# Design Document: Pattern Call Management System

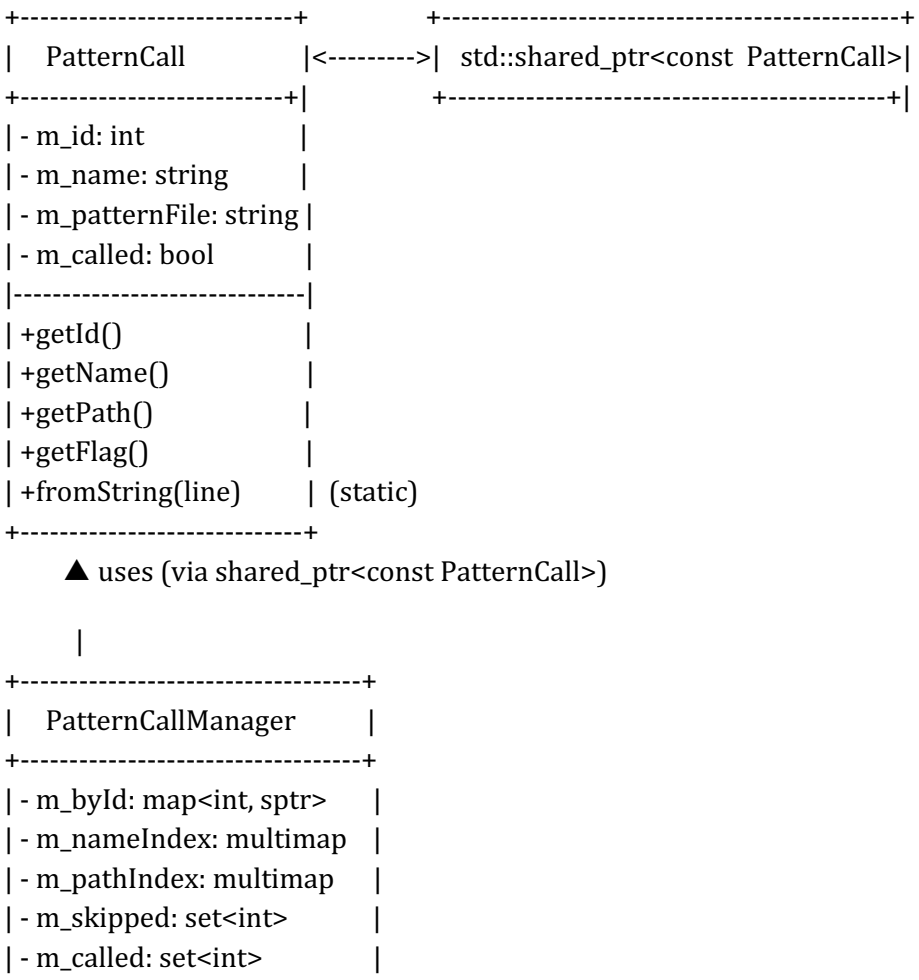
## 1. Overview

The Pattern Call Management System is designed to manage and query a collection of pattern call definitions represented in text form. The key responsibilities of the system include:

- Parsing pattern call definitions from input files.
- Storing and managing pattern call objects.
- Providing efficient lookup mechanisms by ID, name, file path, and call status.
- Writing pattern calls to output files.

Modern C++ principles are used, including std::shared_ptr to enable shared ownership and automatic memory management. The pattern call data is made immutable via shared_ptr<const PatternCall>.

## 2. Class Diagram



```
|-----------------------------------|
| +loadFromFile(path)               |
| +writeToFile(path)                |
| +addPatternCall(sptr)             |
| +getById(id)                      |
| +getByName(name)                  |
| +getByPath(path)                  |
| +getSkipped()                     |
| +getCalled()                      |
+-----------------------------------+
```

## 3. Class Descriptions

### 3.1 PatternCall

Represents a single pattern call instance with the following properties:

- Data Members:
  - m_id: A unique integer identifier.
  - m_name: Descriptive name of the pattern.
  - m_patternFile: Path to the pattern definition file.
  - m_called: Boolean flag indicating whether the pattern was invoked.

- Key Methods:
  - fromString(const std::string& line): Parses a line into a PatternCall object. Returns nullptr if the line is invalid.
  - friend operator<<: Outputs the object in a serialized format.
  - Getter functions provide read-only access to fields.

### 3.2 PatternCallManager

Handles storage, indexing, and querying of pattern call objects.

- Data Members:
  - m_byId: Primary storage of shared_ptr<const PatternCall>, keyed by ID.
  - m_nameIndex: Multimap from name to IDs, allows duplicate names.
  - m_pathIndex: Multimap from path to IDs, allows duplicate paths.
  - m_skipped, m_called: Sets of IDs indexed by their call status.

- Key Operations:
  - loadFromFile(filePath): Reads and parses each line. Valid entries are stored via addPatternCall().
  - addPatternCall(ptr): Adds a pattern call to all relevant containers.
  - getById(id): Returns the pattern call for a given ID.
  - getByName(name): Returns all pattern calls with a matching name.

- getByPath(path): Returns all pattern calls from a particular file path.
- getSkipped(), getCalled(): Returns lists based on call status.
- writeToFile(filePath): Serializes and writes all pattern calls to a file.

## 4. Design Benefit

### 4.1 Use of shared_ptr<const PatternCall>

- Ensures **shared ownership**, simplifying memory management.
- Guarantees **immutability**, enhancing safety and predictability.

### 4.2 Multiple Indices

- Allows **O(log n)** or **O(1)** average-time lookups by different fields.
- Maintains consistent and efficient access patterns.

### 4.3 Separation of Concerns

- PatternCall is a lightweight data class.
- PatternCallManager handles persistence, indexing, and querying.

## 5. Improvement potential

- Add PatternCallValidator class to improve input validation.Support serialization formats beyond csv (e.g., JSON).
- Use std::unordered_map for better average performance (if ordering is not needed). Consider thread-safe access if used in concurrent environments.
- Enhanced logging mechanism storing the result in logger file for future analysis

## 6. Assumptions

### 6.1 Well-Formed Input Data

- The input file contains well-structured lines. The malformed lines can be cleanly skipped via fromString().
- The format and fields (e.g., ID, name, path, called flag) are known in advance.
- Each tuple is present in each line.

### 6.2 PatternCall Objects Are Immutable Post-Creation

- Once a PatternCall is constructed and stored, it is not modified. This supports the use of std::shared_ptr<const PatternCall>.

### 6.3 Lookups Are Frequent, Modifications Are Infrequent

- The design is optimized for read-heavy usage. It assumes that objects are added once and queried many times, making maps and sets appropriate.

### 6.4 IDs Are Unique

- Each `PatternCall` must have a unique ID. Duplicate IDs are not handled and would result in overwrites.

### 6.5 Memory Is Not Constrained

- The system assumes sufficient heap space is available for storing multiple smart pointers and containers.

### 6.6 Thread safety

- The system is not designed to be thread-safe. It is assumed to run in a single-threaded environment**.** If multi-threaded access becomes a requirement, synchronization mechanisms (e.g., mutexes) will need to be introduced to protect shared resources.

## 7. Trade-Offs

### 7.1 Shared Pointers vs. Raw Pointers or Values

- `shared_ptr` adds overhead due to reference counting but simplifies ownership and lifecycle management. This is acceptable for simplicity and safety, but may not be ideal for high-performance.

### 7.2 Multiple Indices vs. Memory Usage

- Maintaining `m_nameIndex`, `m_pathIndex`, `m_skipped`, and `m_called` adds memory overhead. This trade-off is accepted in order to achieve faster query.

### 7.3 Using std::map Instead of std::unordered_map

- The current use of `std::map` ensures ordered iteration. Switching to `unordered_map` would provide better average performance but lose order guarantees.

## 8. Complexity of the queries

### 8.1 Query by ID

- Look up is taking place over `std::map` so the time complexity is `O(log n)` where n is the number of elements in the map.

### 8.2 Query by Name / Path

- Calling `equal_range()` over `std::multimap` has the complexity `O(log n)` where n is the number of elements in the multimap.
- Iterating over the `range` takes place `k` times, where `k` is the number of items in the range
- Lookup over `std::map` is of complexity `O(log n)`
- `push_back` in a vector is of complexity `O(1)`
- Total complexity = `O(log n)+ k * [O(log n) + O(1)] = O(k*log n)`

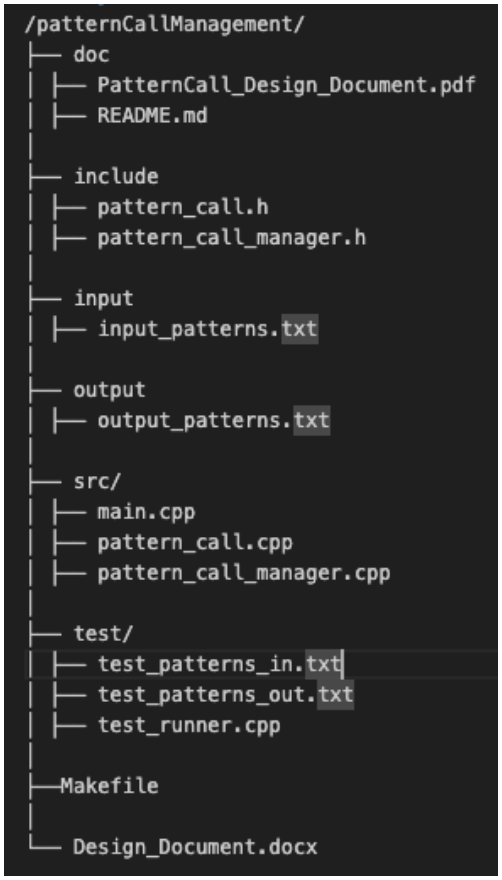
### 8.3 Query by flag (called / skipped)

- Iterating over the (`called/skipped`) `std::vector` takes place `k` times, where `k` is the number of items in the vector
- Lookup over `std::map` is of complexity `O(log n)`
- `push_back` in a vector is of complexity `O(1)`
- Total complexity = `k * [O(log n) + O(1)] = O(k*log n)`

## 9. Time Distribution

| Phase | Effort | Explanation |
|---|---|---|
| Design | 20% | Time spent on,<br>- analyzing requirements<br>- deciding on data structures and indexing strategies<br>- immutability guarantees |
| Coding | 35% | - writing header, source and make files |
| Testing | 15% | - verifying parsing correctness<br>- checking querying logic<br>- testing corner cases |
| Refactoring | 20% | -deciding on `shared_ptr<const T>`<br>- updating Testcases accordingly<br>- handling erroneous inputs |
| Documentation | 10% | - writing design document<br>- writing README |

## 10. Project structure




## 11. Conclusion

This design cleanly separates data modeling and management logic while leveraging modern C++ features like smart pointers and const-correctness. It ensures safety and performance for managing pattern call data.