CEng 445 Spring 2023 Projects Class Interface

The following class descriptions may not be final or not the best way to implement the projects. You can modify the arguments, add new methods as long as you can justify it.

In the projects, abbreviation CRUD implies implementation of the following methods:

Method	Description
`constructor(...)`	Create a new instance of the class from arguments
`get()`	Read. Return a textual representation of the item, you can use JSON like representation or simply records separated by punctuation (CSV)
`update(**kw)`	Update. Update the current item with new values. Updated parameters are given as keyword arguments. Other parameters remain the same
`delete()`	Delete. Delete the item.

In all projects, edited objects are maintained in global structures or singleton catalogue object of your choice. Each constructed object will be assigned a unique id and `getid(...)` method returns it. Users can call `attach(id)` method to get the object with given id and `detach()` method on object to release (name and paramater may change based on the topic). Depending on the project, this may result in getting notifications and not getting notifications. Also `listobject()` class methods give a list of objects to attach with their names or descriptions.

In all projects, users and authentication are required. However it is delayed until the second phase. In the second phase, the following interface should be implemented:

`User` class:

Method	Description
`constructor(username, email, fullname, passwd)`	Create the user with a password
CRUD	methods in addition to constructor
`auth(plainpass)`	Check if supplied password matches user password
`login()`	Start a session for the user, return a random token to be used during the session
`checksession(token)`	Check if the token is valid, returned by the last `login`
`logout()`	End the session invalidating the token

Use `hashlib` for storing the passwords. Do not store plain passwords. Use `uuid` module when you need to generate unique ids.

For the first phase, just implement a method `switchuser(userid)` to switch to an arbitrary user id, to show multi user functionality. Each created object will have current user is registered as implicity owner and will be returned in outputs (ie. `get()` methods).

There will be no notification in the first phase.

Also you do not need to implement any persistency in this phase. All objects are lost when program stops.

All coordinates starts at bottom left corner. When a rectangle type is required it is given as `(x,y, w,y)`, a rectangle with offset (bottom left) is at `(x,y)` and with given width and height.

1. Schedule Manager

In this project, users edit schedules and a `Schedule` consists of `Event` objects. An `Event` is always associated with a `Schedule` and constructed as:

Method	Description
`constructor(type, start, end, period, description, location, protection, assignee)`	Create an event between the given start and end date/time. `protection` defines who can view/modify the event, `assignee` is the list of users to invite the event.

Possible event types are: `MEETING, SEMINAR, LECTURE, APPOINTMENT, OFFICEHOUR`, and `FUN`.

A `Schedule` is the edited instance and have the following methods.

Method	Description
`constructor(description, protection)`	Create the Schedule instance that contains the events
CRUD	methods in addition to constructor
CRUD `Event`	All CRUD operatations of events as `newEvent()`, `getEvent()`, `updateEvent()`, `deleteEvent()`. First parameters of last 3 are event ids
`listEvents()`	List all events in the schedule. Only events that are readable by the user are listed.
`search(**kw)`	Search and return an iterator of the events matching a criteria and readable by the users.

The `search` is a combination of different event criteria. For string based paramters it is a case insensitive substring search. For example `search(description='lecture')` matches all events having the description containing `'lecture'` string. For `start` and `end`, it defines the interval overlapping with the event. All events starts before `end` or ends after `start` are matched.

Users can create `View` objects that are not shared. Their purpose is to have a unified view of multiple schedules.

Method	Description
`constructor(desciption)`	Creates a view
`addSchedule(schid)`	Adds schedule with given id to a view.
`deleteSchedule(schid)`	Remove schedule from the view
`setFilter(schid, **kw)`	Adds a filter to existing schedule in the view. `**kw` is sent to `search()` method
ov the `Schedule` and only matching items are listed in the view	
`removeFilters(schid)`	Remove all filters from the view
`listItems()`	List `Event` items of a view as an iterator of `Event` objects
`attachView(viewid)`	Sets the view as the main view. When view changes via updates on the matching schedule items, user is notified
`detachView(viewid)`	Detach from the view stopping notifications
`listViews()`	Class method to list all views of current user. Returns an iterator of view ids and descriptions as tuples
`getView()`	returns the view matching the id as `View` object

`protection` member of `Event` and `Schedule` objects are used to implement access control. In the first phase no implementation is required. You can leave them empty. In second phase, role based authentication will be implemented. The interface will be defined later.

2. Map Making Game

In this project users play on a global `Map` object. System has a repository of maps and user joins to one of them, releasing the previously joined map. The joined map contains all data of all players. `Map` class is also used for keeping track of the **teams view** of the map. For each team a `Map` is created and players update their local views on this objects so all team members can see it. A map has a background image and a list of associated objects. A team map starts

from blank background and as discovered it contains patches from original map. `Map` has the following members:

Method	Description
`constructor(name, size, config)`	Creates a map with given dimensions (width,height) tuple as `size`. The background image is given as a file path in the `config`. Background is blank if image is not provided
`addObject(name, type, x, y)`	Adds an object on the 2D grid. `type` is explained below.
`removeObject(id)`	remove the object with id
`listObjects()`	return all objects in the map with `(id, name, type, x, y)` tuple as an iterator.
`getimage(x,y,r)`	get the background image defined by the rectangle. Rectangle is defined as a `2*r` by `2*r` square centered at `x,y`
`setimage(x,y,r, image)`	set the background image patch at the rectangle.
`query(x,y,r)`	return all objects in a rectangular area as an iterator. Area is defined as a `2*r` by `2*r` square centered at `x,y`
`join(player,team)`	A player joins the game of the global map. `team` is a string for the team name. If team name exists, user is associated with the `Map` of the team, otherwise the `Map` is created.
`leave(player,team)`	A player leaves the game. Users have to call `join()` before leaving the map and `leave()` existing games before joining again.
`teammap(team)`	Returns the team map of the given team name

The objects on a map can have the following types:

- A `Player` is an object created per user. As user joins a map, it is inserted at default position `(0,0)`. It can move across the map. Players have a health value and a collection of objects that they can insert at their current position. When health gets to 0, player leaves the map.
- A `Mine` is a bomb trigerred when someone gets close to a proximity `p`. `Mine` starts when it is left on a map. It runs a loop of wait for a while to load, then wait until someone gets in the range, then explode. After running for `k` iterations, `Mine` is deleted, it disappears. When mine explodes, the players in proximity gets their health reduced with amount `d.p`, `d` and `k` are given as constructor parameters.
- A `Health` increases the health of players to parameter `m` when they get as close as 3 pixels. Some `Health` objects have infite capacity and remains forever in the map. Some of them are carried by users and disappear after being used. Players throw them on 5 pixels away so they can be

used by other users. The boolean constructor parameter `inf` defines this behaviour.
- A `Freezer` works like a `Mine` but stunns the player for `d` seconds.

The `config` parameter of `Map` constructor may contain the values in a dictionary:

- `image`: file path given the background image of the map. It should match the dimensions of the `Map`.
- `playervision`: default size of the square area that a user can query and get image of.
- `playerh`: default player health assigned when a player joins the map.
- `playerrepo`: list of items given to player when s/he joins the map. It is a list of tuples as `(cls, p0, p1, p2,...)`. `cls` is the type of the object and following values are constructor parameters.
- `objects`: list of items initially put on the map. It is a list of triples as `(x,y,object)`. Objects are constructed and put into this list.

`Player` instance is created and returned by `Map.join()` when a user joins a `Map`. The class will have the following methods:

Method	Description
`constructor(user, team, health, repo, map)`	Default values are assigned based on the configuration of the map
`move(DIRECTION)`	Player moves on a direction ('N,NW,W,SW,...') on the map. Each move will result on a query and image update of the team map.
`drop(objecttype)`	Player drops and object in its repository at the current position

When the players of different teams meet on a point, a competition or fight will take place. The rules are left for the second phase.

In the following phases, the objects in the `Map` will work in different threads to implement the timing and they are going to synchronize on proximity based events. For this phase, there will be no timing, proximity triggers and notifications.

3. Virtual Bulletin Board

The main edited class is a `Board` in this project. A `Board` contains `Message` objects. The `Board` will have two dimensions and the `Message` objects will be pinned on 2D points. A `Message` will have the following interface:

Method	Description
`constructor(owner, title, category, content, colors, x, y, width, height, expiration, protection)`	`x,y,width,height` values are integers, expiration is a `datetime` object, `owner` is a `User`. `title`, `category` and `content` are arbitrary strings. `protection` defines a list of users and their permissions.
CRUD	CRUD operations in addition to the constructor
`access(user, accesstype)`	Returns if `user` can do the `accesstype` operation
`increment()`	Increment view count of the object

By default a message is assumed to have a `VIEW` access. Based on the `protection` parameter, it can change to `NONE` or `WRITE` (read and write). The parameter is a list of tuples. Each tuple has a list of users and a string denoting the permission as `([user1,user2,user3],'NONE')`. `access` method matches the current user to first of the lists. If there is no match, it is assumed to be `VIEW`. User `*` matches all users. CRUD operations are subject to access control as `update` and `delete` requires `WRITE` permission.

The `Board` has the following interface:

Method	Description
`constructor(owner, title, colors, width, height, protection)`	`width,height` values are integers, `owner` is a `User`. `title`, `protection` defines list of usernames that can attach the board.
CRUD	CRUD operations other than the constructor
`pinMessage(message, x, y)`	Pin (add) the message on the virtual board. `WRITE` permission on the board is required.
`unpinMessage(messageid)`	Unpin (remove) the message on the virtual board. `WRITE` permission on the board and the message are required
`getMessage(messageid)`	Get the message with given id. `VIEW` permission for the message is required.
`query(rect, category, title, content)`	Queries the given rectangular area on board for matching values. The values are case insensitive substrings of the matched messages. Results in an iterator. Only messages with `VIEW` permission are listed
`firstArea(message, x,y)`	It scans the board for pinning the message for and offset `>=x, >=y`. Returns the first offset that pinning is possible (neither an overlap nor lack of permissions). You can make an exhaustive search.

Method	Description
`addAccess(rect, users, protection)`	Adds an access control rule for the region. `users` is a list of users, permission is set of `READ`, and `WRITE` values
`getAccessList()`	Get the list of access control rules
`updateAccess(itemid, **kw)`	Updates the access control rule with the given id
`deleteAccess(itemid)`	Delete the access control rule with the given id

The board makes an area overlap check for all pin requests and denies if there is an overlap. `firstArea` can be used to find an available area on the board.

By default, all areas the board is assumed to be a readable and writable by everyone. The access lists change it by defining rectangular regions and user based access control. The list contains a rectangle, a user list and one of `READ`, `WRITE`, `READWRITE`, or `NONE`. On pin and unpin requests, the request is matched against all items in the list for overlapping rectangles and username. The first match is considered as the permission of the user for the message area. Adding a value `((0,0,w,h), ['*'], 'READWRITE')` to access list give the board full access to all users.

Users can create their `View` objects to attach to more than one boards. Each view object is associated with a list of board and query tuples. Methods of `View` are:

Method	Description
`constructor(owner,name)`	
`addBoard(board, **query)`	Adds the given board to the view with given query parameters
`removeBoard(board, **query)`	Remove all queries to the board. There might be more than one query associated to a board. Removes all.
`listBoards()`	List all boards in the view with their query parameters.
`getMessages()`	Gets all messages matches from all boards in the view

The protection or the access control is the essential part of this project. The boards will have a list of area based protections. By default the whole board has `READ` and `WRITE` access.

The expiration of the message will not be implemented on this phase.

Room Reservation System

The `Organization` class is the main class that is collaboratively edited in this project. It is basically a collection of `Room` and `Event` items.

A `Room` object represents a room to be reserved in the organization. It has the following methods:

Method	Description
`constructor(name, x, y, capacity, workinghours, permissions)`	A room has geographical coordinates wrt organization map and capacity and working hours as a datetime interval. `permissions` defines the users can reserve the room
CRUD	CRUD operations on a room

An `Event` can be anything like a `MEETING`, `LECTURE`, `CONCERT`, `SEMINAR`, `STUDY`, etc. Initially it has a description, duration, and a required capacity. Its star time and location is not fixed initially. When the room is assigned to the events those information will be set. It has the following methods:

Method	Description
`constructor(title, description, category, capacity, duration, weekly,permissions)`	Constructs a new event. `duration` is an integer in minutes. `capacity` is maximum number of people. If `weekly` is a datetime, it repeats every week until the given datetime. If `None`, it does not repeat.
CRUD	CRUD operations of `Event`

`Organization` has the following methods:

Method	Description
`constructor(owner,name,map)`	A new organization is constructed with an optional background image, map of the area.
CRUD	CRUD operations
RUD on `Room`	Organization has a collection of `Room` objects. `getRoom(id)`, `updateRoom(id,...)`, `deleteRoom(id)`
reserve(event, room, start)	Reserve the `Room` for the `Event` if `Room` is available for the event in `start` to `start+duration` interval. User should have `WRITE` permission for the `Event` and the `Room`. For periodic events, `Room` requires `PERWRITE` permission.

Method	Description
`findRoom(event, rect, start, end)`	Find and return `Room` objecst within the rectangle and available in the interval defined by `start` and `end`. Room capacities should match the `Event`. It returns an iterator.
`findSchedule(eventlist, rect, start, end)`	It tries to find a schedule for a group of events. The schedule should be compatible with existing room assignments and it should not have any conflict.
`reassign(event, room)`	Change existing reservation of the `event` to new `room`. If request is valid, old reservation is cancelled and new reservation is made.
`query(rect, title, category, room)`	It returns `(event, room, start)` tuples matching the query as an iterator. `Room` should be in the rectangle, `title` and `category` should match (as a substring) the `Event` information. `room` is the specific `Room`. Either `room` or `rect` is specified.

Users can create a private `View` instance to observe the events in the organization. Which is a collection of query results of the organization. It has the methods:

Method	Description
`constructor(owner)`	
`addquery(organization,**kw)`	The `query()` method parameters are registered in the view with the organization object
`delquery(qid)`	Query is deleted from the view
`roomView(start, end)`	The queries in the view are executed and a room based result is generated. A dictionary with room titles as keys are returned. Each room will have the query results reported for the room are listed. Rooms without an event are skipped.
`dayView(start, end)`	The queries in the view are executed and a daily result is generated. A dictionary with days are returned. Each day will have the query results reported for the day are included. Days without an event are skipped.

The periodic events in the queries and views are listed only once with an indicator.