DoDate Project — Iteration 1

User Stories

Description

DoDate is built around a single planner that represents all of the data for one user. The planner stores identifying information about itself and maintains the collections of data the user interacts with. These collections include a list of categories, which are used to group related work, and a list of events, which represent scheduled items tied to specific dates. The planner is responsible for locating, adding, updating, and removing items within these collections, as well as answering higher-level questions such as what work is overdue, what tasks are due soon, and what events are coming up.

Work is organized into categories, each of which represents a distinct area or purpose chosen by the user. A category stores its own identifying information, visual attributes, and description, and it maintains a list of tasks that belong to it. Categories are responsible for managing the tasks they contain, including creating new tasks, modifying existing ones, removing tasks, and summarizing incomplete work based on the current date.

Tasks represent individual units of work. Each task stores an identifier, a title, a description, a due date, a status value, and a weight that reflects the effort required. Tasks may also store a list of steps, allowing a single task to be broken into smaller checklist-style items that can be toggled independently. A task can be marked as part of today's focus and is able to determine whether it is overdue based on the current date.

In addition to tasks, the planner also keeps track of events. Events represent scheduled items tied to a specific date, and in some cases a start and end time. Each event stores its own identifying and descriptive information and may be connected to a broader area of work, allowing events to appear alongside related tasks when viewing the planner. Like tasks, events can be included in the user's focus for the current day so that time-sensitive commitments are visible at a glance.

The system includes an Application Controller that acts as a middle layer between the front end and the planning logic. When a user interacts with the app, the front end does not work with planners, tasks, or events directly. Instead, it sends requests to the application controller, which figures out what needs to happen and makes the appropriate updates to the system.

The application controller handles actions such as creating a planner, opening an existing one, saving changes, and providing information the front end needs to display. This keeps user interface concerns separate from the rules and behavior of the planning system, making the application easier to understand and maintain.

Saving and loading planner data is handled by another part of the system that focuses only on storage. When planner data needs to be read or written, the application controller relies on this component rather than handling files itself. Keeping storage separate helps ensure that planning logic and user interaction are not tied to how or where data is stored.

Instructions

In this iteration, you will create a UML class diagram that represents the structure described above. Use draw.io or another diagramming tool to create your class diagram. Your diagram should show the classes that make up the system, the attributes each class stores, the methods each class provides, and the relationships between the classes.

The emphasis in this iteration is on design, not implementation. You are expected to infer appropriate method signatures, visibility, and relationships based on the system description. The diagram should be detailed enough that another developer could understand how the system is organized and how responsibilities are divided, but it should not include implementation code.

This is an individual assignment, though you are encouraged to discuss design ideas with classmates.

Submission

Export or save your diagram as an easy to read file type (pdf, png, jpg, etc.). Make sure you do **not** have a transparent background or grid on your submission. **Your submission should be large enough to be clearly read.** If your diagram is unreadble or unable to be opened without external software (html, xml, etc.), you will recieve a zero.

Place the diagram inside the Design directory of your project.

Open a terminal and **make sure you are inside the repository directory** before running any Git commands.

Add the PDF file to Git by running the following command:

```
git add your_file_name.pdf
```

Commit the file with a clear message describing what you added:

```
git commit –m "Add Iteration 1 user stories"
```

Push your commit to GitHub:

```
git push
```

After the push completes, **open your repository on GitHub in a web browser and confirm that the PDF file appears in the repository**. If you do not see the file on GitHub, your work has not been successfully pushed. Reach out to a TA or the Instructor for help.

Grading Rubric (15 points)

Category	Poosible Points	Description

Category	Poosible Points	Description
Syntax	3	Correct UML notation, visibility, and readable layout
Classes	6	Classes include appropriate attributes and methods with correct visibility, data types for attributes, and parameter and return types for methods
Relationships	4	Relationships between classes are correctly represented, including ownership and multiplicity
Professionalism	2	The diagram is rendered clean, professional, and easy to read