
            basic_info.setMaximumHeight(150)

            basic_content = f"""
            <h2 style='color: #2E7D32;'>{english.title()} ↔ {khmer}</h2>
            <p><strong>Word ID:</strong> {word_id}</p>
            <p><strong>Type:</strong> {word_type.title()}</p>
            <p><strong>Definition:</strong> {definition or 'No definition provided'}</p>
            <p><strong>Example:</strong> {example or 'No example provided'}</p>
            """
            basic_info.setHtml(basic_content)

            # AI explanation
            ai_info = QTextBrowser()
            self.font_manager.apply_font(ai_info)

            ai_explanation = self.ai_generator.generate_explanation(self.word_data)
            ai_info.setHtml(ai_explanation)

            scroll_layout.addWidget(basic_info)
            scroll_layout.addWidget(ai_info)
            scroll_widget.setLayout(scroll_layout)
            scroll.setWidget(scroll_widget)
```

```
            scroll.setWidgetResizable(True)

            layout.addWidget(scroll)

        # Buttons
        button_box = QDialogButtonBox(QDialogButtonBox.StandardButton.Ok)
        self.font_manager.apply_font(button_box)
        button_box.accepted.connect(self.accept)
        layout.addWidget(button_box)

        self.setLayout(layout)

# Enhanced Translator Widget with Expert System but Standard UI
class TranslatorWidget(QWidget):
    word_searched = pyqtSignal(str, str, bool)

    def __init__(self, db, font_manager, expert_engine, user_profile, ai_generator):
        super().__init__()
        self.db = db
        self.font_manager = font_manager
        self.expert_engine = expert_engine
        self.user_profile = user_profile
        self.ai_generator = ai_generator
        self.init_ui()

    def init_ui(self):
        # Apply font to entire widget
        self.font_manager.apply_font(self)

        layout = QVBoxLayout()

        # Title section
        title_frame = QFrame()
        title_frame.setFrameStyle(QFrame.Shape.StyledPanel)
        title_layout = QVBoxLayout()

        title = QLabel("Khmer-English Dictionary (AI-Enhanced)")
        self.font_manager.apply_font(title, bold=True)
        title.setAlignment(Qt.AlignmentFlag.AlignCenter)

        subtitle = QLabel("វចនានុក្រមអង់គ្លេស-ខ្មែរ (ជាក់បញ្ជាសិប្បនិម្មិត)")
        self.font_manager.apply_font(subtitle)
```

```
        subtitle.setAlignment(Qt.AlignmentFlag.AlignCenter)

        title_layout.addWidget(title)
        title_layout.addWidget(subtitle)
        title_frame.setLayout(title_layout)
        layout.addWidget(title_frame)

        # Expert suggestions section (scrollable)
        suggestions_group = QGroupBox("Expert System Recommendations")
        self.font_manager.apply_font(suggestions_group, bold=True)
        suggestions_group.setFixedHeight(80)
        suggestions_layout = QVBoxLayout()

        self.suggestions_scroll = QScrollArea()
        self.suggestions_display = QLabel("Welcome! Start searching to get personalized recommendations.")
        self.suggestions_display.setWordWrap(True)
        self.font_manager.apply_font(self.suggestions_display)
        self.suggestions_scroll.setWidget(self.suggestions_display)
        self.suggestions_scroll.setWidgetResizable(True)
        self.suggestions_scroll.setMaximumHeight(60)

        suggestions_layout.addWidget(self.suggestions_scroll)
        suggestions_group.setLayout(suggestions_layout)
        layout.addWidget(suggestions_group)

        # Search section with scroll area
        search_group = QGroupBox("Search Translation")
        search_group.setFixedHeight(480)
        self.font_manager.apply_font(search_group, bold=True)
        search_layout = QVBoxLayout()

        # Search controls with voice search button integrated
        controls_layout = QHBoxLayout()

        direction_label = QLabel("Direction:")
        self.font_manager.apply_font(direction_label)

        self.search_combo = QComboBox()
        self.search_combo.addItems(["English → Khmer", "Khmer → English", "Smart Search"])
        self.font_manager.apply_font(self.search_combo)

        word_label = QLabel("Word:")
        self.font_manager.apply_font(word_label)
```

```
1657
1658        self.search_input = QLineEdit()
1659        self.search_input.setPlaceholderText("Type word to translate...")
1660        self.font_manager.apply_font(self.search_input)
1661        self.search_input.returnPressed.connect(self.search_word)
1662
1663        # Voice search button (between input and search button)
1664        self.voice_button = QPushButton("🎤")
1665        self.font_manager.apply_font(self.voice_button)
1666        self.voice_button.clicked.connect(self.start_voice_search)
1667        self.voice_button.setToolTip("Click and speak your search term")
1668        self.voice_button.setFixedWidth(50)
1669        self.voice_button.setStyleSheet("QPushButton { min-height: 25px; }")
1670
1671        self.search_button = QPushButton("Search")
1672        self.font_manager.apply_font(self.search_button)
1673        self.search_button.clicked.connect(self.search_word)
1674        self.search_button.setDefault(True)
1675
1676        controls_layout.addWidget(direction_label)
1677        controls_layout.addWidget(self.search_combo)
1678        controls_layout.addWidget(word_label)
1679        controls_layout.addWidget(self.search_input)
1680        controls_layout.addWidget(self.voice_button)
1681        controls_layout.addWidget(self.search_button)
1682
1683        search_layout.addLayout(controls_layout)
1684
1685        # Voice search status (separate line)
1686        self.voice_status_label = QLabel("Ready for voice search")
1687        self.font_manager.apply_font(self.voice_status_label)
1688        self.voice_status_label.setStyleSheet("color: #666; font-style: italic; margin-left: 5px;")
1689        search_layout.addWidget(self.voice_status_label)
1690
1691        # Results display with scroll
1692        results_label = QLabel("Results:")
1693        self.font_manager.apply_font(results_label)
1694
1695        self.results_scroll = QScrollArea()
1696        self.results_display = QTextBrowser()
1697        self.font_manager.apply_font(self.results_display)
1698        self.results_scroll.setWidget(self.results_display)
1699        self.results_scroll.setWidgetResizable(True)
```

```
1700            self.results_scroll.setMaximumHeight(380)
1701
1702            search_layout.addWidget(results_label)
1703            search_layout.addWidget(self.results_scroll)
1704
1705            search_group.setLayout(search_layout)
1706            layout.addWidget(search_group)
1707
1708            # Action buttons
1709            button_layout = QHBoxLayout()
1710
1711            self.clear_button = QPushButton("Clear")
1712            self.font_manager.apply_font(self.clear_button)
1713            self.clear_button.clicked.connect(self.clear_search)
1714
1715            self.random_button = QPushButton("Random Word")
1716            self.font_manager.apply_font(self.random_button)
1717            self.random_button.clicked.connect(self.show_smart_random_word)
1718
1719            self.similar_button = QPushButton("Find Similar")
1720            self.font_manager.apply_font(self.similar_button)
1721            self.similar_button.clicked.connect(self.find_similar_words)
1722
1723            button_layout.addWidget(self.clear_button)
1724            button_layout.addWidget(self.random_button)
1725            button_layout.addWidget(self.similar_button)
1726            button_layout.addStretch()
1727
1728            layout.addLayout(button_layout)
1729            layout.addStretch()
1730
1731            self.setLayout(layout)
1732
1733            # Initialize voice search components
1734            self.voice_thread = None
1735            self.init_voice_search()
1736
1737            # Update expert suggestions on startup
1738            self.update_expert_suggestions()
1739
1740        def init_voice_search(self):
1741            """Initialize voice search functionality"""
```

```
        # Check if voice recognition is available
        if not SPEECH_RECOGNITION_AVAILABLE:
            self.voice_button.setEnabled(False)
            self.voice_button.setToolTip("Install speech recognition: pip install SpeechRecognition pyaudio")
            self.voice_status_label.setText("Voice search requires: pip install SpeechRecognition pyaudio")
            self.voice_status_label.setStyleSheet("color: #f44336;")
        else:
            self.voice_button.setEnabled(True)
            self.voice_button.setToolTip("Click and speak your search term")
            self.voice_status_label.setText("Ready for voice search")
            self.voice_status_label.setStyleSheet("color: #666; font-style: italic;")

    def start_voice_search(self):
        """Start voice recognition"""
        if not SPEECH_RECOGNITION_AVAILABLE:
            msg = self.font_manager.create_message_box(
                self, QMessageBox.Icon.Warning,
                "Voice Search Not Available",
                "Voice search requires additional packages:\n\npip install SpeechRecognition pyaudio\n\nPlease install them and restart the application."
            )
            msg.exec()
            return

        if self.voice_thread and self.voice_thread.isRunning():
            self.stop_voice_search()
            return

        try:
            self.voice_thread = VoiceSearchThread()
            self.voice_thread.speech_recognized.connect(self.on_voice_search_completed)
            self.voice_thread.speech_error.connect(self.on_voice_search_error)
            self.voice_thread.recording_started.connect(self.on_voice_recording_started)
            self.voice_thread.recording_stopped.connect(self.on_voice_recording_stopped)

            self.voice_thread.start()
        except Exception as e:
            self.on_voice_search_error(f"Failed to start voice recognition: {str(e)}")

    def stop_voice_search(self):
        """Stop voice recognition"""
        if self.voice_thread:
            self.voice_thread.stop_listening()
            self.voice_thread.wait(1000)  # Wait up to 1 second
```

```
            if self.voice_thread.isRunning():
                self.voice_thread.terminate()
            self.on_voice_recording_stopped()

    def on_voice_recording_started(self):
        """Handle recording start"""
        self.voice_button.setText("🔴")
        self.voice_button.setStyleSheet("background-color: #ffebee; color: #f44336;")
        self.voice_status_label.setText("🎤 Listening... Speak now!")
        self.voice_status_label.setStyleSheet("color: #f44336; font-weight: bold;")

    def on_voice_recording_stopped(self):
        """Handle recording stop"""
        self.voice_button.setText("🎤")
        self.voice_button.setStyleSheet("")
        self.voice_status_label.setText("Processing speech...")
        self.voice_status_label.setStyleSheet("color: #ff9800; font-style: italic;")

    def on_voice_search_error(self, error_message):
        """Handle speech recognition error"""
        self.voice_status_label.setText(f"❌ {error_message}")
        self.voice_status_label.setStyleSheet("color: #f44336;")

        # Reset status after 5 seconds
        QTimer.singleShot(5000, self.reset_voice_status)

    def reset_voice_status(self):
        """Reset voice status to default"""
        self.voice_status_label.setText("Ready for voice search")
        self.voice_status_label.setStyleSheet("color: #666; font-style: italic;")

    def on_voice_search_completed(self, recognized_text):
        """Handle completed voice search"""
        # Clean up the recognized text
        cleaned_text = recognized_text.strip().lower()

        # Set the text in search input
        self.search_input.setText(cleaned_text)

        # Update status
        self.voice_status_label.setText(f"✓ Recognized: '{recognized_text}'")
        self.voice_status_label.setStyleSheet("color: #4caf50; font-weight: bold;")
```

```
1827
1828            # Automatically perform the search
1829            self.search_word()
1830
1831            # Reset status after 3 seconds
1832            QTimer.singleShot(3000, self.reset_voice_status)
1833
1834        def search_word(self):
1835            search_term = self.search_input.text().strip()
1836            if not search_term:
1837                msg = self.font_manager.create_message_box(
1838                    self, QMessageBox.Icon.Warning,
1839                    "Input Error",
1840                    "Please enter a word to search!"
1841                )
1842                msg.exec()
1843                return
1844
1845            search_direction = self.search_combo.currentText()
1846
1847            # Determine search type
1848            if search_direction.startswith("English"):
1849                search_type = "english"
1850            elif search_direction.startswith("Khmer"):
1851                search_type = "khmer"
1852            else:  # Smart search
1853                search_type = "smart"
1854
1855            # Perform search
1856            results = []
1857            if search_type == "smart":
1858                # Try both English and Khmer
1859                results = self.db.read_word(search_term, "english")
1860                if not results:
1861                    results = self.db.read_word(search_term, "khmer")
1862            else:
1863                results = self.db.read_word(search_term, search_type)
1864
1865            # Record search in user profile
1866            found = len(results) > 0
1867            self.user_profile.record_search(search_term, search_type, found)
1868
```

```
        # Update expert system with search context
        self.expert_engine.add_fact("search_term", search_term)
        self.expert_engine.add_fact("search_type", search_type)
        self.expert_engine.add_fact("not_found", not found)
        self.expert_engine.add_fact("word_length", len(search_term))

        # Add user profile facts
        user_facts = self.user_profile.get_user_facts()
        for key, value in user_facts.items():
            self.expert_engine.add_fact(key, value)

        if results:
            self.display_results(results)
            # Record word type for user modeling
            if results:
                word_type = results[0][3]  # word_type column
                self.user_profile.record_word_interaction(word_type)
        else:
            # Get AI-powered suggestions
            suggestions = self.db.get_smart_suggestions(search_term, search_type)
            self.display_no_results_with_suggestions(search_term, search_direction, suggestions)

        # Get expert system recommendations
        expert_suggestions = self.expert_engine.infer()
        self.display_expert_suggestions(expert_suggestions)

        # Emit signal
        self.word_searched.emit(search_term, search_type, found)

    def display_results(self, results):
        html_content = "<h3>Translation Results:</h3>"

        for result in results:
            word_id, english, khmer, word_type = result[0], result[1], result[2], result[3]
            definition, example = result[4], result[5]

            html_content += f"""
            <div style='border: 1px solid #ccc; margin: 10px 0; padding: 10px; background-color: #f9f9f9;'>
                <h4>{english.title()} ↔ {khmer}</h4>
                <p><strong>Type:</strong> {word_type.title()} | <strong>ID:</strong> {word_id}</p>
                {f"<p><strong>Definition:</strong> {definition}</p>" if definition else ""}
                {f"<p><strong>Example:</strong> <em>{example}</em></p>" if example else ""}
                <p><small><em>Click 'View Details' in Dictionary Manager for AI analysis</em></small></p>
```

```
            </div>
            """

        self.results_display.setHtml(html_content)

    def display_no_results_with_suggestions(self, search_term, search_direction, suggestions):
        html_content = f"""
        <h3>No results found for '{search_term}'</h3>
        <p><strong>Search Details:</strong></p>
        <ul>
            <li>Search Direction: {search_direction}</li>
            <li>Search Term: {search_term}</li>
        </ul>
        """

        if suggestions:
            html_content += "<h4>AI Suggestions:</h4>"
            for suggestion in suggestions:
                word_id, english, khmer, word_type = suggestion[0], suggestion[1], suggestion[2], suggestion[3]
                definition = suggestion[4]

                html_content += f"""
                <div style='border-left: 3px solid #4CAF50; padding-left: 10px; margin: 8px 0;'>
                    <strong>{english} → {khmer}</strong> ({word_type})
                    {f"<br><small>{definition}</small>" if definition else ""}
                </div>
                """

        html_content += """
        <p><strong>Suggestions:</strong></p>
        <ul>
            <li>Check spelling</li>
            <li>Try simpler words</li>
            <li>Use the Dictionary Manager to add new words</li>
            <li>Try 'Find Similar' button</li>
        </ul>
        """

        self.results_display.setHtml(html_content)

    def display_expert_suggestions(self, suggestions):
        """Display expert system suggestions"""
```

```
        if not suggestions:
            return

        suggestion_text = " | ".join(suggestions[:2])  # Show first 2 suggestions
        self.suggestions_display.setText(f"Expert Advice: {suggestion_text}")

    def update_expert_suggestions(self):
        """Update expert system suggestions based on user profile"""
        user_recommendations = self.user_profile.get_recommendations()
        if user_recommendations:
            self.suggestions_display.setText(f"Personal: {' | '.join(user_recommendations[:2])}")

    def find_similar_words(self):
        """Find words similar to current input"""
        search_term = self.search_input.text().strip()
        if not search_term:
            msg = self.font_manager.create_message_box(
                self, QMessageBox.Icon.Warning,
                "Input Required",
                "Please enter a word to find similar words!"
            )
            msg.exec()
            return

        similar_words = self.db.find_similar_words(search_term, 5)

        if similar_words:
            html_content = f"<h3>Words similar to '{search_term}':</h3>"

            for word, similarity in similar_words:
                # Try to get full word data
                word_results = self.db.read_word(word, "english")
                if not word_results:
                    word_results = self.db.read_word(word, "khmer")

                if word_results:
                    result = word_results[0]
                    english, khmer, word_type = result[1], result[2], result[3]
                    similarity_percent = int(similarity * 100)

                    html_content += f"""
                    <div style='border: 1px solid #ddd; margin: 8px 0; padding: 10px;
                                background-color: #f9f9f9; border-radius: 5px;'>
```

```
                    <strong>{english} → {khmer}</strong> ({word_type})
                    <br><small>Similarity: {similarity_percent}%</small>
                </div>
                """

            self.results_display.setHtml(html_content)
        else:
            self.results_display.setHtml(f"<h3>No similar words found for '{search_term}'</h3>")

    def show_smart_random_word(self):
        """Show random word adapted to user level"""
        user_facts = self.user_profile.get_user_facts()
        difficulty_preference = user_facts.get("user_level", "beginner")

        # Get random words and filter by difficulty if possible
        random_words = self.db.get_random_words(10)

        # Simple filtering based on word length as proxy for difficulty
        if difficulty_preference == "beginner":
            filtered_words = [w for w in random_words if len(w[1]) <= 6]  # Shorter English words
        elif difficulty_preference == "advanced":
            filtered_words = [w for w in random_words if len(w[1]) > 8]   # Longer English words
        else:
            filtered_words = random_words

        if not filtered_words:
            filtered_words = random_words

        if filtered_words:
            selected_word = filtered_words[0]
            self.display_results([selected_word])
            self.search_input.setText(selected_word[1])  # Set English word in input

    def clear_search(self):
        self.search_input.clear()
        self.results_display.clear()
        self.search_input.setFocus()
        self.update_expert_suggestions()

# Enhanced Dictionary Manager with Standard UI but Expert Features
class DictionaryManagerWidget(QWidget):
    word_added = pyqtSignal(str)
```

```
2039        word_updated = pyqtSignal(int)
2040        word_deleted = pyqtSignal(int)
2041
2042        def __init__(self, db, font_manager, ai_generator):
2043            super().__init__()
2044            self.db = db
2045            self.font_manager = font_manager
2046            self.ai_generator = ai_generator
2047            self.current_edit_id = None
2048            self.table_model = DictionaryTableModel()
2049            self.init_ui()
2050            self.refresh_dictionary()
2051
2052        def init_ui(self):
2053            # Apply font to entire widget
2054            self.font_manager.apply_font(self)
2055
2056            layout = QVBoxLayout()
2057
2058            # Create splitter for form and table
2059            splitter = QSplitter(Qt.Orientation.Vertical)
2060
2061            # Form section with scroll area
2062            form_widget = QWidget()
2063            form_layout = QVBoxLayout()
2064
2065            form_group = QGroupBox("Add/Edit Dictionary Entry (AI-Enhanced)")
2066            self.font_manager.apply_font(form_group, bold=True)
2067            form_group.setFixedHeight(170)
2068            form_group_layout = QVBoxLayout()
2069
2070            # CREATE form inputs using single QHBoxLayout (original style)
2071            inputs_layout = QHBoxLayout()
2072
2073            # English Word
2074            english_layout = QVBoxLayout()
2075            english_label = QLabel("English Word:")
2076            self.font_manager.apply_font(english_label)
2077            self.english_input = QLineEdit()
2078            self.english_input.setPlaceholderText("e.g., computer")
2079            self.font_manager.apply_font(self.english_input)
2080            english_layout.addWidget(english_label)
2081            english_layout.addWidget(self.english_input)
2082
```

```
        # Khmer Word
        khmer_layout = QVBoxLayout()
        khmer_label = QLabel("Khmer Word:")
        self.font_manager.apply_font(khmer_label)
        self.khmer_input = QLineEdit()
        self.khmer_input.setPlaceholderText("e.g., កុំព្យូទ័រ")
        self.font_manager.apply_font(self.khmer_input)
        khmer_layout.addWidget(khmer_label)
        khmer_layout.addWidget(self.khmer_input)

        # Word Type
        type_layout = QVBoxLayout()
        type_label = QLabel("Word Type:")
        self.font_manager.apply_font(type_label)
        self.type_combo = QComboBox()
        self.type_combo.addItems(["noun", "verb", "adjective", "adverb", "greeting", "expression", "response"])
        self.font_manager.apply_font(self.type_combo)
        type_layout.addWidget(type_label)
        type_layout.addWidget(self.type_combo)

        # Definition
        definition_layout = QVBoxLayout()
        definition_label = QLabel("Definition:")
        self.font_manager.apply_font(definition_label)
        self.definition_input = QLineEdit()
        self.definition_input.setPlaceholderText("Brief definition...")
        self.font_manager.apply_font(self.definition_input)
        definition_layout.addWidget(definition_label)
        definition_layout.addWidget(self.definition_input)

        # Example
        example_layout = QVBoxLayout()
        example_label = QLabel("Example:")
        self.font_manager.apply_font(example_label)
        self.example_input = QLineEdit()
        self.example_input.setPlaceholderText("Example sentence...")
        self.font_manager.apply_font(self.example_input)
        example_layout.addWidget(example_label)
        example_layout.addWidget(self.example_input)

        # Add all input groups to single horizontal layout
        inputs_layout.addLayout(english_layout)
        inputs_layout.addLayout(khmer_layout)
```

```
2126            inputs_layout.addLayout(type_layout)
2127            inputs_layout.addLayout(definition_layout)
2128            inputs_layout.addLayout(example_layout)
2129
2130            # CRUD operation buttons
2131            button_layout = QHBoxLayout()
2132
2133            # CREATE button
2134            self.add_button = QPushButton("Create Word")
2135            self.font_manager.apply_font(self.add_button)
2136            self.add_button.clicked.connect(self.create_word)
2137            self.add_button.setDefault(True)
2138
2139            # UPDATE button
2140            self.update_button = QPushButton("Update Word")
2141            self.font_manager.apply_font(self.update_button)
2142            self.update_button.clicked.connect(self.update_word)
2143            self.update_button.setVisible(False)
2144
2145            self.cancel_button = QPushButton("Cancel Edit")
2146            self.font_manager.apply_font(self.cancel_button)
2147            self.cancel_button.clicked.connect(self.cancel_edit)
2148            self.cancel_button.setVisible(False)
2149
2150            self.clear_button = QPushButton("Clear Form")
2151            self.font_manager.apply_font(self.clear_button)
2152            self.clear_button.clicked.connect(self.clear_form)
2153
2154            self.ai_assist_button = QPushButton("AI Assist")
2155            self.font_manager.apply_font(self.ai_assist_button)
2156            self.ai_assist_button.clicked.connect(self.ai_assist)
2157
2158            button_layout.addWidget(self.add_button)
2159            button_layout.addWidget(self.update_button)
2160            button_layout.addWidget(self.cancel_button)
2161            button_layout.addWidget(self.clear_button)
2162            button_layout.addWidget(self.ai_assist_button)
2163            button_layout.addStretch()
2164
2165            form_group_layout.addLayout(inputs_layout)
2166            form_group_layout.addLayout(button_layout)
2167            form_group.setLayout(form_group_layout)
2168            form_layout.addWidget(form_group)
```

```
2169        form_widget.setLayout(form_layout)
2170
2171        # Table section with scroll area
2172        table_widget = QWidget()
2173        table_layout = QVBoxLayout()
2174
2175        table_group = QGroupBox("Dictionary Entries")
2176        self.font_manager.apply_font(table_group, bold=True)
2177        table_group_layout = QVBoxLayout()
2178
2179        # Filter for READ operations
2180        filter_layout = QHBoxLayout()
2181        filter_label = QLabel("Filter:")
2182        self.font_manager.apply_font(filter_label)
2183
2184        self.filter_input = QLineEdit()
2185        self.filter_input.setPlaceholderText("Filter entries...")
2186        self.font_manager.apply_font(self.filter_input)
2187        self.filter_input.textChanged.connect(self.filter_dictionary)
2188
2189        self.refresh_button = QPushButton("Refresh")
2190        self.font_manager.apply_font(self.refresh_button)
2191        self.refresh_button.clicked.connect(self.refresh_dictionary)
2192
2193        filter_layout.addWidget(filter_label)
2194        filter_layout.addWidget(self.filter_input)
2195        filter_layout.addWidget(self.refresh_button)
2196
2197        # Table view for displaying data
2198        self.table_view = QTableView()
2199        self.font_manager.apply_font(self.table_view)
2200        self.table_view.setModel(self.table_model)
2201        self.table_view.setSelectionBehavior(QAbstractItemView.SelectionBehavior.SelectRows)
2202        self.table_view.setAlternatingRowColors(True)
2203
2204        # Configure headers
2205        header = self.table_view.horizontalHeader()
2206        header.setStretchLastSection(True)
2207        header.setSectionResizeMode(QHeaderView.ResizeMode.Interactive)
2208        self.font_manager.apply_font(header, bold=True)
2209
```

```
        # CRUD action buttons
        table_button_layout = QHBoxLayout()

        # READ operation - View details with AI
        self.view_button = QPushButton("View Details")
        self.font_manager.apply_font(self.view_button)
        self.view_button.clicked.connect(self.view_selected_word)

        # UPDATE operation - Edit
        self.edit_button = QPushButton("Edit Selected")
        self.font_manager.apply_font(self.edit_button)
        self.edit_button.clicked.connect(self.edit_selected_word)

        # DELETE operation - Delete
        self.delete_button = QPushButton("Delete Selected")
        self.font_manager.apply_font(self.delete_button)
        self.delete_button.clicked.connect(self.delete_selected_word)

        self.stats_label = QLabel()
        self.font_manager.apply_font(self.stats_label)

        table_button_layout.addWidget(self.view_button)
        table_button_layout.addWidget(self.edit_button)
        table_button_layout.addWidget(self.delete_button)
        table_button_layout.addStretch()
        table_button_layout.addWidget(self.stats_label)

        table_group_layout.addLayout(filter_layout)
        table_group_layout.addWidget(self.table_view)
        table_group_layout.addLayout(table_button_layout)

        table_group.setLayout(table_group_layout)
        table_layout.addWidget(table_group)
        table_widget.setLayout(table_layout)

        # Add to splitter
        splitter.addWidget(form_widget)
        splitter.addWidget(table_widget)
        splitter.setSizes([300, 500])

        layout.addWidget(splitter)
        self.setLayout(layout)
```

```
2252
2253    def ai_assist(self):
2254        """AI assistance for filling form fields"""
2255        english_word = self.english_input.text().strip()
2256        if not english_word:
2257            msg = self.font_manager.create_message_box(
2258                self, QMessageBox.Icon.Information,
2259                "AI Assistant",
2260                "AI Tips for creating dictionary entries:\n\n" +
2261                "• Use clear, simple definitions\n" +
2262                "• Include cultural context when relevant\n" +
2263                "• Choose appropriate word type\n" +
2264                "• Use descriptive examples\n\n" +
2265                "Enter an English word first to get specific suggestions!"
2266            )
2267            msg.exec()
2268            return
2269
2270        # Simple AI assistance based on word patterns
2271        suggestions = []
2272
2273        # Word type suggestions
2274        if english_word.endswith('ing'):
2275            suggestions.append("Consider 'verb' as word type")
2276            self.type_combo.setCurrentText("verb")
2277        elif english_word.endswith('ly'):
2278            suggestions.append("Consider 'adverb' as word type")
2279            self.type_combo.setCurrentText("adverb")
2280        elif english_word in ['hello', 'goodbye', 'hi', 'bye']:
2281            suggestions.append("Consider 'greeting' as word type")
2282            self.type_combo.setCurrentText("greeting")
2283
2284        if suggestions:
2285            suggestion_text = "\n• ".join(suggestions)
2286            msg = self.font_manager.create_message_box(
2287                self, QMessageBox.Icon.Information,
2288                "AI Suggestions Applied",
2289                f"AI has analyzed '{english_word}' and suggests:\n\n• {suggestion_text}\n\n" +
2290                "Review and modify as needed!"
2291            )
2292            msg.exec()
2293        else:
2294            msg = self.font_manager.create_message_box(
2295                self, QMessageBox.Icon.Information,
```

```
            "AI Assistant",
            f"No specific suggestions for '{english_word}', but here are general tips:\n\n" +
            "• Check word type carefully\n" +
            "• Consider cultural context\n" +
            "• Add helpful examples\n" +
            "• Use appropriate difficulty level"
        )
        msg.exec()

    def create_word(self):
        """Enhanced CREATE operation"""
        english = self.english_input.text().strip()
        khmer = self.khmer_input.text().strip()
        word_type = self.type_combo.currentText()
        definition = self.definition_input.text().strip()
        example = self.example_input.text().strip()

        if not english or not khmer:
            msg = self.font_manager.create_message_box(
                self, QMessageBox.Icon.Warning,
                "Input Error",
                "Please enter both English and Khmer words!"
            )
            msg.exec()
            return

        try:
            word_id = self.db.create_word(english, khmer, word_type, definition, example)
            self.clear_form()
            self.refresh_dictionary()
            self.word_added.emit(english)

            msg = self.font_manager.create_message_box(
                self, QMessageBox.Icon.Information,
                "Success",
                f"Word '{english}' → '{khmer}' created successfully!\nWord ID: {word_id}"
            )
            msg.exec()

        except ValueError as e:
            msg = self.font_manager.create_message_box(
                self, QMessageBox.Icon.Warning,
                "Error",
```

```
            str(e)
        )
        msg.exec()

    def view_selected_word(self):
        """Enhanced READ operation with AI analysis"""
        selection = self.table_view.selectionModel().selectedRows()
        if not selection:
            msg = self.font_manager.create_message_box(
                self, QMessageBox.Icon.Warning,
                "No Selection",
                "Please select a word to view AI analysis!"
            )
            msg.exec()
            return

        row = selection[0].row()
        word_data = self.table_model.get_row_data(row)
        if word_data:
            dialog = WordDetailsDialog(word_data, self.font_manager, self.ai_generator, self)
            dialog.exec()

    def update_word(self):
        """Enhanced UPDATE operation"""
        if self.current_edit_id is None:
            return

        english = self.english_input.text().strip()
        khmer = self.khmer_input.text().strip()
        word_type = self.type_combo.currentText()
        definition = self.definition_input.text().strip()
        example = self.example_input.text().strip()

        if not english or not khmer:
            msg = self.font_manager.create_message_box(
                self, QMessageBox.Icon.Warning,
                "Input Error",
                "Please enter both English and Khmer words!"
            )
            msg.exec()
            return

```

```
2381            try:
2382                self.db.update_word(self.current_edit_id, english, khmer, word_type, definition, example)
2383                self.cancel_edit()
2384                self.refresh_dictionary()
2385                self.word_updated.emit(self.current_edit_id)
2386
2387                msg = self.font_manager.create_message_box(
2388                    self, QMessageBox.Icon.Information,
2389                    "Success",
2390                    f"Word updated successfully!\nNew values: '{english}' → '{khmer}'"
2391                )
2392                msg.exec()
2393            except ValueError as e:
2394                msg = self.font_manager.create_message_box(
2395                    self, QMessageBox.Icon.Warning,
2396                    "Update Error",
2397                    str(e)
2398                )
2399                msg.exec()
2400
2401        def edit_selected_word(self):
2402            """Prepare UPDATE operation - Load selected word into form for editing"""
2403            selection = self.table_view.selectionModel().selectedRows()
2404            if not selection:
2405                msg = self.font_manager.create_message_box(
2406                    self, QMessageBox.Icon.Warning,
2407                    "No Selection",
2408                    "Please select a word to edit!"
2409                )
2410                msg.exec()
2411                return
2412
2413            row = selection[0].row()
2414            word_data = self.table_model.get_row_data(row)
2415            if word_data:
2416                word_id, english, khmer, word_type, definition, example = word_data[:6]
2417
2418                self.current_edit_id = word_id
2419                self.english_input.setText(english)
2420                self.khmer_input.setText(khmer)
2421                self.type_combo.setCurrentText(word_type)
2422                self.definition_input.setText(definition or "")
2423                self.example_input.setText(example or "")
```

```
2424
2425            # Switch to update mode
2426            self.add_button.setVisible(False)
2427            self.update_button.setVisible(True)
2428            self.cancel_button.setVisible(True)
2429
2430            # Focus on first input
2431            self.english_input.setFocus()
2432
2433    def delete_selected_word(self):
2434        """Enhanced DELETE operation"""
2435        selection = self.table_view.selectionModel().selectedRows()
2436        if not selection:
2437            msg = self.font_manager.create_message_box(
2438                self, QMessageBox.Icon.Warning, "No Selection", "Please select a word to delete!"
2439            )
2440            msg.exec()
2441            return
2442
2443        row = selection[0].row()
2444        word_data = self.table_model.get_row_data(row)
2445        if word_data:
2446            word_id, english, khmer = word_data[0], word_data[1], word_data[2]
2447
2448            msg = self.font_manager.create_message_box(
2449                self, QMessageBox.Icon.Question,
2450                "Confirm Delete",
2451                f"Are you sure you want to delete this word?\n\n'{english}' → '{khmer}'\n\nThis action cannot be undone.",
2452                QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
2453            )
2454
2455            if msg.exec() == QMessageBox.StandardButton.Yes:
2456                try:
2457                    if self.db.delete_word(word_id):
2458                        self.refresh_dictionary()
2459                        self.word_deleted.emit(word_id)
2460
2461                        success_msg = self.font_manager.create_message_box(
2462                            self, QMessageBox.Icon.Information,
2463                            "Deleted", f"Word '{english}' → '{khmer}' deleted successfully!"
2464                        )
2465                        success_msg.exec()
```

```
                else:
                    error_msg = self.font_manager.create_message_box(
                        self, QMessageBox.Icon.Warning,
                        "Delete Failed", "Failed to delete the word. It may have already been removed."
                    )
                    error_msg.exec()
            except ValueError as e:
                error_msg = self.font_manager.create_message_box(
                    self, QMessageBox.Icon.Critical, "Delete Error", str(e)
                )
                error_msg.exec()

    def refresh_dictionary(self):
        """Refresh the table view with current database data"""
        words = self.db.read_all_words()
        self.table_model.update_data(words)
        self.stats_label.setText(f"Total entries: {len(words)}")

    def filter_dictionary(self):
        """Filter dictionary entries based on search text"""
        filter_text = self.filter_input.text().strip().lower()

        if not filter_text:
            self.refresh_dictionary()
            return

        words = self.db.read_all_words()
        filtered_words = []

        for word in words:
            word_id, english, khmer, word_type, definition, example = word[:6]
            if (filter_text in english.lower() or
                filter_text in khmer or
                filter_text in word_type.lower() or
                filter_text in (definition or "").lower()):
                filtered_words.append(word)

        self.table_model.update_data(filtered_words)
        self.stats_label.setText(f"Showing {len(filtered_words)} of {len(words)} entries")

    def cancel_edit(self):
        """Cancel edit mode and return to create mode"""
        self.current_edit_id = None
```

```
2509            self.clear_form()
2510
2511            # Switch back to create mode
2512            self.add_button.setVisible(True)
2513            self.update_button.setVisible(False)
2514            self.cancel_button.setVisible(False)
2515
2516        def clear_form(self):
2517            """Clear all form inputs"""
2518            self.english_input.clear()
2519            self.khmer_input.clear()
2520            self.type_combo.setCurrentIndex(0)
2521            self.definition_input.clear()
2522            self.example_input.clear()
2523            self.english_input.setFocus()
2524
2525    # Enhanced Statistics Widget with Standard UI
2526    class StatisticsWidget(QWidget):
2527        def __init__(self, db, font_manager, user_profile):
2528            super().__init__()
2529            self.db = db
2530            self.font_manager = font_manager
2531            self.user_profile = user_profile
2532            self.search_count = 0
2533            self.init_ui()
2534            self.update_stats()
2535
2536        def init_ui(self):
2537            # Apply font to entire widget
2538            self.font_manager.apply_font(self)
2539
2540            layout = QVBoxLayout()
2541
2542            # Statistics section with scroll
2543            stats_group = QGroupBox("Dictionary Statistics (AI-Enhanced)")
2544            self.font_manager.apply_font(stats_group, bold=True)
2545            stats_layout = QVBoxLayout()
2546
2547            stats_scroll = QScrollArea()
2548            stats_content = QWidget()
2549            stats_content_layout = QVBoxLayout()
2550
```

```
2551            self.total_words_label = QLabel()
2552            self.font_manager.apply_font(self.total_words_label)
2553
2554            self.word_types_label = QLabel()
2555            self.font_manager.apply_font(self.word_types_label)
2556            self.word_types_label.setWordWrap(True)
2557
2558            self.user_stats_label = QLabel()
2559            self.font_manager.apply_font(self.user_stats_label)
2560            self.user_stats_label.setWordWrap(True)
2561
2562            self.searches_label = QLabel()
2563            self.font_manager.apply_font(self.searches_label)
2564
2565            stats_content_layout.addWidget(self.total_words_label)
2566            stats_content_layout.addWidget(self.word_types_label)
2567            stats_content_layout.addWidget(self.user_stats_label)
2568            stats_content_layout.addWidget(self.searches_label)
2569
2570            stats_content.setLayout(stats_content_layout)
2571            stats_scroll.setWidget(stats_content)
2572            stats_scroll.setWidgetResizable(True)
2573            stats_scroll.setMaximumHeight(150)
2574
2575            stats_layout.addWidget(stats_scroll)
2576            stats_group.setLayout(stats_layout)
2577            layout.addWidget(stats_group)
2578
2579            # AI insights section with scroll
2580            insights_group = QGroupBox("AI Learning Insights")
2581            insights_group.setFixedHeight(180)
2582            self.font_manager.apply_font(insights_group, bold=True)
2583
2584            insights_layout = QVBoxLayout()
2585
2586            insights_scroll = QScrollArea()
2587            self.insights_display = QTextBrowser()
2588            self.font_manager.apply_font(self.insights_display)
2589            insights_scroll.setWidget(self.insights_display)
2590            insights_scroll.setWidgetResizable(True)
2591
2592            insights_layout.addWidget(insights_scroll)
```

```
2593        insights_group.setLayout(insights_layout)
2594        layout.addWidget(insights_group)
2595
2596        # Sample words with scroll
2597        sample_group = QGroupBox("Sample Khmer Words")
2598        sample_group.setFixedHeight(280)
2599        self.font_manager.apply_font(sample_group, bold=True)
2600
2601        sample_layout = QVBoxLayout()
2602
2603        sample_scroll = QScrollArea()
2604        self.sample_display = QTextBrowser()
2605        self.font_manager.apply_font(self.sample_display)
2606        sample_scroll.setWidget(self.sample_display)
2607        sample_scroll.setWidgetResizable(True)
2608
2609        sample_layout.addWidget(sample_scroll)
2610        sample_group.setLayout(sample_layout)
2611        layout.addWidget(sample_group)
2612
2613        # Buttons
2614        button_layout = QHBoxLayout()
2615
2616        update_btn = QPushButton("Update Statistics")
2617        self.font_manager.apply_font(update_btn)
2618        update_btn.clicked.connect(self.update_stats)
2619
2620        export_btn = QPushButton("📊 Export Data")
2621        self.font_manager.apply_font(export_btn)
2622        export_btn.clicked.connect(self.export_data)
2623        export_btn.setToolTip("Export dictionary data, analytics, and user profile in multiple formats (CSV, JSON, TXT, HTML)")
2624
2625        reset_btn = QPushButton("Reset Profile")
2626        self.font_manager.apply_font(reset_btn)
2627        reset_btn.clicked.connect(self.reset_profile)
2628
2629        button_layout.addWidget(update_btn)
2630        button_layout.addWidget(export_btn)
2631        button_layout.addWidget(reset_btn)
2632        button_layout.addStretch()
2633
```

```
        layout.addLayout(button_layout)
        layout.addStretch()

        self.setLayout(layout)

    def update_stats(self):
        """Update enhanced statistics with AI insights"""
        words = self.db.read_all_words()

        # Basic statistics
        total_words = len(words)
        self.total_words_label.setText(f"Total Dictionary Entries: {total_words}")

        # Word type analysis
        type_counts = {}
        for word in words:
            word_type = word[3]
            type_counts[word_type] = type_counts.get(word_type, 0) + 1

        type_text = "Word Types: " + ", ".join([f"{t}: {c}" for t, c in type_counts.items()])
        self.word_types_label.setText(type_text)

        # User profile statistics
        user_facts = self.user_profile.get_user_facts()
        user_text = (f"Your Profile: Level: {user_facts.get('user_level', 'Unknown').title()}, "
                     f"Searches: {user_facts.get('search_count', 0)}, "
                     f"Session Time: {user_facts.get('session_time', 0)}s")
        self.user_stats_label.setText(user_text)

        self.searches_label.setText(f"Searches This Session: {self.search_count}")

        # Generate AI insights
        self.generate_ai_insights(words, user_facts)

        # Display sample words
        self.display_samples(words)

    def generate_ai_insights(self, words, user_facts):
        """Generate AI-powered learning insights"""
        insights = []
```

```
        # Analyze user progress
        user_level = user_facts.get('user_level', 'beginner')
        search_count = user_facts.get('search_count', 0)

        if search_count < 5:
            insights.append("<strong>Getting Started:</strong> You're beginning your Khmer learning journey! Focus on basic vocabulary first.")
        elif search_count < 20:
            insights.append("<strong>Building Momentum:</strong> Good progress! Try exploring different word types to expand your vocabulary.")
        else:
            insights.append("<strong>Advanced Learner:</strong> Excellent dedication! Consider focusing on cultural context and grammar patterns.")

        # Learning recommendations
        recommendations = []
        if user_level == "beginner":
            recommendations.append("Focus on daily vocabulary (food, family, greetings)")
            recommendations.append("Practice simple sentence structures")
        elif user_level == "intermediate":
            recommendations.append("Explore word relationships and synonyms")
            recommendations.append("Study cultural context and formal speech")
        else:
            recommendations.append("Master complex grammar patterns")
            recommendations.append("Study literature and advanced expressions")

        # Combine insights and recommendations
        html_content = "<h4>Your Learning Insights:</h4><ul>"
        for insight in insights:
            html_content += f"<li>{insight}</li>"
        html_content += "</ul><h4>AI Recommendations:</h4><ul>"
        for rec in recommendations:
            html_content += f"<li>{rec}</li>"
        html_content += "</ul>"

        self.insights_display.setHtml(html_content)

    def display_samples(self, words):
        """Display sample words"""
        import random
        sample_words = random.sample(words, min(8, len(words)))

        html_content = "<h4>Sample Words:</h4>"
        for word in sample_words:
            english, khmer, word_type = word[1], word[2], word[3]
            html_content += f"<p><strong>{english}</strong> → {khmer} ({word_type})</p>"
```

```
2718
2719            self.sample_display.setHtml(html_content)
2720
2721        def export_data(self):
2722            """Export data with comprehensive options"""
2723            try:
2724                # Show export options dialog
2725                export_dialog = ExportDialog(self.font_manager, self)
2726                if export_dialog.exec() != QDialog.DialogCode.Accepted:
2727                    return
2728
2729                # Get export settings
2730                formats, data_types, options = export_dialog.get_export_settings()
2731
2732                if not formats or not data_types:
2733                    msg = self.font_manager.create_message_box(
2734                        self, QMessageBox.Icon.Warning,
2735                        "Export Error",
2736                        "Please select at least one format and one data type to export."
2737                    )
2738                    msg.exec()
2739                    return
2740
2741                # Choose save location
2742                base_filename, _ = QFileDialog.getSaveFileName(
2743                    self,
2744                    "Export Dictionary Data",
2745                    f"khmer_dictionary_export_{datetime.now().strftime('%Y%m%d')}",
2746                    "All Files (*)"
2747                )
2748
2749                if not base_filename:
2750                    return
2751
2752                # Remove extension if provided
2753                base_filename = str(Path(base_filename).with_suffix(''))
2754
2755                # Create exporter and export data
2756                exporter = DataExporter(self.db, self.user_profile, self.font_manager)
2757
2758                # Show progress dialog
2759                progress_msg = QMessageBox(self)
2760                progress_msg.setWindowTitle("Exporting Data")
```

```
2761            progress_msg.setText("Exporting data, please wait...")
2762            progress_msg.setStandardButtons(QMessageBox.StandardButton.NoButton)
2763            progress_msg.show()
2764            QApplication.processEvents()
2765
2766            # Perform export
2767            results = exporter.export_data(formats, data_types, options, base_filename)
2768
2769            progress_msg.close()
2770
2771            # Show results
2772            success_files = [filename for filename, success in results if success]
2773            failed_files = [filename for filename, success in results if not success]
2774
2775            result_text = f"Export completed!\n\n"
2776
2777            if success_files:
2778                result_text += f"✅ Successfully exported {len(success_files)} files:\n"
2779                for filename in success_files:
2780                    result_text += f"• {Path(filename).name}\n"
2781                result_text += "\n"
2782
2783            if failed_files:
2784                result_text += f"❌ Failed to export {len(failed_files)} files:\n"
2785                for filename in failed_files:
2786                    result_text += f"• {filename}\n"
2787                result_text += "\n"
2788
2789            result_text += f"📁 Files saved to: {Path(base_filename).parent}\n\n"
2790
2791            # Add export summary
2792            result_text += "📊 Export Summary:\n"
2793            result_text += f"• Formats: {', '.join(formats).upper()}\n"
2794            result_text += f"• Data Types: {', '.join(data_types).title()}\n"
2795            result_text += f"• Include Timestamps: {options.get('include_timestamps', False)}\n"
2796            result_text += f"• Include Metadata: {options.get('include_metadata', False)}\n"
2797
2798            icon = QMessageBox.Icon.Information if success_files else QMessageBox.Icon.Warning
2799            msg = self.font_manager.create_message_box(
2800                self, icon,
2801                "Export Complete" if success_files else "Export Failed",
2802                result_text
2803            )
```

```
            msg.exec()

        except Exception as e:
            error_msg = self.font_manager.create_message_box(
                self, QMessageBox.Icon.Critical,
                "Export Error",
                f"An error occurred during export:\n\n{str(e)}\n\nPlease try again or check file permissions."
            )
            error_msg.exec()

    def reset_profile(self):
        """Reset user profile"""
        msg = self.font_manager.create_message_box(
            self, QMessageBox.Icon.Question,
            "Reset Profile",
            "Reset your learning profile?\n\nThis will clear search history and preferences.",
            QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
        )

        if msg.exec() == QMessageBox.StandardButton.Yes:
            # Reset profile data
            self.user_profile.profile = {
                "search_history": [],
                "word_preferences": defaultdict(int),
                "language_preference": "english",
                "difficulty_level": "beginner",
                "search_count": 0,
                "session_time": 0,
                "favorite_word_types": defaultdict(int),
                "error_patterns": [],
                "learning_progress": {"beginner": 0, "intermediate": 0, "advanced": 0},
                "last_active": datetime.now().isoformat()
            }
            self.user_profile.save_profile()
            self.search_count = 0
            self.update_stats()

    def increment_search_count(self):
        self.search_count += 1
        self.searches_label.setText(f"Searches This Session: {self.search_count}")

# Main Application with Expert System Integration but Standard UI
class KhmerEnglishDictionaryApp(QMainWindow):
```

```
2847    def __init__(self):
2848        super().__init__()
2849        self.db = DictionaryDatabase()
2850        self.font_manager = FontManager()
2851        self.expert_engine = ExpertSystemEngine()
2852        self.user_profile = UserProfileManager()
2853        self.ai_generator = AIExplanationGenerator()
2854        self.init_ui()
2855        self.connect_signals()
2856
2857    def init_ui(self):
2858        # Apply font to entire application
2859        self.font_manager.apply_font(self)
2860
2861        self.setWindowTitle("Khmer-English Dictionary (Expert System + Voice Search) • វចនានុក្រមអង់គ្លេស-ខ្មែរ")
2862        self.setGeometry(100, 100, 1200, 800)
2863
2864        # Use standard Qt styling with Khmer font
2865        self.setStyleSheet(f"""
2866            QMainWindow {{
2867                background-color: #f5f5f5;
2868                font-family: '{self.font_manager.get_font_family()}';
2869                font-size: {self.font_manager.font_size}pt;
2870            }}
2871            QTabWidget::pane {{
2872                border: 1px solid #c0c0c0;
2873                background-color: white;
2874                font-family: '{self.font_manager.get_font_family()}';
2875                font-size: {self.font_manager.font_size}pt;
2876            }}
2877            QTabBar::tab {{
2878                background-color: #e0e0e0;
2879                padding: 8px 16px;
2880                margin-right: 2px;
2881                font-family: '{self.font_manager.get_font_family()}';
2882                font-size: {self.font_manager.font_size}pt;
2883            }}
2884            QTabBar::tab:selected {{
2885                background-color: white;
2886                border-bottom: 2px solid #4a90e2;
2887            }}
2888            QGroupBox {{
2889                font-weight: bold;
```

```
2890                border: 2px solid #cccccc;
2891                border-radius: 5px;
2892                margin-top: 1ex;
2893                padding-top: 10px;
2894                font-family: '{self.font_manager.get_font_family()}';
2895                font-size: {self.font_manager.font_size}pt;
2896            }}
2897            QGroupBox::title {{
2898                subcontrol-origin: margin;
2899                left: 10px;
2900                padding: 0 5px 0 5px;
2901                font-family: '{self.font_manager.get_font_family()}';
2902                font-size: {self.font_manager.font_size}pt;
2903            }}
2904            QLineEdit, QComboBox, QPushButton, QTextEdit, QTextBrowser {{
2905                font-family: '{self.font_manager.get_font_family()}';
2906                font-size: {self.font_manager.font_size}pt;
2907                padding: 5px;
2908            }}
2909            QTableView {{
2910                font-family: '{self.font_manager.get_font_family()}';
2911                font-size: {self.font_manager.font_size}pt;
2912                gridline-color: #d0d0d0;
2913            }}
2914            QHeaderView::section {{
2915                background-color: #e8e8e8;
2916                padding: 8px;
2917                border: 1px solid #c0c0c0;
2918                font-family: '{self.font_manager.get_font_family()}';
2919                font-size: {self.font_manager.font_size}pt;
2920                font-weight: bold;
2921            }}
2922            QLabel {{
2923                font-family: '{self.font_manager.get_font_family()}';
2924                font-size: {self.font_manager.font_size}pt;
2925            }}
2926            QScrollArea {{
2927                border: 1px solid #cccccc;
2928                border-radius: 3px;
2929            }}
2930        """)
2931
```

```
2932        central_widget = QWidget()
2933        self.font_manager.apply_font(central_widget)
2934        self.setCentralWidget(central_widget)
2935
2936        layout = QVBoxLayout()
2937        central_widget.setLayout(layout)
2938
2939        self.tab_widget = QTabWidget()
2940        self.font_manager.apply_font(self.tab_widget)
2941
2942        # Create enhanced tabs with expert system features but standard UI
2943        self.translator_tab = TranslatorWidget(self.db, self.font_manager,
2944                                               self.expert_engine, self.user_profile,
2945                                               self.ai_generator)
2946        self.manager_tab = DictionaryManagerWidget(self.db, self.font_manager,
2947                                                   self.ai_generator)
2948        self.stats_tab = StatisticsWidget(self.db, self.font_manager,
2949                                          self.user_profile)
2950
2951        self.tab_widget.addTab(self.translator_tab, "Translator")
2952        self.tab_widget.addTab(self.manager_tab, "Dictionary Manager")
2953        self.tab_widget.addTab(self.stats_tab, "Statistics")
2954
2955        layout.addWidget(self.tab_widget)
2956
2957        # Status bar with Khmer font
2958        status_bar = self.statusBar()
2959        self.font_manager.apply_font(status_bar)
2960        status_bar.showMessage("Ready - Expert System + Voice Search Active - ស្វមស្វាគមន៍មកកាន់វចនានុក្រម")
2961
2962        # Focus on the first tab
2963        self.tab_widget.setCurrentIndex(0)
2964
2965    def connect_signals(self):
2966        """Connect signals between enhanced widgets"""
2967        self.translator_tab.word_searched.connect(
2968            lambda word, search_type, found: [
2969                self.stats_tab.increment_search_count(),
2970                self.statusBar().showMessage(f"Searched: {word} ({'Found' if found else 'Not found'})")
2971            ]
2972        )
2973
```

```
2974            self.manager_tab.word_added.connect(
2975                lambda word: [
2976                    self.statusBar().showMessage(f"✓ Created word: {word}"),
2977                    self.stats_tab.update_stats()
2978                ]
2979            )
2980
2981            self.manager_tab.word_updated.connect(
2982                lambda word_id: [
2983                    self.statusBar().showMessage(f"✓ Updated word ID: {word_id}"),
2984                    self.stats_tab.update_stats()
2985                ]
2986            )
2987
2988            self.manager_tab.word_deleted.connect(
2989                lambda word_id: [
2990                    self.statusBar().showMessage(f"✓ Deleted word ID: {word_id}"),
2991                    self.stats_tab.update_stats()
2992                ]
2993            )
2994
2995        def closeEvent(self, event):
2996            """Save user profile when application closes"""
2997            self.user_profile.save_profile()
2998            event.accept()
2999
3000    def main():
3001        app = QApplication(sys.argv)
3002
3003        # Create font manager and set application-wide font
3004        font_manager = FontManager()
3005        app.setFont(font_manager.get_font())
3006
3007        app.setApplicationName("Expert System Khmer-English Dictionary with Voice Search")
3008        app.setApplicationVersion("2.1 - AI Enhanced + Voice")
3009        app.setOrganizationName("AI Learning Tools")
3010
3011        try:
3012            window = KhmerEnglishDictionaryApp()
3013            window.show()
3014            window.showMaximized()
3015
```

```
3016            sys.exit(app.exec())
3017
3018        except Exception as e:
3019            print(f"❌ Error starting expert system application: {e}")
3020            import traceback
3021            traceback.print_exc()
3022
3023            # Show error dialog
3024            try:
3025                error_msg = QMessageBox()
3026                error_msg.setIcon(QMessageBox.Icon.Critical)
3027                error_msg.setWindowTitle("Application Error")
3028                error_msg.setText(f"An error occurred while starting the application:\n\n{str(e)}")
3029                error_msg.setDetailedText(traceback.format_exc())
3030                error_msg.exec()
3031            except:
3032                pass
3033
3034            sys.exit(1)
3035
3036    if __name__ == '__main__':
3037        main()
```