Modular Task Reporter and Reminder

Course: SWENG 421- Software Architecture

Group Members: Dipen A. Rathod

Motivation

This section aims to describe the problems faced and propose a solution for the same.

Current Problem

A task could be many things. Checking the weather forecast, a stock's performance, holidays, ordering an item online, completing an assignment, reading, meeting someone, etc.

As life gets busy, remembering all your tasks, especially when spread across various services, can be a hurdle, let alone accomplishing them.

Performing tasks such as viewing the weather forecast every morning is important, but the time spent on these repetitive actions can be invested in other tasks.

Time is of the essence, but trying to use it more effectively is daunting.

Proposed Solution

The proposed solution is a native Windows application that will allow a user to make reminders and reports on their machine. The users can import reminders from external services, such as Google Calendar.

Users can set up routines to send reports containing information from external sources, such as Open Weather Map for weather forecasts.

Users can have their reminders and reports delivered via any medium, for example, desktop and email. They can also choose the frequency of those deliveries, from daily to annually.

By delivering reminders and reports where and how the user wants them, the tool aims to become a hub to centralize the user's responsibilities and help automate repetitive tasks to save time.

Functional Requirements

This section provides a list of functions the software must perform. **Functional requirements that are satisfied by the required design patterns are stated, along with the reasoning for using a pattern.**

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.

Non-Functional Requirements

This section provides a list of the system's non-functional requirements

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 500 MB of RAM

Design Pattern Choices

This section lists the patterns chosen from the mandatory pattern lists.

Creational Pattern (Factory Method, Builder, Abstract Factory or Prototype)

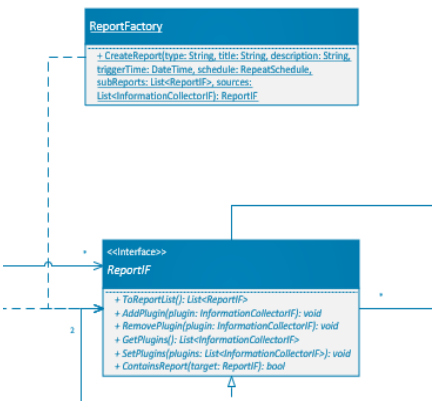
Pattern Chosen: Factory Method

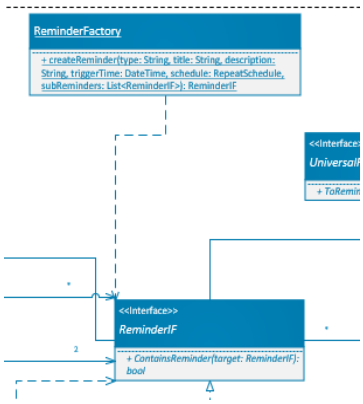
Reason: The Factory design pattern is used to create Report and Reminder objects **(FR 1 to 5)**. It encourages interface-driven development, which is useful when basic and composite variants use the same functions. There are two variants for Reminders and Reports each: Basic and Composite. Therefore, though they share similar interfaces, a factory is required to create different variations. The user can set up the repeat schedule and delivery mediums for the object during object creation.

The Factory Pattern is also used to instantiate information collectors, notification mediums, and external reminder source plugins based on user input.

Objects involved don't require rule-based customization, eliminating the need for **Builder. Abstract Factory** is useful when creating a group of similar objects together, which is not the case here. **Prototype** is not required as objects won't be cloned (deep copied)

(Screenshots of the class diagram focusing on the usage of factory for Reports and Reminders. Please refer to the main diagram, or the images for the composite pattern to see the concrete classes)





Partitioning Pattern (Filter or Composite)

Pattern Chosen: Composite

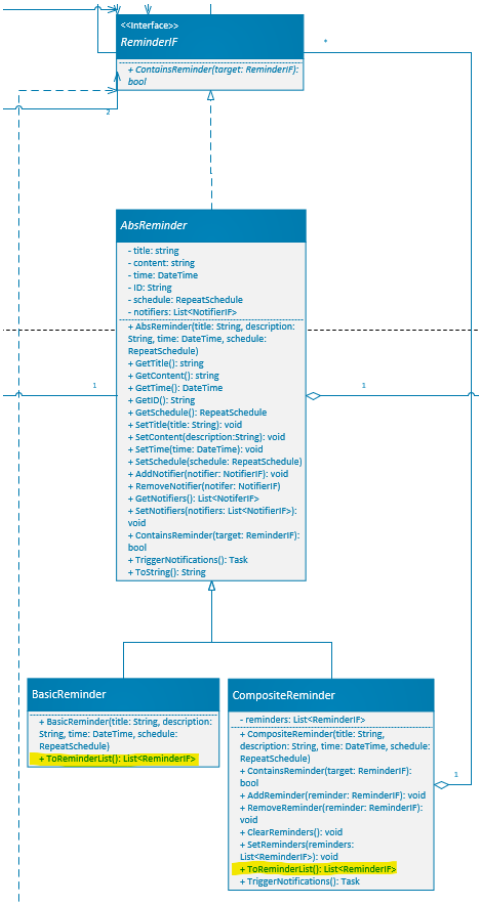
Reason: Some reminders may be built using multiple reminders using the composite pattern and are called composite reminders **(FR8).** Similarly, some reports may be built using multiple reports using the composite pattern and are called composite reports.

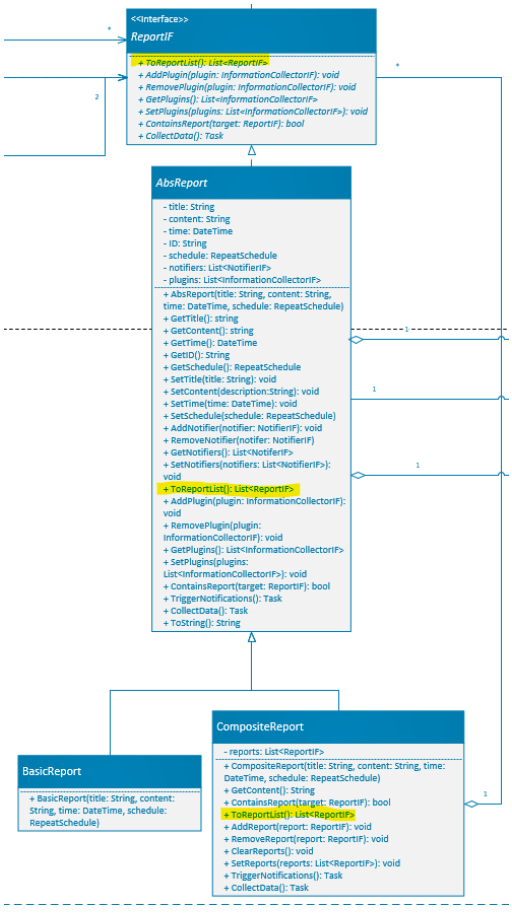
An example of a composite reminder is a multiple-phased deadline reminder- when the user receives multiple reminders simultaneously for future reminders of a project)

Composite reports are useful for building early morning digests comprising information scheduled to be delivered from other reports.

Filter is not required because different operations need not be performed on the data in an arbitrary order.

(Screenshots of the class diagram focusing on the usage of the composite pattern)





Structural Pattern (Bridge, Decorator, or Dynamic Linkage)

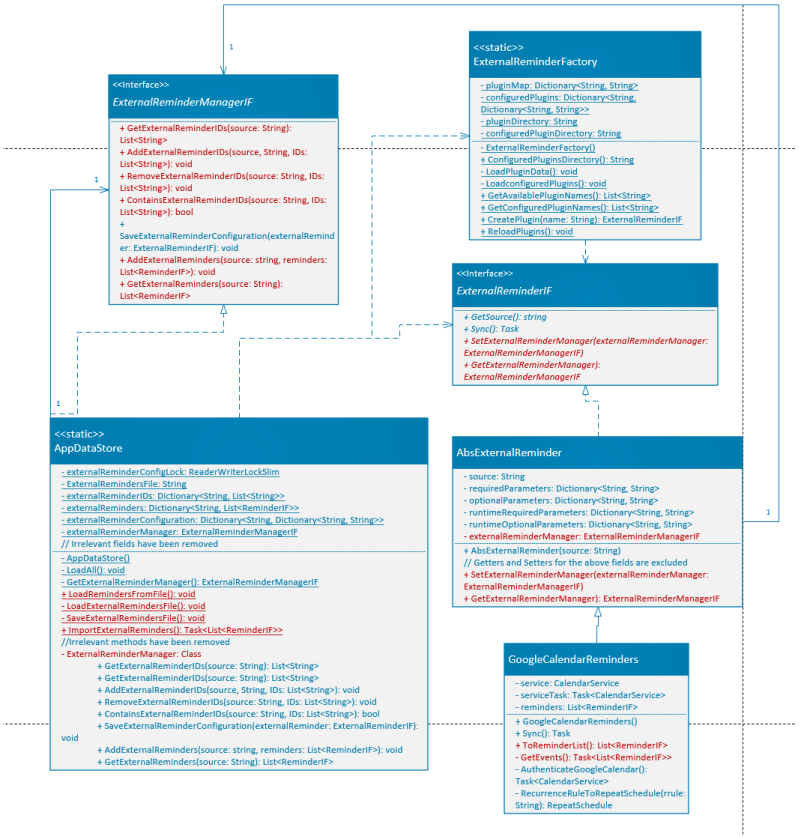
Pattern Chosen: Dynamic Linkage

Reason: Dynamic Linkage allows for modularity and a plugin system. This flexibility allows developers to add their own functionalities, such as reminder and information sources without much change to the underlying code. These plugins can be loaded at run time with only two things that need to be done by the programmer: Putting the plugin code file in the appropriate directory and then entering it as a plugin in the plugin-class file name mapping dictionary. The plugins can then be configured via the GUI.

For example, when reminders are imported via Google Calendar **(FR6),** the plugin importing google calendar reminders can interact with the environment (class that is managing read/write of reminders) to check whether a reminder has already been imported or not. In the future, plugins can also be used to trigger events in their parent environments, such as when to send notifications.

Since the program is being developed from the ground-up, **Bridge** is not required. A class's features are not extended by bundling it with other classes as one, therefore, **Decorator** is not required.

(Screenshot focusing on the usage of dynamic linkage. Please not only relevant classes are included. They are spread apart in the final UML class diagram)



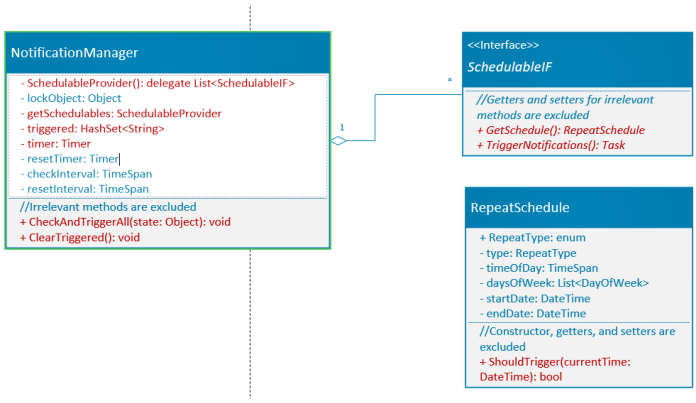
Behavioral Pattern (Chain of Responsibility, Observer, State, or Visitor)

Pattern Chosen: Observer

Reason: 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 **(FR13)**.

There is no hierarchy for the **Chain of Responsibility** to be used. The program does not take a State-based approach to use the **State pattern**. Like Bridge, since the program is developed from the ground up, it does not require a patch like **Visitor**.

(Screenshot showcasing the usage of the Observer pattern in the UML class diagram)



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

Pattern Chosen: Read/Write Lock, and Future

Reason: Read/Write Lock can be used to write to the database and the set. It can also modify the set that keeps track of past notifications. Because there is a possibility that two parties may 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. **ReaderWriterLockSlim** type objects are used to achieve this functionality. **(FR12)**

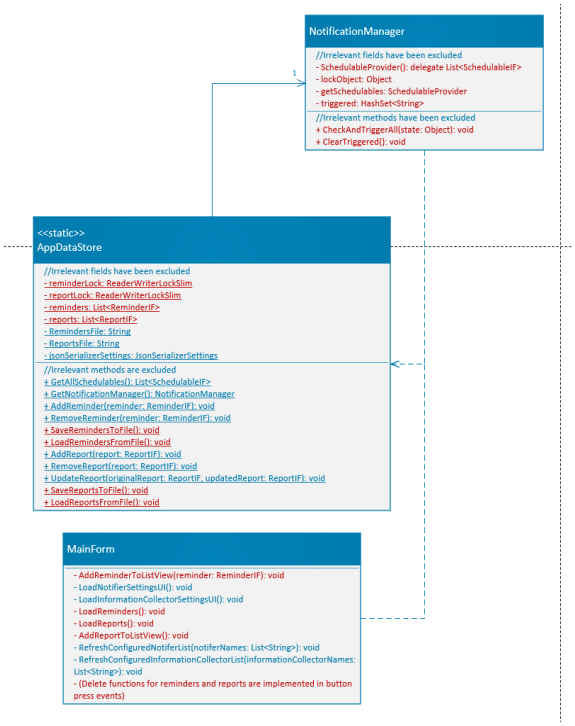
The **Future pattern** can collect information and trigger notifications of reports and reminders asynchronously. The parent object can use **await** to check if a task is complete before moving on with the program. The parent reminder will be delivered only after all its children's reminders are delivered. The parent report will build the report by collecting information from its children's reports. It will wait for a child to give its data before making the final report String. **(FR10, 11)**

The Future Pattern can also get information from external sources, such as weather forecasts and Google Calendar reminders, without blocking other program features. (It is a promise that data will be available, or at least an error as to why it could not happen once the retrieval process is triggered).

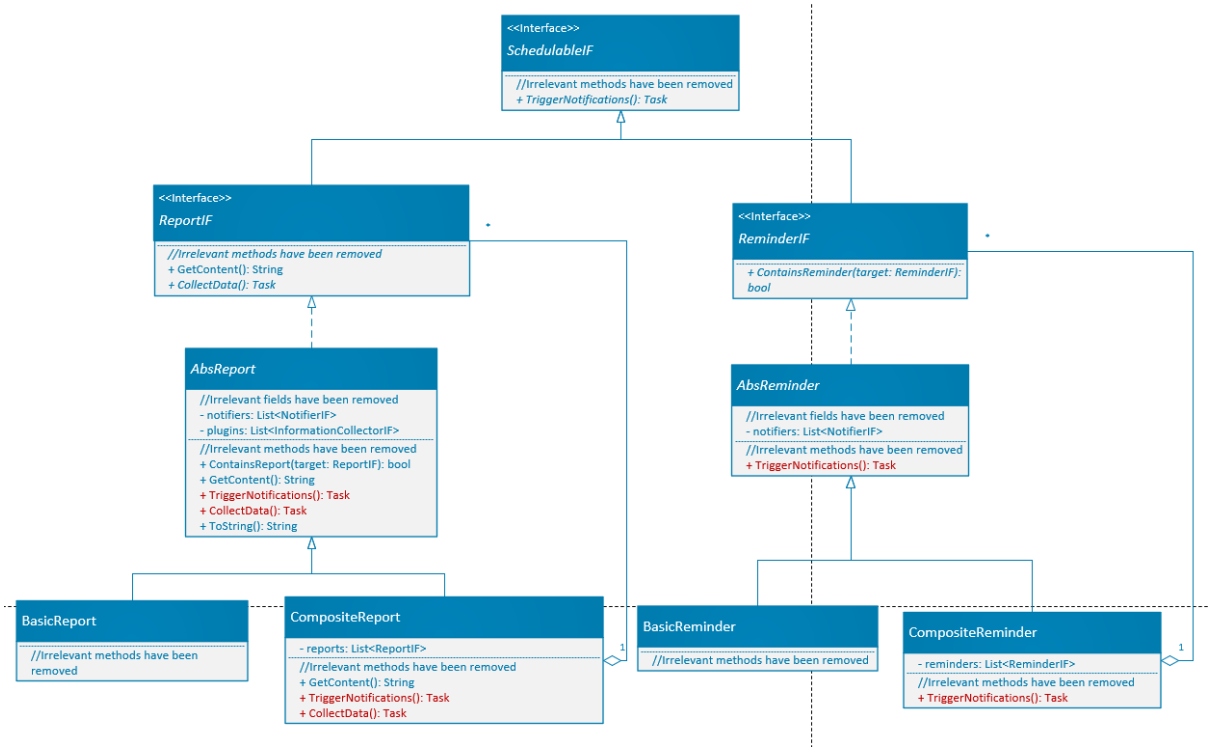
The scheduler pattern is unnecessary because any two objects scheduled have the same priority. The order of delivery does not matter.

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.

(Screenshot focusing on the usage of the Read/Write Lock design pattern)



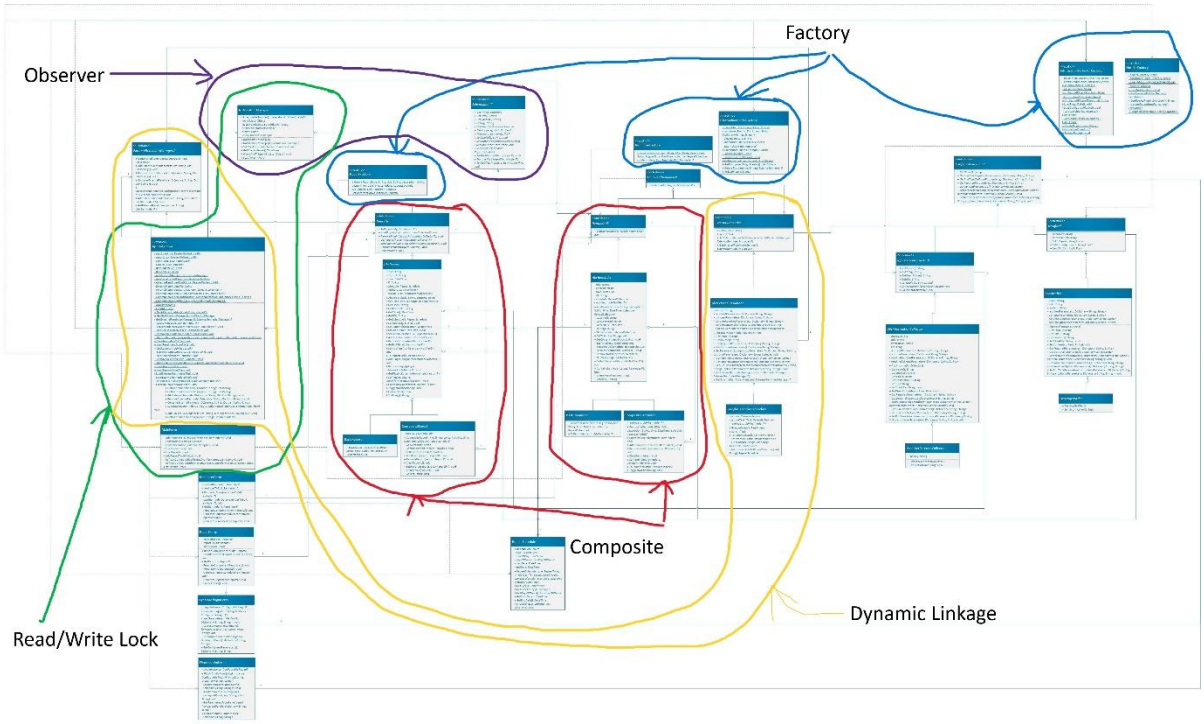
(Screenshot focusing on the usage of the Future design pattern)



Language Chosen: C#

I have chosen C# because it supports OOP principles like encapsulation, polymorphism, and inheritance. It has good resources and documentation for concurrency from Microsoft and the community. It is a preferred language for making native Windows applications. Additional libraries can also be imported via the NuGet Package Manager.

The UML diagram with all the design patterns highlighted. (It has also been submitted separately for better clarity)



Application Screenshots and Mapping

Figures 2 and 4 show the Reminders and Reports creation page (Factory Pattern).

Please focus on the components checklist box in the bottom-right of those images. If the user changes BasicReminder to CompositeReminder or BasicReport to CompositeReport (depending on what object we are creating), the check-list box will be enabled, and the user can add child reminders or reports respectively **(Composite pattern)**

Other mappings between the GUI and the design patterns can't be shown as they are at the code level

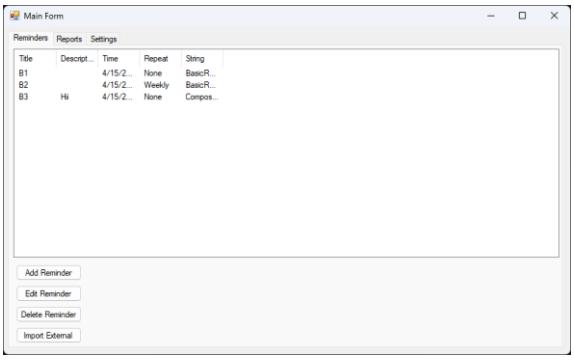


Figure 1: Reminders Main form Tab

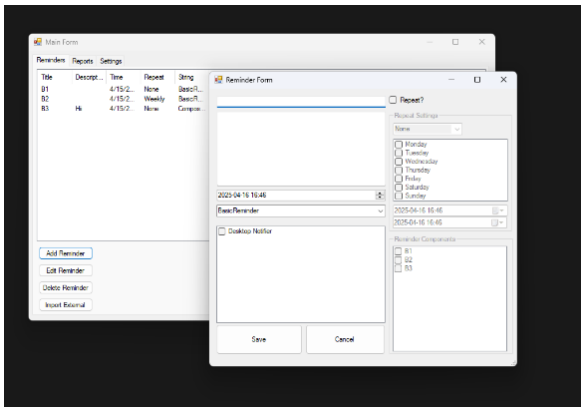


Figure 2: Reminders Creation Form

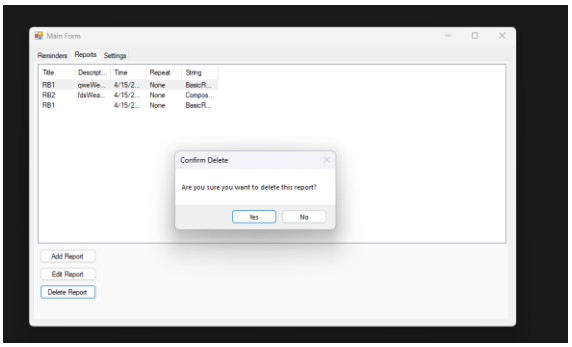


Figure 3: Reports Main form tab

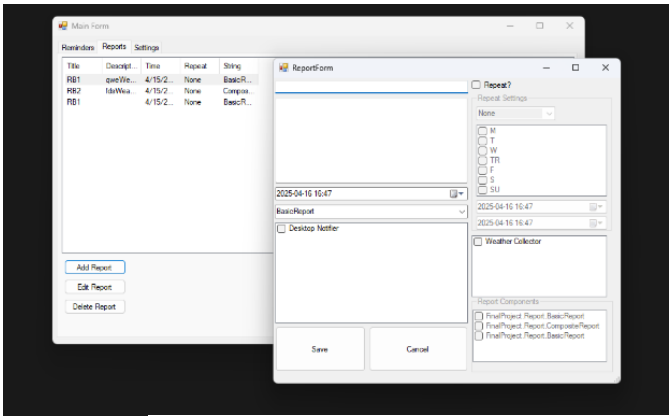


Figure 4: Reports Creation Form

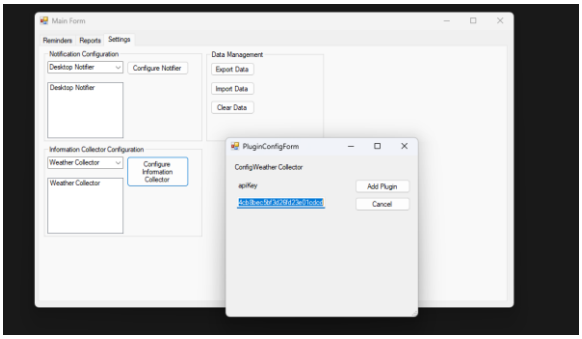


Figure 5: Weather Collector Plugin Configuration Page