CS3307A Final Project Report – Deliverable 3

Qlink: An Interactive Mental Model Simulator with Link Prediction

Dhir Pathak - 251236762
dpathak8@uwo.ca

Video Demo: https://youtu.be/m3_jfGLxcH0

1.1 System Modules

When I first set out to build Qlink, I knew from the start that organization would make or break this project. With so many moving parts like the link prediction algorithms, LLM integration, NLP, graph visualization, and modification functionality like undo/ redo, I needed to implement a solution that had a clean separation of concerns to ensure everything was maintainable. This resulted in a 3 layer architecture where each module has a specific job and doesn't step on anyone's toes.

Module 1 **UI Layer (ui/):** This is what users see and interact with. Everything here is about presenting information in an appealing way, making sure the user experience is not too complicated.

- **MainWindow**: Primary application window coordinating all UI components.
    - This is responsible for orchestrating everything pertaining to the menu bar, panels, and status updates.
- **GraphWidget**: Interactive graph visualization.
    - This is where the mental models are represented as interactive graphs using Qt's graphics framework. It handles all the visual concepts as nodes, relationships as edges, zoom and pan interactions, as well as the drag and drop positioning.
- **SuggestionPanel**: Link prediction and AI suggestion display.
    - This panel lives on the side showing recommendations for new connections/ relationships between already established concepts.
- **Custom Graphics Items**: Building blocks.
    - Here we have ConceptGraphicsItem and RelationshipGraphicsItem that are basically adapters to translate the domain model (as C++ objects) into Qt graphics, acting as the bridge between what we store and what users see.

Module 2 **Core Domain Layer (core/):** This is where the business logic and application algorithms live. Note that nothing in this layer knows anything about Qt widgets or visual rendering, and that's intentional so we can test this logic in isolation.

- Model Package (core/model/): The data structures representing 'knowledge'.
    - **MentalModel**: Central data structure managing concepts and relationships.
        - This holds all the concepts, relationships of the graph and also emits signals whenever something changes, so the UI can stay in sync without constantly polling.
    - **Concept**: Node entities representing knowledge concepts.
        - Each concept has a unique ID, a name, a description, optional tags, and a position on the graph.
    - **Relationship**: Edge entities connecting different concepts/ nodes.
        - Captures how concepts relate. They can be directed (A influences B) or undirected (A and B are related), and they carry types and weights.

- AI Package (core/ai/): Graph theory algorithms to predict connections & LLM integration
    a. **AIAssistant**: Natural language processing and concept explanations.
        i. This handles natural language tasks like generating concept descriptions or explaining why two ideas might be connected. Uses the Cohere API for this (note that you'll have to add your own token in a .env file to use this feature since I don't want to make my token public).
    b. **ILinkPredictor**: Strategy interface for link prediction algorithms.
        i. We have multiple algorithms for suggesting links, and this interface lets us swap them out at runtime. Different algorithms have different strengths, so users can pick what works best for their mental model.
            1. The graph algorithms (CommonNeighborPredictor, JaccardCoefficientPredictor, and PreferentialAttachmentPredictor) were made using the IGraph library. They calculate the similarity and attachment scores uniquely based on the calculations needed for each predictor.
- NLP Package (core/nlp/): Natural language processing and the command pattern implementation. *This is what lets users type "add concept Gravity", for example, instead of clicking through menus.*
    a. **ICommand**: The command pattern interface.
        i. Every action in Qlink like adding a concept, creating a relationship, deleting something is encapsulated as a command object, and this is what makes the undo/ redo functionality work.
    b. **Commands**: The concrete implementations.
        i. AddConceptCommand knows how to add a concept and how to undo that action. RemoveConceptCommand remembers not just the concept being deleted, but all its relationships too, so undo can restore everything.
    c. **CommandFactory**: The parser.
        i. It takes raw user text like "connect Energy to Mass" and figures out which command to create. Also handles validation for syntax.
    d. **CommandManager**: The controller for execution and history management.
        i. It is responsible for executing the commands, maintaining the undo/ redo history, and makes sure everything happens in the right order.
    e. **NLPProcessor**: Natural language command parsing.
        i. This is the tokenizer and pattern matcher. This breaks down natural language into tokens and matches patterns to extract intent. This is essentially the first step in understanding what users want to do.

- Persistence Package (core/persistence/): The saving and loading of mental models.
    a. **ModelManager**: JSON serialization/deserialization of mental models.
        i. Handles all the JSON serialization and deserialization. It saves the mental model to a file and can load it back (preserving every concept, relationship, position, and tag).
- Common Package (core/common/): The Shared utilities and data structures used throughout.
    a. **DataStructures**: Shared types (Position, LinkSuggestion, ChangeType).
        i. This is where we have the cmmon types needed, like the position for 2D coordinates, LinkSuggestion for AI recommendations, ChangeType for event notifications. Having these in one place prevents duplication and keeps types consistent.
    b. **QLinkException**: The custom exception hierarchy (pretty self-explanatory, when something goes wrong, it throws meaningful exceptions with helpful messages)

Module 3: Testing Layer (tests/)

- Here we have 158 unit tests covering everything from basic functionality to obscure edge cases.
    a. Almost all classes in the core layer has corresponding tests.
    b. Uses GoogleTest fixtures to set up common scenarios once and reuse them across multiple tests, keeping things DRY.
    c. High test coverage for edge cases and failure scenarios:
        i. Empty models, null pointers, duplicate IDs, deleting concepts with relationships, undoing after redo, redo after new command, and many more.

1.2 Data Flow Architecture

The user interaction flow is as follows:

Path 1: Natural Language Command Processing

1. User enters text in NLP panel
2. MainWindow::executeNaturalLanguageCommand() triggered
3. CommandFactory::createCommand() parses input
4. ICommand object created (AddConceptCommand, etc.)
5. MainWindow::executeCommand() adds to undo/redo history
6. Command::execute() modifies MentalModel
7. MentalModel emits Qt signal (conceptAdded, etc.)
8. GraphWidget receives signal and updates visualization
9. Command logged to history panel with success/failure status

Path 2: Direct UI Manipulation

1. User right-clicks graph → contextMenuEvent
2. Context menu displays actions (Add Concept, Delete, AI Explain)
3. Action selected creates Command object
4. Goes through same execute() → model update → signal → UI refresh

Path 3: AI-Powered Link Suggestion

1. User clicks "Generate Suggestions" in SuggestionPanel
2. Selected algorithm retrieved (Common Neighbor, Jaccard, etc.)
3. ILinkPredictor::predictLinks() analyzes graph structure
4. LinkSuggestion objects created with confidence scores
5. Displayed in tree view with filtering/sorting
6. User accepts → CreateRelationshipCommand executed
7. Model updated, graph refreshed

Path 4: Undo/Redo Operation

1. User presses Ctrl+Z or Ctrl+Shift+Z
2. MainWindow::undo() or redo() slot called
3. undoRedoHistoryIndex adjusted
4. Command::undo() or execute() invoked
5. Model state reverted or re-applied
6. UI automatically updates via signals
7. Action tooltips show what will be undone/redone

1.3 Key Architectural Decisions:

Decision 1: Using Qt Signals/Slots for Event Driven Updates

Rationale: In traditional UI programming, we often see code like this:

```
model->addConcept(concept);
graphWidget->refresh();          // Manual refresh
suggestionPanel->updateStats();  // More manual updates
statusBar->showMessage("Added"); // Even more coupling
```

This creates tight coupling as the model knows about all its observers and must call update methods on each one. Adding a new UI component means modifying the model code, violating the Open/Closed Principle. I chose Qt's signals/slots mechanism because it implements the Observer pattern at the framework level with a ton of advantages. For instance,

1. Type Safety: Unlike string-based events or dynamic dispatch, Qt's moc generates code at compile time, catching type mismatches before runtime.
2. Thread Safety: Signal emissions are thread safe by default. If a signal is emitted from a different thread than the receiver, Qt automatically queues the call and executes it in the receiver's thread.
3. Automatic Connection Management: When a QObject is destroyed, all its signal/slot connections are automatically cleaned up. No dangling pointers.
4. Many-to-Many: One signal can connect to multiple slots, and one slot can receive multiple signals. This flexibility costs nothing in code complexity.

Therefore, after going with the Qt signals/ slots the implementation looks as follows.

```
class MentalModel : public QObject {
    Q_OBJECT  // Required macro for moc compiler
signals:
    void conceptAdded(const QString& conceptId);
    void conceptRemoved(const QString& conceptId);
    void relationshipAdded(const QString& relationshipId);
    void relationshipRemoved(const QString& relationshipId);
    void modelChanged();  // Coarse-grained for full rebuilds
};

GraphWidget.cpp:
  void GraphWidget::setModel(MentalModel* m) {
      // Disconnect old model to prevent memory leaks
      if (model && model != m) {
          disconnect(model, nullptr, this, nullptr);
      }
      model = m;
      // Connect using Qt's new syntax (compile-time checking)
      connect(model, &MentalModel::conceptAdded,
              this, &GraphWidget::onConceptAdded);
      connect(model, &MentalModel::conceptRemoved,
              this, &GraphWidget::onConceptRemoved);
      // ... more connections
  }
```

This has numerous benefits since GraphWidget now has zero knowledge of MentalModel's implementation, meaning, adding a SuggestionPanel required no changes to MentalModel code. Here, unit tests can create mock models that emit signals without Qt dependency and any future additions like the undo/redo implementation can emit signals to update UI automatically.

Decision 2: Command Pattern for All Model Modifications

Rationale: Adding in functionality to undo/redo functionality is a hard requirement to implement and to add it begs to ask the following questions:

- How do you reverse a delete operation when the deleted object is gone?
- What happens if you undo, then make a new change (branching history)?
- How do you undo a cascade delete (concept + all its relationships)?

The Command Pattern solves all of these by encapsulating each operation as as object that knows how to:

1. Execute itself (do the operation)
2. Undo itself (reverse the operation)
3. Describe itself (for history UI)

An alternative to do this could have also been to do this via a transaction Log, where we store JSON diffs of model changes. But this was rejected due to complex serialization logic and difficulty handling relationship cascades.

Why I used a shared_ptr for Commands:

- Multiple ownership: Command lives in undo history AND potentially redo stack
- History pruning: When new command executes after undo, old redo branch must be deleted. shared_ptr handles reference counting automatically.
- Exception safety: If execute() throws, shared_ptr ensures cleanup

RemoveConceptCommand was the most complex command because deleting a concept requires:

1. Removing the concept itself
2. Removing ALL relationships connected to it (cascade delete)
3. Storing both concept AND relationships for undo

This looks like:

```
class RemoveConceptCommand : public ICommand {
    MentalModel* model;
    std::string conceptId;
    std::unique_ptr<Concept> removedConcept;  // Stored for undo
    std::vector<std::unique_ptr<Relationship>> removedRelationships;

public:
    void execute() override {
        // CRITICAL: Must store state BEFORE deletion
        const Concept* concept = model->getConcept(conceptId);
        if (!concept) return;  // Already deleted, idempotent

        // Deep copy the concept (ID, name, description, tags, position)
        removedConcept = std::make_unique<Concept>(*concept);

        // Find and store ALL relationships touching this concept
        for (const auto& rel : model->getRelationships()) {
            if (rel->getSourceConceptId() == conceptId ||
                rel->getTargetConceptId() == conceptId) {
```

```
                removedRelationships.push_back(
                    std::make_unique<Relationship>(*rel)
                );
            }
        }

        // Now safe to delete from model
        model->removeConcept(conceptId);
    }

    void undo() override {
        // Restore in correct order: concept first, then relationships
        if (removedConcept) {
            // Move the semantics transfer ownership to model
            model->addConcept(
                std::make_unique<Concept>(*removedConcept)
            );

            // Restore all relationships
            for (auto& rel : removedRelationships) {
                model->addRelationship(
                    std::make_unique<Relationship>(*rel)
                );
            }

            // Clear stored state (prevents double-restore)
            removedConcept.reset();
            removedRelationships.clear();
        }
    }
};
```

This means the history management is as follows:

```
MainWindow maintains undo stack as std::vector<shared_ptr<ICommand>>:

  void MainWindow::executeCommand(shared_ptr<ICommand> cmd) {
      // Branch pruning: If we're in the middle of history (after undo),
      // executing new command creates a branch. We discard the "future".
      if (undoRedoHistoryIndex < undoRedoHistory.size() - 1) {
          undoRedoHistory.erase(
              undoRedoHistory.begin() + undoRedoHistoryIndex + 1,
              undoRedoHistory.end()
          );
      }

      cmd->execute();  // May throw exception
      undoRedoHistory.push_back(cmd);
      undoRedoHistoryIndex++;
      updateUndoRedoActions();  // Update menu tooltips
  }

  void MainWindow::undo() {
      if (undoRedoHistoryIndex >= 0) {
          undoRedoHistory[undoRedoHistoryIndex]->undo();
          undoRedoHistoryIndex--;
          updateUndoRedoActions();
      }
  }
```

Decision 3: Strategy Pattern for AI Algorithms

Rationale: Since link prediction is a graph theory problem with many possible algorithms, each with different characteristics, we need a way to choose the most optimal/ combined one.

- Common Neighbors: Two nodes who share a common friend are more likely to become friends themselves
- Jaccard Coefficient: Normalizes by degree, better for hubs
- Preferential Attachment: Favors high-degree nodes

Since we don't know which algorithm works best for a user's specific mental mode, the solution is to let users choose at runtime and compare results.

The below alternatives were also considered:

- Hardcode one algorithm: Rejected since users have different needs
- Template Method Pattern: Rejected since algorithms have completely different implementations, meaning there's minimal shared code

Utilizing the strategy interface in the ILinkPredictor looks like:

```
class ILinkPredictor {
public:
    virtual ~ILinkPredictor() = default;

    virtual std::vector<LinkSuggestion> predictLinks(
        const MentalModel& model,
        int maxSuggestions = 10
    ) = 0;

    virtual std::string getAlgorithmName() const = 0;
    virtual std::string getDescription() const = 0;

    virtual void setParameter(const std::string& key,double value) {} };
```

Concrete Strategy: Common Neighbors

Algorithm: For each pair of unconnected concepts (u, v), count how many neighbors they share. More shared neighbors = higher confidence.

```
class CommonNeighborPredictor : public ILinkPredictor {
public:
    std::vector<LinkSuggestion> predictLinks(
        const MentalModel& model,
        int maxSuggestions
    ) override {
        // Build adjacency map for O(1) neighbor lookup
        std::unordered_map<std::string, std::unordered_set<std::string>>
            neighbors;

        for (const auto& rel : model.getRelationships()) {
            neighbors[rel->getSourceConceptId()].insert(
                rel->getTargetConceptId()
            );
            // If undirected, add reverse edge
            if (!rel->isDirected()) {
                neighbors[rel->getTargetConceptId()].insert(
                    rel->getSourceConceptId()
                );
            }
        }
```

```
        std::vector<LinkSuggestion> suggestions;
        const auto& concepts = model.getConcepts();

        // Check all pairs
        for (size_t i = 0; i < concepts.size(); i++) {
            for (size_t j = i + 1; j < concepts.size(); j++) {
                const std::string& u = concepts[i]->getId();
                const std::string& v = concepts[j]->getId();

                // Skip if already connected
                if (neighbors[u].count(v) > 0) continue;

                // Count common neighbors (set intersection size)
                int commonCount = 0;
                for (const auto& neighbor : neighbors[u]) {
                    if (neighbors[v].count(neighbor) > 0) {
                        commonCount++;
                    }
                }

                if (commonCount > 0) {
                    // Confidence = commonCount / maxPossible
                    double confidence = static_cast<double>(commonCount) /
                                        std::max(neighbors[u].size(),
                                                 neighbors[v].size());

                    suggestions.push_back(LinkSuggestion{
                        u, v,
                        concepts[i]->getName(),
                        concepts[j]->getName(),
                        confidence,
                        "Common Neighbors",
                        "Share " + std::to_string(commonCount) +
                        " neighbors"
                    });
                }
            }
        }

        // Sort by confidence descending
        std::sort(suggestions.begin(), suggestions.end(),
                  [](const auto& a, const auto& b) {
                      return a.confidence > b.confidence;
                  });

        // Limit results
        if (suggestions.size() > maxSuggestions) {
            suggestions.resize(maxSuggestions);
        }

        return suggestions;
    }

    std::string getAlgorithmName() const override {
        return "Common Neighbors";
    }

    std::string getDescription() const override {
        return "Suggests links between concepts that share many " "neighbors (triadic closure principle)";
    }
};
```

Then in SuggestionPanel, we have the runtime algorithm selection via factory function:

```
void SuggestionPanel::generateSuggestions() {
    QString algorithm = algorithmCombo->currentText();

    // Factory creates appropriate strategy
    std::unique_ptr<ILinkPredictor> predictor;
```

```
    if (algorithm == "Common Neighbors") {
        predictor = std::make_unique<CommonNeighborPredictor>();
    } else if (algorithm == "Jaccard Coefficient") {
        predictor = std::make_unique<JaccardCoefficientPredictor>();
    } else if (algorithm == "Preferential Attachment") {
        predictor = std::make_unique<PreferentialAttachmentPredictor>();
    }

    // Use strategy polymorphically
    auto suggestions = predictor->predictLinks(*model, 20);

    // Display in tree view with confidence scores
    displaySuggestions(suggestions);
}
```

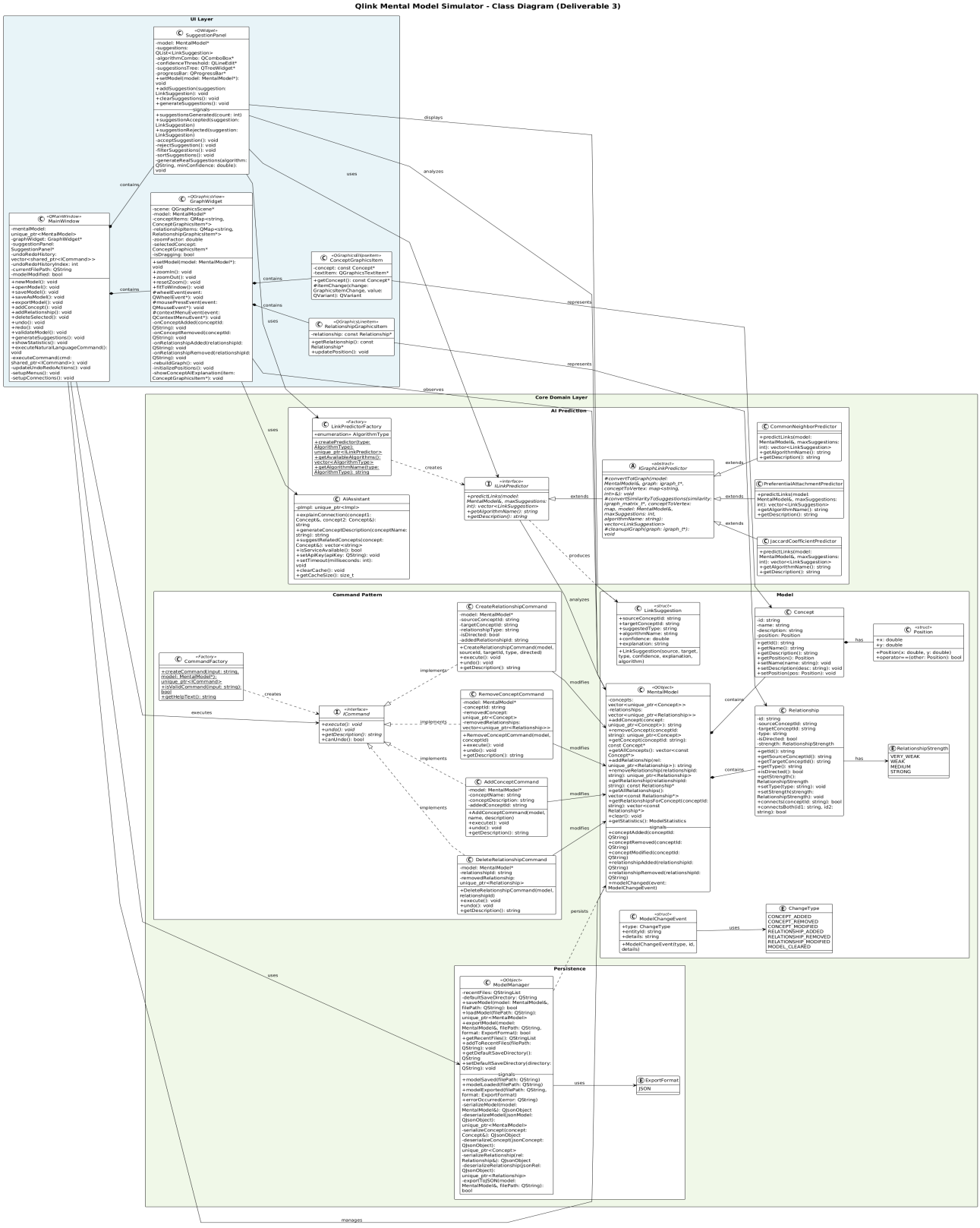
This is extensible because adding in a new algorithm requires:

1. Create new class implementing ILinkPredictor (1 file)
2. Add option to SuggestionPanel combo box (1 line)
3. Add case to factory function (3 lines)

This means that no changes are needed in the:

- ILinkPredictor interface
- MentalModel data structure
- Existing algorithms
- UI display logic

1.4 Class Diagram:



2. Design Patterns Used:

2.1 Command Pattern

Pattern: Command (Behavioral)

Location: core/nlp/ICommand.h, Commands.h/cpp

Rationale:

- Enables full undo/redo functionality for all user operations
- Encapsulates operations as objects for queuing and logging
- Provides consistent interface for natural language and UI commands
- Allows command history tracking and macro recording

```
Interface (ICommand):
  class ICommand {
  public:
      virtual void execute() = 0;
      virtual void undo() = 0;
      virtual std::string getDescription() const = 0;
      virtual bool canUndo() const { return true; }
  };
```

Concrete Commands:

- AddConceptCommand: Stores concept details, removes on undo
- RemoveConceptCommand: Stores removed concept/relationships, restores on undo
- CreateRelationshipCommand: Creates edge, removes on undo
- DeleteRelationshipCommand: Stores deleted edge, restores on undo

Command Execution Flow:

```
  void MainWindow::executeCommand(shared_ptr<ICommand> cmd) {
      // Clear redo stack when new command executed
      if (undoRedoHistoryIndex < undoRedoHistory.size() - 1) {
          undoRedoHistory.erase(
              undoRedoHistory.begin() + undoRedoHistoryIndex + 1,
              undoRedoHistory.end()
          );
      }

      cmd->execute();
      undoRedoHistory.push_back(cmd);
      undoRedoHistoryIndex++;
      updateUndoRedoActions();
  }
```

Improvements this makes to the system:

- Users can experiment freely knowing they can undo mistakes
- Natural language commands have same capabilities as UI commands
- Command history provides audit trail of all operations
- Future: Could save command sequences as reusable macros

2.2 Strategy Pattern

Pattern: Strategy (Behavioral)

Location: core/ai/ILinkPredictor.h, CommonNeighborPredictor.h

Rationale:

- Multiple link prediction algorithms need to be interchangeable
- Algorithm selection happens at runtime based on user preference
- New algorithms can be added without modifying existing code
- Allows A/B testing and comparison of algorithm effectiveness

```
class ILinkPredictor {
public:
    virtual vector<LinkSuggestion> predictLinks(
        const MentalModel& model,
        int maxSuggestions
    ) = 0;
    virtual string getAlgorithmName() const = 0;
    virtual string getDescription() const = 0;
};
```

Concrete Strategies:

- CommonNeighborPredictor: Counts shared neighbors
- JaccardCoefficientPredictor: Normalizes by union of neighbors
- PreferentialAttachmentPredictor: Product of node degrees

```
Context (SuggestionPanel):
  void SuggestionPanel::generateSuggestions() {
      QString algorithm = algorithmCombo->currentText();
      unique_ptr<ILinkPredictor> predictor;

      if (algorithm == "Common Neighbors") {
          predictor = make_unique<CommonNeighborPredictor>();
      } else if (algorithm == "Jaccard Coefficient") {
          predictor = make_unique<JaccardCoefficientPredictor>();
      }

      auto suggestions = predictor->predictLinks(*model, maxCount);
      // Display suggestions...
  }
```

Improvements this makes to the system:

- Users can compare different algorithms on same graph
- Adding new algorithms requires only new subclass
- Algorithms can have different performance characteristics
- Confidence scores allow ranking of suggestions

2.3 Observer Pattern

Pattern: Observer (Behavioral)

Location: core/model/MentalModel.h (Qt signals)

Rationale:

- UI components need automatic updates when model changes
- Multiple observers (GraphWidget, MainWindow, SuggestionPanel) for same model
- Decouples model from UI, model doesn't know about observers
- Qt's signals/slots mechanism provides thread-safe implementationO

```
Subject (MentalModel):
  class MentalModel : public QObject {
      Q_OBJECT
  signals:
      void conceptAdded(const QString& conceptId);
      void conceptRemoved(const QString& conceptId);
      void relationshipAdded(const QString& relationshipId);
      void relationshipRemoved(const QString& relationshipId);
      void modelChanged(const ModelChangeEvent& event);
  };
```

Observers:

```
  // GraphWidget observes model for visualization updates
  void GraphWidget::setModel(MentalModel* m) {
      connect(m, &MentalModel::conceptAdded,
              this, &GraphWidget::onConceptAdded);
      connect(m, &MentalModel::conceptRemoved,
              this, &GraphWidget::onConceptRemoved);
  }

  void GraphWidget::onConceptAdded(const QString& id) {
      const Concept* concept = model->getConcept(id.toStdString());
      createConceptItem(concept);
  }
```

Improvements this makes to the system:

- Automatic UI synchronization without manual refresh calls
- New UI components can observe model without modifying it
- Changes propagate to all observers simultaneously
- Eliminates polling and manual state checking

2.4 Factory Pattern

Pattern: Factory Method (Creational)

Location: core/nlp/CommandFactory.h

Rationale:

- Command creation logic centralized for maintainability
- Natural language parsing requires complex object construction
- Validation happens before object creation
- Extensible for new command types

```
Factory:
  class CommandFactory {
  public:
      static unique_ptr<ICommand> createCommand(
          const string& input,
          MentalModel* model
      );
      static bool isValidCommand(const string& input);
      static string getHelpText();
  };
```

Creation Logic:

```
  unique_ptr<ICommand> CommandFactory::createCommand(
      const string& input,
      MentalModel* model
  ) {
      // Parse input (e.g., "add concept Energy")
      if (starts_with(input, "add concept")) {
          string name = extract_name(input);
          string desc = extract_description(input);
          return make_unique<AddConceptCommand>(model, name, desc);
      }
      // Other command types...
  }
```

Improvements this makes to the system:

- Single point of validation for command syntax
- Easy to add new command types
- Consistent error handling for invalid commands
- Help text generation from same location

2.5 Model View Controller

Pattern: MVC (Architectural)

Location: Entire codebase structure

Rationale:

- Clear separation between data, presentation, and logic
- Multiple views of same data (graph, suggestions, command history)
- Testable business logic independent of UI
- Standard pattern for GUI applications

Implementation Details:

- Model: MentalModel, Concept, Relationship (core/model/)
    - Pure data structures with business logic
    - No Qt UI dependencies
    - Emits change notifications via signals
- View: GraphWidget, SuggestionPanel, MainWindow (ui/)
    - Display model data
    - Capture user input
    - Update on model change signals
- Controller: CommandManager, MainWindow slots (core/nlp/, ui/)
    - Translate user actions to model operations
    - Validate input
    - Coordinate between view and model

- Improvements this makes to the system:
- Core logic testable without UI
- Can add new views (e.g., 3D visualization) easily
- Model can be used in different contexts (CLI, web)
- Clean dependency structure prevents spaghetti code

2.6 Adapter Pattern

Pattern: Adapter (Structural)

Location: ui/GraphWidget.cpp (Qt Graphics items)

Rationale:

- Domain model (Concept, Relationship) incompatible with Qt graphics API
- Qt requires QGraphicsItem subclasses for rendering
- Domain model should remain UI-agnostic
- Multiple graphical representations of same data possible

Adaptee (Domain Model):

```
class Concept {
    string id, name, description;
    Position position;
    vector<string> tags;
};
```

Adapter:

```
class ConceptGraphicsItem : public QGraphicsEllipseItem {
    const Concept* concept;  // Reference to domain object
    QGraphicsTextItem* textItem;
public:
    ConceptGraphicsItem(const Concept* c) : concept(c) {
        // Adapt Concept data to Qt graphics API
        setText(QString::fromStdString(c->getName()));
        setPos(c->getPosition().x, c->getPosition().y);
    }
};
```

Usage:

```
void GraphWidget::createConceptItem(const Concept* concept) {
    auto* item = new ConceptGraphicsItem(concept);
    scene->addItem(item);
    conceptItems[concept->getId()] = item;
}
```

Improvements this makes to the system:

- Domain model remains pure C++, no Qt dependencies
- Can create different visual representations (circle, icon, image)
- Graphics items can be tested separately from model
- Model can be serialized/tested without UI framework

3. Testing Report:

3.1 Testing framework setup

- Framework: Google Test (gtest) version 1.14.0
- Build Integration: CMake with enable_testing()
- Test Organization: Mirrors source structure (tests/model/, tests/ai/, tests/nlp/)
- Execution: Run via 'make test' or './build/tests/QlinkTests'

Test Files:

- tests/test_main.cpp: GoogleTest initialization
- tests/model/test_Concept.cpp: Concept class unit tests
- tests/model/test_MentalModel.cpp: MentalModel class unit tests
- tests/model/test_Relationship.cpp: Relationship class unit tests
- tests/ai/test_CommonNeighborPredictor.cpp: AI algorithm tests
- tests/nlp/test_Commands.cpp: Command pattern tests
- tests/nlp/test_CommandFactory.cpp: Factory pattern tests

3.2 Test Design and Strategy

Each test suite uses a fixture for common setup, as seen below

```
class MentalModelTest : public ::testing::Test {
protected:
    void SetUp() override {
        // Fresh model for each test (test isolation)
        model = std::make_unique<MentalModel>("Test Model");
    }

    void TearDown() override {
        // Automatic cleanup via unique_ptr
        model.reset();
    }

    // Helper methods for common operations
    std::string addTestConcept(const std::string& name) {
        auto concept = std::make_unique<Concept>(name);
        std::string id = concept->getId();
        model->addConcept(std::move(concept));
        return id;
    }

    std::unique_ptr<MentalModel> model;
};
```

I’m using fixtures here to have test isolation, so each test starts with a clean state. It also follows the DRY principle where the setup code was only needed to be written once and then can be reused across 20+ tests. Finally, it also helps constructing the helper Methods as common operations (like addTestConcept) help reduce test verbosity.

In the tests, the assertion style I’m using is EXPECT_* over ASSERT_* typically for better error reporting. As seen below:

```
// ASSERT: Stops test on failure
```

```
ASSERT_NE(model, nullptr);  // If fails, rest of test skipped
EXPECT_EQ(model->getConcepts().size(), 1);  // Never runs

// EXPECT: Continues test on failure
EXPECT_NE(model, nullptr);  // If fails, reports error
EXPECT_EQ(model->getConcepts().size(), 1);  // Still runs!
```

I'm only using ASSERT_* when continuing would cause a segfault/ null pointe or when the rest of the test is meaningless like when the precondition failed. Otherwise, I'm using EXPECT_* so I can see the failures in one test run and get the most info about what's been broken.

3.3 Edge Case Handling

Edge Case 1: Empty Collections

Problem: Operations on empty models/vectors could crash

Solution: Early returns, size checks before access

Tests:

- EmptyModelReturnsNoSuggestions
- ClearRemovesAllConceptsAndRelationships
- GetNonexistentConceptReturnsNull

Edge Case 2: Dangling References After Deletion

Problem: Deleting concept could leave invalid relationship pointers

Solution: RemoveConcept automatically removes connected relationships

Test: RemoveConceptAlsoRemovesRelationships

Implementation:

```
void MentalModel::removeConcept(const string& conceptId) {
    // First remove all relationships connected to this concept
    auto it = relationships.begin();
    while (it != relationships.end()) {
        if ((*it)->connects(conceptId)) {
            it = relationships.erase(it);
        } else {
            ++it;
        }
    }

    // Then remove the concept
    concepts.erase(
        remove_if(concepts.begin(), concepts.end(),
            [&](const auto& c) { return c->getId() == conceptId; }),
        concepts.end()
    );
}
```

Edge Case 3: Multiple Undo/Redo Cycles

Problem: Command state could become inconsistent after multiple undo/redo

Solution: Each command stores complete state for restoration

Test: AddConceptMultipleExecuteUndo

Verification:

```
cmd.execute();   // State A -> B
cmd.undo();      // State B -> A
cmd.execute();   // State A -> B (should be identical to first execute)
```

Edge Case 4: Undo After New Command (Branch Pruning)

Problem: Undo chain has fork point when new command executed after undo

Solution: Clear redo stack when new command executed

Implementation:

```
void MainWindow::executeCommand(shared_ptr<ICommand> cmd) {
    // Prune redo branch
    if (undoRedoHistoryIndex < undoRedoHistory.size() - 1) {
        undoRedoHistory.erase(
            undoRedoHistory.begin() + undoRedoHistoryIndex + 1,
            undoRedoHistory.end()
        );
    }

    cmd->execute();
    undoRedoHistory.push_back(cmd);
    undoRedoHistoryIndex++;
}
```

Edge Case 5: Duplicate Concept IDs

Problem: Multiple concepts could theoretically have same ID

Solution: Using UUID ID generation to ensure uniqueness

Test: IdIsUniqueForEachConcept

Implementation:

```
Concept::Concept(const string& name, const string& description) {
    // Generate unique ID using timestamp + random number
    id = generateUniqueId();
}
```

Edge Case 6: Invalid Command Syntax

Problem: Natural language commands could have malformed syntax

Solution: CommandFactory validates before creating command

Implementation:

```
unique_ptr<ICommand> CommandFactory::createCommand(...) {
    if (!isValidCommand(input)) {
        throw QLinkException("Invalid command syntax");
    }
    // Parse and create command...
}
```

Edge Case 7: Null Model Pointers

Problem: Commands/widgets could receive null model pointer

Solution: Null checks in setters, throw exception if invalid

Implementation:

```
void GraphWidget::setModel(MentalModel* m) {
    if (!m) {
        throw invalid_argument("Model cannot be null");
    }
    // Disconnect old model signals...
    model = m;
    // Connect new model signals...
}
```

4. Challenges, Trade-offs, and Lessons Learned

Challenge 1: Undo/Redo State Management for Complex Operations

Here, the RemoveConceptCommand needed to store not just the removed concept, but also all relationships connected to it. This required careful tracking of dependent entities.

Solution:

- Store removed concept in unique_ptr for ownership
- Store vector of removed relationships separately
- On undo, restore concept first, then relationships
- Used move semantics to transfer ownership cleanly

This looks like:

```
void RemoveConceptCommand::execute() {
    // Store concept and its relationships for undo
    const Concept* concept = model->getConcept(conceptId);
    removedConcept = make_unique<Concept>(*concept);

    // Store all connected relationships
    for (const auto& rel : model->getRelationships()) {
        if (rel->connects(conceptId)) {
            removedRelationships.push_back(
                make_unique<Relationship>(*rel)
            );
        }
    }

    model->removeConcept(conceptId);
}
```

Lessons Learned: Complex undo operations require careful ownership transfer using move semantics and smart pointers to prevent memory leaks.

Challenge 2: Qt Context Menu Delete Causing Crashes

Here, right clicking on delete concepts caused segmentation faults. The issue was subtle: the lambda capture stored a reference to concept->getId(), which became invalid when the concept was deleted.

Debugging Process:

1. Added debug output to see which delete path was being called
2. Discovered menu bar delete worked but context menu delete crashed
3. Examined lambda captures - found const reference issue
4. Testing showed reference became invalid mid-deletion

The solution was to store the concept ID in a local variable before deletion:

```
// This was wrong (crashes):
QAction* deleteAction = menu.addAction("Delete Concept");
connect(deleteAction, &QAction::triggered, [this, concept]() {
    model->removeConcept(concept->getId());  // getId() ref invalid!
});

// While this was right (works):
const std::string conceptId = concept->getId();  // Copy first
const std::string conceptName = concept->getName();
QAction* deleteAction = menu.addAction("Delete Concept");
connect(deleteAction, &QAction::triggered, [this, conceptId, conceptName]() {
    model->removeConcept(conceptId);  // Safe copy
});
```

The crash occurred because of C++ lambda capture semantics and Qt's deferred execution model. Here's the detailed sequence:

1. Context menu created: concept pointer is valid
2. Lambda captures [this, concept]: Stores pointer to ConceptGraphicsItem
3. User clicks menu action
4. Qt queues the lambda for execution (may be several ms later)
5. removeConcept() called:
    a. MentalModel::removeConcept() removes Concept from vector
    b. unique_ptr destructor runs, deleting Concept object
    c. Concept::~Concept() runs
    d. MentalModel emits conceptRemoved signal
    e. GraphWidget::onConceptRemoved() slot runs
    f. ConceptGraphicsItem removed from scene and deleted
6. Lambda tries to execute: concept pointer is now dangling
7. Segfault when accessing concept->getId()

The fix works because std::string makes a deep copy of the ID before the concept object is deleted. The lambda captures the copied string by value, so it owns its own copy that remains valid even after Concept is destroyed.

Why didn't this crash in other code paths?

- Menu bar delete: conceptId captured by value earlier in call stack
- Direct model->removeConcept(id): id already a string, no pointer involved
- Only context menu had pointer capture living across deletion boundary

Lessons learned: Lambda captures with references to objects that will be destroyed during the lambda's execution are dangerous. Always capture by value or make a copy first.

Challenge 3: Styling macOS Native Menu Bar

Here, I found that QSS stylesheets worked for all UI elements except the menu bar, which remained white with white text (invisible). And upon testing I found this was only macOS-specific behavior because macOS uses native menu bar rendering that bypasses Qt's styling system.

To resolve this I had to

1. Disable native menu bar: menuBar()->setNativeMenuBar(false)
2. Set explicit QPalette colors for menu bar text
3. Use QSS for menu styling

This looks like:

```
// Disable macOS native menu bar
menuBar()->setNativeMenuBar(false);

// Set palette for menu text
QPalette palette = menuBar()->palette();
palette.setColor(QPalette::WindowText, QColor("#000000"));
menuBar()->setPalette(palette);
```

Lessons Learned: When developing any application, make sure it is compatible for cross platform. Here, I found that Qt applications need platform-specific workarounds. Always test UI styling on target platforms (macOS, Windows, Linux)

Trade Offs:

- Use Unique IDs vs. Human-Readable IDs
    - Here, I decided to use UUID style unique IDs instead of the user provided concept names since this allows duplicate conflict names and prefects conflicts. Also this allows for all internal references to stay valid.
- Smart Pointers vs. Raw Pointers
    - Ended up using unique_ptr for ownership, shared_ptr for undo history, and raw pointers for non owning references (Qt requires this). This is because it has more exception safety with automatic memory management. I had justified this because the memory safety and clarity are worth the minor verbosity, and could still use raw pointers only for Qt's parent-child system since it handles its own memory management.
- Qt Signals/Slots vs. Direct Function Calls
    - Here we're using signals/slogs for the model to UI communication. This helps decouple the components and allows us to have multiple observer, not to mention, it can help us queue events. Justified this because the decoupling and extensibility is worth the performance cost.
- Complete Undo State vs. Reverse Operations
- The decision here was to store tje complete state for undo (like removed concept + relationships) instead of trying to reverse operations. This allowed for much simpler undo logic and can handle complex cascading deletions. While this would take up more memory, I had justified this because correctness is more important than memory efficiency, not to mention these mental models are small enough that memory overhead is negligible.
- QSS Stylesheets vs. Inline Styling
    - Instead of creating inline setStyleSheet(), I created a centralized QSS file. This was so it can be the single source of truth for styling and no code changes were needed for UI updates. While this does make it harder to do conditional styling, I justified this because the maintainability is more important than code locality.

5. System Features Summary:

Qlink has a breadth of different features jam packed into this semester long project. The core features implemented are as follows:

1. Interactive Graph Visualization
    a. Nodes represent concepts with labels
    b. Edges represent relationships
    c. Drag nodes to reposition
    d. Zoom in/out with mouse wheel
    e. Pan by dragging background
    f. Context menus on right click
2. Natural Language Command Processing
    a. Text input panel for commands
    b. "add concept [name]" - creates new concept
    c. "remove concept [name]" - deletes concept
    d. "connect [concept1] to [concept2]" - creates relationship
    e. Command history with success/failure logging
    f. Syntax validation and error messages
3. Graph Modification System
    a. Full Undo/Redo System
        i. Keyboard shortcuts: Ctrl+Z (undo), Ctrl+Shift+Z (redo)
        ii. Menu items with dynamic tooltips showing what will be undone
        iii. Multi-level undo stack (unlimited depth)
        iv. Works for all operations (add, remove, connect, delete)
        v. History cleared on new model or load
4. AI-Powered Link Prediction
    a. Common Neighbor algorithm
    b. Jaccard Coefficient algorithm
    c. Preferential Attachment algorithm
    d. Confidence scores for each suggestion
    e. Sortable/filterable suggestion list
    f. Accept/reject individual suggestions
    g. Batch suggestion generation
    h. Cohere LLM integration to get relationship/ concept suggestions and explanations
5. Context Sensitive Menus
    a. Right-click on concept: Delete, AI Explain, Generate Description
    b. Right-click on relationship: Delete, AI Explain
    c. Right-click on background:
    d. Add Concept at cursor position
    e. Confirmation dialogs for destructive operations

6. File Operations
    a. New Model (Ctrl+N)
    b. Open Model (Ctrl+O) - JSON format
    c. Save Model (Ctrl+S)
    d. Save As (Ctrl+Shift+S)
7. Modern UI Styling
    a. Centralized QSS stylesheet (resources/styles/modern.qss)
8. Statistics and Validation
    a. Concept count
    b. Relationship count
    c. Average degree
    d. Connected component analysis
    e. Orphaned concept detection
9. Model Persistence
    a. JSON serialization/deserialization
    b. Stores tags and metadata
    c. Relationship types and weights
    d. Model name and timestamp
10. Error Handling
    a. Custom exception hierarchy (QLinkException)
    b. User-friendly error messages
    c. Logging to status bar
    d. Validation before destructive operations