Singleton Pattern

The Singleton pattern ensures that there is only one instance of a class and provides a global point of access to that instance. In the application, the `DatabaseManager` class follows the Singleton pattern.

Example:

```
class DatabaseManager:
    def __init__(self, database_name):
        self.conn = sqlite3.connect(database_name)
        self.create_table()

    # Other methods...
```

In this example, only one instance of DatabaseManager is created per TimeTracker instance. This ensures that there is a single database connection throughout the application. The Singleton pattern helps manage resources efficiently.

Builder Pattern

The Builder pattern separates the construction of a complex object from its representation. In the application, a builder-like approach is seen in the construct_query method of the TimeTracker class.

Example:

```
class TimeTracker:
    def construct_query(self, start_date, end_date, task, tag):
        query = "SELECT * FROM time_records WHERE 1"

        # Building the query based on parameters...

        return query
```

Here, the construct_query method dynamically constructs a SQL query based on the provided parameters. This builder-like behavior allows flexibility in creating different types of queries without modifying the class itself.

Strategy Pattern

The Strategy pattern defines a family of algorithms, encapsulates each algorithm, and makes the algorithms interchangeable. In the application, a strategy-like approach is observed in the construct_query method of the TimeTracker class.

Example:

```
class TimeTracker:
    def construct_query(self, start_date, end_date, task, tag):
        query = "SELECT * FROM time_records WHERE 1"

        # Different strategies for building the query based on parameters...

        return query
```

The construct_query method encapsulates different strategies for constructing the query based on the parameters provided. This allows the algorithm for constructing queries to be easily extended or modified without altering the main class.

Template Method Pattern

The Template Method pattern defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure. In the application, the generate_report method in the TimeTracker class follows the Template Method pattern.

Example:

```
class TimeTracker:
    def generate_report(self, start_date, end_date):
        records = self.query_time(start_date=start_date, end_date=end_date)
        report = f"\nTime Usage Report ({start_date} to {end_date}):\n"

        # Template method structure...

        return report
```

The generate_report method defines the overall structure of generating a time usage report. Subclasses (or specific method calls) can override or extend specific steps of the report generation process without changing the main structure. This promotes code reuse and maintainability.