English Speaking Practice - OOP Architecture Documentation

🎯 Overview

This application has been refactored using **Object-Oriented Programming principles** and **Design Patterns** to create a maintainable, scalable, and testable codebase.

📁 Project Structure

```
app/
├── core/                    # Core application logic
│   ├── interfaces.py        # Abstract base classes & interfaces
│   ├── factories.py         # Factory classes for object creation
│   ├── dependencies.py      # Dependency Injection container
│   ├── config.py            # Configuration management
│   └── bootstrap.py         # Application initialization
│
├── business/                # Business logic layer
│   ├── conversation_service.py   # Conversation management
│   ├── dictionary_service.py     # Dictionary lookup service
│   └── strategies/          # Strategy pattern implementations
│       └── prompt_strategies.py  # Prompt building strategies
│
├── data/                    # Data access layer
│   ├── database/
│   │   ├── base.py          # SQLAlchemy base
│   │   └── session.py       # Database session management
│   ├── models/
│   │   ├── conversation.py      # Conversation model
│   │   └── message.py           # Message model
│   ├── repositories/
│   │   ├── base_repository.py   # Generic repository pattern
│   │   └── ...              # Specialized repositories
│   └── external_api/
│       ├── llm_client.py        # LLM API client
│       └── dictionary_api.py    # Dictionary API client
│
├── presentation/            # Presentation layer (API)
│   ├── conversation_controller.py
│   └── dictionary_controller.py
│
```

```
└── tests/                    # Test suite
    └── test_conversation_service.py
```

🏛️ Architecture Principles

1. SOLID Principles

Single Responsibility Principle (SRP)

Each class has one reason to change:

- `ConversationService` - manages conversations
- `MessageRepository` - handles message data access
- `LLMClient` - communicates with LLM API

Open/Closed Principle (OCP)

Open for extension, closed for modification:

- `BaseRepository<T>` can be extended without modification
- Strategy pattern allows adding new prompt strategies

Liskov Substitution Principle (LSP)

Subtypes are substitutable:

- Any `ILLMClient` implementation works (Ollama, Mock, etc.)
- All strategies implement `IPromptStrategy`

Interface Segregation Principle (ISP)

Clients depend only on interfaces they use:

- Separate interfaces: `IRepository`, `ILLMClient`, `ISpeechService`
- No fat interfaces

Dependency Inversion Principle (DIP)

Depend on abstractions, not concretions:

- Services depend on `IRepository`, not `ConversationRepository`

- Easy to swap implementations

2. Layer Separation

```
Presentation Layer (Controllers)
        ↓
Business Logic Layer (Services)
        ↓
Data Access Layer (Repositories)
        ↓
Database / External APIs
```

Benefits:

- Changes in one layer don't affect others
- Easy to test each layer independently
- Clear responsibility boundaries

🎨 Design Patterns Used

1. Repository Pattern

Purpose: Encapsulate data access logic

```python
# Generic repository with type safety
class BaseRepository(Generic[T], IRepository[T]):
    def create(self, db: Session, **kwargs) -> T:
        # Implementation
        pass
```

Benefits:

- Centralized data access
- Easy to mock for testing
- Type-safe operations

2. Strategy Pattern

Purpose: Define family of algorithms, make them interchangeable

```python
# Different prompt strategies for different levels
class BeginnerPromptStrategy(IPromptStrategy):
    def build_system_prompt(self, topic: str, level: str) -> str:
        # Beginner-specific logic
        pass

class AdvancedPromptStrategy(IPromptStrategy):
    def build_system_prompt(self, topic: str, level: str) -> str:
        # Advanced-specific logic
        pass
```

Benefits:

- Easy to add new strategies
- Swap algorithms at runtime
- Follows Open/Closed principle

3. Observer Pattern

Purpose: Define one-to-many dependency for event notification

```python
class ConversationService(ISubject):
    def notify(self, event: str, data: Any) -> None:
        for observer in self._observers:
            observer.update(self, event, data)

# Observers
class LoggingObserver(IObserver):
    def update(self, subject, event, data):
        # Log the event
        pass

class AnalyticsObserver(IObserver):
    def update(self, subject, event, data):
        # Track analytics
        pass
```

Benefits:

- Loose coupling
- Easy to add new observers
- Real-time event handling

4. Factory Pattern

Purpose: Create objects without specifying exact classes

```python
class ServiceFactory:
    def create_conversation_service(self) -> ConversationService:
        # Create with all dependencies
        return ConversationService(
            conversation_repo=ConversationRepository(),
            message_repo=MessageRepository(),
            llm_client=self.create_llm_client()
        )
```

Benefits:

- Centralized object creation
- Easy to change implementations
- Configuration-based creation

5. Dependency Injection (DI)

Purpose: Provide dependencies from outside

```python
class ConversationService:
    def __init__(
        self,
        conversation_repo: IRepository,  # Injected
        message_repo: IRepository,       # Injected
        llm_client: ILLMClient           # Injected
    ):
        self.conv_repo = conversation_repo
        self.msg_repo = message_repo
        self.llm = llm_client
```

Benefits:

- Easy to test with mocks
- Loose coupling
- Flexible configuration

6. Singleton Pattern

Purpose: Ensure single instance

```python
class Configuration(IConfiguration):
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance
```

Benefits:

- Global configuration access
- Resource efficiency
- Consistent state

7. Builder Pattern

Purpose: Construct complex objects step by step

```python
prompt = (ContextAwarePromptBuilder()
    .set_topic("Job Interview")
    .set_level("Advanced")
    .set_style("conversational")
    .add_special_instruction("Focus on technical questions")
    .build())
```

Benefits:

- Fluent API

- Complex object construction
- Optional parameters handling

🔧 Key Components

1. Interfaces (`app/core/interfaces.py`)

Define contracts for all components:

- `IRepository<T>` - Data access interface
- `ILLMClient` - LLM communication interface
- `IConversationService` - Service interface
- `IObserver` / `ISubject` - Observer pattern
- `IPromptStrategy` - Strategy pattern

2. Base Repository (`app/data/repositories/base_repository.py`)

Generic CRUD operations:

```python
# Type-safe repository
repo = BaseRepository[Conversation](Conversation)
conversation = repo.create(db, user_id=1, topic="Travel")
```

3. Service Layer (`app/business/`)

Business logic with dependency injection:

```python
service = ConversationService(
    conversation_repo=ConversationRepository(),
    message_repo=MessageRepository(),
    llm_client=LLMClient()
)
```

4. Dependency Container (`app/core/dependencies.py`)

Manages all dependencies:

```python
```

```
container = get_container()
service = container.get('conversation_service')
```

5. Strategy Factory (`app/business/strategies/prompt_strategies.py`)

Creates appropriate strategies:

```
python

strategy = PromptStrategyFactory.create(level="Beginner")
prompt = strategy.build_system_prompt("Ordering Coffee", "Beginner")
```

🚀 Usage Examples

Starting a Conversation

```
python

# Get service from container
container = get_container()
service = container.get('conversation_service')

# Start conversation
conversation = service.start_conversation(
    db=db,
    user_id=1,
    topic="Job Interview",
    level="Advanced",
    style="conversational",
    user_preferences={
        'focus_areas': ['pronunciation', 'grammar'],
        'pace': 'moderate'
    }
)
```

Adding Custom Observers

```
python
```

```python
class CustomObserver(IObserver):
    def update(self, subject, event, data):
        if event == 'message_sent':
            # Custom logic
            send_to_analytics_service(data)

# Attach observer
service.attach(CustomObserver())
```

Creating Custom Prompt Strategy

```python
class BusinessEnglishStrategy(BasePromptStrategy):
    def _build_response_style(self, level: str) -> str:
        return "Use formal business English. Include industry terminology."

# Register strategy
PromptStrategyFactory.register_strategy('Business', BusinessEnglishStrategy)

# Use it
strategy = PromptStrategyFactory.create(level="Advanced", style="Business")
```

Testing with Mocks

```python
# Create service with mock LLM
mock_llm = MockLLMClient(responses=["Great!", "Tell me more."])
service = ConversationService(
    conversation_repo=ConversationRepository(),
    message_repo=MessageRepository(),
    llm_client=mock_llm  # Use mock instead of real LLM
)

# Test without actual API calls
reply = service.send_message(db, conversation_id, "Hello")
assert reply == "Great!"
```

🧪 Testing Strategy

Unit Tests

Test individual components in isolation:

```python
def test_prompt_strategy():
    strategy = BeginnerPromptStrategy()
    prompt = strategy.build_system_prompt("Travel", "Beginner")
    assert "simple" in prompt.lower()
```

Integration Tests

Test component interactions:

```python
def test_send_message_integration():
    service = create_test_service()
    conversation = service.start_conversation(...)
    reply = service.send_message(...)
    assert reply is not None
```

Mock Objects

Use mocks for external dependencies:

```python
mock_llm = MockLLMClient()
service = ConversationService(..., llm_client=mock_llm)
```

📊 Benefits of This Architecture

1. Maintainability

- Clear separation of concerns
- Easy to locate and fix bugs
- Self-documenting code

2. Scalability

- Add new features without breaking existing code
- Horizontal scaling possible
- Performance optimizations isolated

3. Testability

- Mock external dependencies
- Test each layer independently
- High test coverage possible

4. Flexibility

- Swap implementations easily
- Configure via dependency injection
- Runtime behavior changes

5. Code Reusability

- Generic base classes
- Composable components
- Strategy pattern for variations

🔄 Migration Guide

From Old Code to New

Old:

```python
# Direct instantiation
service = ConversationService()
reply = service.send_message(db, conv_id, text)
```

New:

```python
```

```
# Use dependency injection
container = get_container()
service = container.get('conversation_service')
reply = service.send_message(db, conv_id, text)
```

Adding New Features

Example: Add new LLM provider

1. Implement interface:

```
python

class OpenAIClient(ILLMClient):
    def generate_reply(self, messages):
        # OpenAI implementation
        pass
```

2. Register in factory:

```
python

def create_llm_client(self):
    if self.config.get('llm_type') == 'openai':
        return OpenAIClient()
```

3. Configure:

```
python

# .env
LLM_TYPE=openai
```

🎓 Learning Resources

- **SOLID Principles:** Uncle Bob's SOLID Principles
- **Design Patterns:** "Design Patterns: Elements of Reusable Object-Oriented Software" (Gang of Four)
- **Clean Architecture:** "Clean Architecture" by Robert C. Martin
- **Python OOP:** Real Python OOP

📝 Best Practices

1. **Always use interfaces** - Code against abstractions
2. **Inject dependencies** - Don't create inside constructors
3. **Single Responsibility** - One class, one job
4. **Write tests first** - TDD approach
5. **Keep methods small** - Max 20-30 lines
6. **Use type hints** - Helps catch errors early
7. **Document interfaces** - Clear contracts
8. **Follow naming conventions** - Consistent names

🤝 Contributing

When adding new features:

1. Create interface first
2. Implement concrete class
3. Add to factory/container
4. Write tests
5. Update documentation

📧 Support

For questions about the architecture, refer to:

- Interface definitions in `app/core/interfaces.py`
- Examples in `tests/` directory
- This documentation

Version: 2.0.0
Last Updated: December 2024
Architecture: Clean Architecture + SOLID + Design Patterns