Modular Task Reporter and Reminder

Final Report

Course: SWENG 421- Software Architecture

Group Members: Dipen A. Rathod

# UML Class Diagram

The complete UML class diagram with highlights to show the designed patterns used from the required list is shared here. The diagram is also shared separately for better visibility.

The Future pattern is used in various areas and, therefore, is not highlighted.

A diagram of a computer

AI-generated content may be incorrect.

# Functional Requirements and Their Fulfilment

This section shares the functional requirements for this project and discusses how they are satisfied with the design choices.

### Functional Requirements

The following are the functional requirements the final application must satisfy:

FR1) Create, edit, and delete reminders

FR2) Create, edit, and delete reports

FR3) Schedule reminders and reports for delivery according to the frequency the user decides

FR4) The customizable schedule should allow selecting repeat type, days of week, start and end dates, and time

FR5) Deliver reminders and reports notifications using desktop notifications

FR6) Import reminders from external sources, such as Google Calendar. Only those reminders that were not previously imported must be brought in

FR7) Collect weather information using Open Weather Map

FR8) Support Basic and Composite variants of reminders and reports to allow users to group many reminders into one and group multiple reports into one

FR9) Information Collector Plugins should require additional configuration when used in a report

FR10) Children of composite reminders must be triggered for delivery first, and the parent composite reminder can be delivered only after all its children are delivered

FR11) A composite report must wait for all its children’s reports to collect their information. The composite report description will be made using information from each of its children.

FR12) All reminder information, report information, and plugin configurations, should be stored as JSON with read/write lock protection to protect data integrity in an asynchronous environment

FR13) Only report and reminder objects that are due should be notified to execute their delivery procedure.

## Discussion on the Fulfilment of Functional Requirements

This section discusses how each functional requirement is satisfied by the design choices. This section is split into variable-size chunks as a set of requirements may be satisfied by the same design choice.

### FR 1 and 2

Functional Requirements 1 to 2 are satisfied using the **Factory creational design pattern.** The Factory pattern allows for interface-driven development- objects of different classes implementing the same interface can be instantiated using the same factory.

Depending on the user's choice, a Reminder or Report form is presented. The user can enter the details for that object in the form, allowing for simple customization, such as the title, description, when the object is due for delivery, and how often it should be repeated. The customization required is not rule-based, and the Builder pattern needs to be introduced.

When editing a reminder or report, the user enters the changes into a form populated with values from the object being edited. A temporary object is created with fresh values (again with the Factory method). The values are copied to the original object if they are valid.

### FR 3 and 4

The customizable repeat schedule does not need a dedicated factory or any other design pattern. It is a simple object configured using user input. Due to it being an object inside a reminder and report object, its configuration GUI is embedded into the reminder and report forms.

### FR 5

All notification delivery medium objects are created using the **factory pattern** (Notifier factory). The user needs to configure these mediums in the settings tab of the application once. After configuration, the user can use these as a medium of delivery for their reminders and reports. When a notifier object is created for a report or reminder object, the notifier factory configures it using the user-entered setting when creating the medium object.

### FR 6

Importing external reminders uses the **factory**, **dynamic linkage**, and the **read/write lock design patterns**.

The factory method is used to instantiate an external reminder plugin (which requires a one-time configuration in the settings tab). These plugins are instantiated in the central class that handles all read/writes to the disk.

The dynamic linkage pattern creates coordination between an external reminder plugin and the central class (database manager). The external reminder object needs to know what reminders it has already converted to save time on not computing them again. Similarly, the database manager needs to update its records of what new reminders have been imported from external sources and not re-import them in the future.

The read/write lock pattern protects the JSON files from being updated by multiple external reminder objects simultaneously, leading to better data integrity.

### FR7 and 9

Collecting weather information from an external source is done by an information collector plugin. These plugins are created using the **Factory** pattern. These plugins require one-time configuration in the settings tab and run-time configuration every time they are to be used in a report.

For example, FR7 talks about weather collection using Open Weather Map (OWM). The API key for OWM is the one-time configuration. The location for which the weather is required will vary for each report and needs to be configured when being added to a report object. The user can enter the run-time configuration (location information) using a small pop-up form presented during report creation.

When an information collector object is created for a report object using the factory method, the factory sets all the required one-time and run-time parameters before returning the object to be added to the report.

### FR8, 10, and 11

Composite reminders and reports are supported using the **Composite, Future, and Factory design patterns.**

(FR8) Basic and composite reminders implement the same interface. Basic and composite reports implement the same interface. The composite concrete classes of each type have a list to store their interface (ReminderIF or ReportIF) type objects. The notification delivery method of composite classes is different from their basic variants.

(FR10) The composite reminder delivery method is overloaded to send out all its children reminders and then wait for them to be sent (using the await keyword and the Future pattern). Once all the children's reminders are delivered, the parent composite reminder is delivered.

(FR11) The composite report delivery method is overloaded to collect information from all its child report objects asynchronously. The Future pattern is used to collect all information from children's objects before building the composite report for delivery.

The composite variant is selectable in the Reminder or Report forms. The children's objects can be selected in that same form. The factory method takes care of adding the selected children objects to the reminder or report object.

### FR12

Reports, reminders, IDs of external reminders, and plugin configurations and information are stored in JSON files.

To prevent data integrity violations in an asynchronous environment, **read/write locks** allow multiple objects to read from the JSON files but only one to write to that file.

It is possible for two parties to try to update a database simultaneously. For example, when the user attempts to add a reminder and so does the external reminders manager, locking the database becomes necessary.

### FR13

The **observer pattern** can be used to notify the different interested parties of changes in the configuration of plugins, reminders, reports, etc.

For example, the notification manager can check whether it is time to trigger some reminders/reports and then only trigger those reminders/reports that are due.

# Non-Functional Requirements and Their Fulfilment

This section shares the non-functional requirements for this project and discusses how they are satisfied with the design choices.

## Non-Functional Requirements

The following are the functional requirements the final application must satisfy:

NFR1) The application should be a native Windows application, supporting Windows 11

NFR2) The application main page should be visible within 5 seconds of loading the application when using a machine with an i7 13700, 32GB DDR5 RAM, and a SATA HDD to store the application

NFR3) The system should use less than 600 MB of RAM

## Discussion on the Fulfilment of Non-Functional Requirements

This section discusses how each non-functional requirement is satisfied by the design choices. This section is split into variable-size chunks as a set of requirements may be satisfied by the same design choice.

### NFR1

C# Winforms with .NET Framework 7.3 was used to develop the project application. Furthermore, the OS of the development and testing machine was Windows 11, allowing me to check if all features worked on Windows 11.

### NFR2

These are the system specifications of the development and testing machine. The application should run on a similar machine in normal circumstances.

### NFR3

The use of object references and storing only text data was critical to minimizing RAM usage. Images, audio, and video were avoided for better RAM use.

# Miscellaneous Requirements

A few goals of this project which don’t fall into the functional and non-functional categories were achieved using some design choices. This section discusses how those miscellaneous goals were satisfied.

## Using External Reminders

Reminders from external sources had to be converted to a common type for complete integration with the application. The **Adapter pattern** was instrumental in converting external reminders to the common reminder interface, ReminderIF.

## Plugins and Scalability

The app was developed with modularity and scalability in mind. Users should be able to develop their own plugins for use.

The plugins (external reminder sources, information collectors, and notification mediums) need to implement their corresponding interface, and the user needs to make an entry for it in the relevant plugin directory (JSON file). The system will automatically pick it up.

The user needs to configure the plugin in the settings tab and can then freely use it.

Along with Factory, the fundamental patterns abstract super classes and interfaces were used to achieve this goal.

# Justification of Design Patterns Used

This section will summarize why patterns from a category were excluded. Why certain patterns were used has been justified in the previous sections.

## Creational Patterns (Factory Method, Builder, Abstract Factory or Prototype)

**Pattern Chosen:** Factory Method

The Factory design pattern is used to meet (wholly or partially) the following functional requirements: FR1, FR2, FR5, FR6, FR7, FR8, and FR9.

**The role of the Factory method in the context of the mentioned requirements is discussed above. This section will focus on why the other patterns in this category were not used.**

**Abstract Factory:** Factory is the better choice compared to Abstract Factory because we are not creating a set of related objects. Also, there is only factory to create concrete classes of an interface.

**Builder**: Factory is better than Builder here because an object’s internal structure does not change depending on any set of rules. All objects created using a factory have the same structure (all reminder objects have the same structure, all report objects have the same structure, all notification medium plugins have the same structure, etc.).

**Prototype:** Prototype is unnecessary here because we are not cloning objects anywhere in our application.

## Partitioning Patterns (Filter or Composite)

**Pattern Chosen:** Composite  
The Composite pattern is used to satisfy the following functional requirements: FR8, FR10, and FR11

**The role of the Composite pattern in the context of the mentioned requirements is discussed above.** **This section will focus on why the other patterns in this category were not used.**

**Filter:** The Filter pattern is not required because different operations need not be performed on the data in an arbitrary order. Operations in the application are synchronous or asynchronous, but they are predetermined for the required behavior. The order of execution is not changed based on user-input or any events.

## Structural Patterns (Bridge, Decorator, or Dynamic Linkage)

**Pattern Chosen:** Dynamic Linkage

The Dynamic Linkage pattern is used to satisfy Functional Requirement - FR6

**The role of the Dynamic Linkage** **pattern in the context of the mentioned requirement is discussed above.** **This section will focus on why the other patterns in this category were not used.**

**Bridge:** Since the program is being developed from the ground up, all related objects already share an interface, or at least a basic relationship with each other. Therefore, a patch like Bridge is not required to fill the gap between related classes.

**Decorator:** A class’s features are not extended by bundling it with other classes as one, therefore, Decorator is not required.

## Behavioral Patterns (Chain of Responsibility, Observer, State, or Visitor)

**Pattern Chosen: Observer**

The Observer pattern is used to satisfy Functional Requirement – FR13

**The role of the Observer** **pattern in the context of the mentioned requirement is discussed above.** **This section will focus on why the other patterns in this category were not used.**

**Chain of Responsibility:** There is no hierarchy for the Chain of Responsibility pattern to be used. Composite reminders and reports have different notification behaviors than their basic variants, but it is important to note that composite types may hold other composite types. The priority of children-composite objects in a parent is the same.

**State:** The program does not take a State-based approach to use the State pattern.

**Visitor:** All the necessary logic for each class is already present. Additional logic for each need not be placed in an additional class (that too at the cost of encapsulation)

## Concurrency Patterns (Scheduling, Read/Write Lock, Two-Phase Termination, or Future)

**Pattern Chosen:** Read/Write Lock and Future

The Read/Write Lock pattern is used to satisfy Functional Requirements – FR6 and FR12

The Future pattern is used to satisfy Functional Requirements – FR10, FR11 (and in many helper functions)

**The role of the Read/Write Lock and Future** **patterns in the context of the mentioned requirements is discussed above.** **This section will focus on why the other patterns in this category were not used.**

**Scheduling:** Since all reminders and reports have the same priority, there is no need for a scheduler to influence execution by enforcing a priority. As said above, composite reminders and reports have a higher priority than their children, but all children have the same priority. There is no need for a complicated mechanism to prioritize parent objects.

**Two-phase termination:** The two-phase termination pattern is unnecessary because an incomplete reminder/report is not saved. No system resources are wasted due to an interrupted operation.

# Conclusion

I hope the explanations made my reasoning behind the chosen patterns clear. The UML class diagram (with highlights) is shared separately. The code files are shared as a zip. The project presentation slides are shared as well.

As per our conversation on the discussion board, I am not including images highlighting the patterns used here. Those images can be found in the presentation slides.

Thank you for all your help this semester.