CEng 445 Spring 2023 Project Class Interface

Project Phases Description

In this first phase, you are going to implement **a class library and its unit tests** in Python. The phases of the project will be:

1. Class library
2. TCP service
3. Web service
4. Dynamic web application

There will be no **persistence** (i.e., objects saved to a database); **everything** will be **in memory**. **Additionally,** there will be no concurrent access.

In the second phase, you will implement concurrency, **persistence,** and make your application remotely accessible via a **text-based** TCP protocol.

In the third phase, you will convert your class library into a web service. A simple web **frontend** can be used to control your application. You will revise your unit tests to call operations over the web.

In the **fourth** phase, your application will use *websockets* to implement server push notifications and a better visualization of the objects in the application.

Generic Class Library

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

In the projects, the abbreviation CRUD implies the 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 a **JSON-like** representation or simply records separated by punctuation (e.g., 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, we need a **catalog** of current objects, which will also let us create new instances globally as a factory. In the following phases, clients can query and observe the objects in this central class or singleton object. You can call it `Directory`, `Repo`, or a name most suitable for the specific project topic.

Method	Description
`create(**kw)`	Creates a new editable object, assigns a unique id, and returns the id.
`list()`	Return a list (or iterator) of `(id, description)` **pairs** for editable objects.
`listattached(user)`	Lists the objects that the user has attached.
`attach(id, user)`	Returns the object with the given id from the `Repo`. The object is marked as *in use* as long as it is kept attached by any user.
`detach(id, user)`	The user detaches the object or stops editing and watching **it**. When the last user detaches **it**, the object will be in a *not in use* state.
`delete(id)`	The user deletes the object. All users should detach the object before deletion.

In a typical scenario, a client (user of the library) will `list` the set of objects, `attach` to an object to get the actual Python object by its id, call methods on it, and `detach` from it when done. There is no concurrency in the first phase, but in the following phases, the same object can be attached by multiple connections.

For the first phase, **persistence** is not required. **When** implemented, the object will be loaded into memory when the first user attaches to it and saved to disk/database when the last user detaches from it. Each constructed object will be assigned a unique id by `create()`, and calling the `getid(...)` method on the object returns it.

The project topics are chosen to contain instance notification scenarios. When an object is updated, all observers will get an immediate notification. In the first phase, you can show this case with terminal output. In the second phase, all attached clients will be notified properly.

Also, you do not need to implement any **persistence** in this phase. All objects are lost when the program stops.

Project Topics

1. Collaborative Portable Text Editor for Documents

The editor is for a **JSON-based** rich text document format for simple documents. The format consists of markups and text in a nested form as:

```
{ "markup": "document",
  "id": "_124131",
  "children": [
    { "markup": "paragraph",
      "id":"_43411",
      "style" : "align:right, border: 1pt",
```

```
      "children":[
        { "markup": "text",
          "id":"_54221",
          "content": "Hello"},
        { "markup": "strong",
          "id":"_8675234",
          "children": [ {
            "id":"_12314111",
            "markup": "text",
            "content": "CENG"} ]
        },
        { "markup": "text",
          "id": "_226242",
          "content": "445"}
        ]
    },
    { "markup": "image",
      "id":"_625252",
      "src": "iVBORw0KGgoAAAANSUhEUgAAAA8AAAAPCAMAAAAMCGV4AAAAAXNSR0IB2cksfwAAAARnQU1BAACx
jwv8YQUAAAAgY0hSTQAAeiYAAICEAAD6AAAAgOgAAHUwAADqYAAAOpgAABdwnLpRPAAAAA9QTFRF
AAAA/wAA////////////dD6a1gAAAAF0Uk5TAEDm2GYAAAABYktHRACIBR1IAAAACXBIWXMAAAsT
AAALEwEAmpwYAAAAB3RJTUUH6QoVCzUkQo+WYAAAAERJREFUCNddTwkOACAIAv7/6FLSaSznOGwK
GMKCxE1FLno5RzheCeFZYoWdzxFtADaahpDTUf7QbtMSxsopzIWp/6DXD1gIALwZq+b4AAAAAElF
TkSuQmCC"
    }
  ]
}
```

The markups supported are: document (only top), paragraph, image, strong, list, table, row, cell, item, and text. All markups other than `text` and `image` can have children. Documents can be accessed through **dictionary-style** selectors. For example, `d['0/1/text']` in the above document (as the `d` object) **would give** the text `CENG`.

Method	Description
constructor	Constructs an empty document, with only `document` markup, no children.
`importJson(jsonstr)`	Constructs a new document from a JSON string if it is compatible with the document layout.
`__getitem__(path)`	Return the `Document` at the given path. If the leaf is `text`, `style`, or `content`, **return** the string.
`__setitem__(path, markup)`	Inserts an empty markup at the given position. If the leaf is `text`, `style`, or `content`, **set** the attribute.

Method	Description
`__del__(path)`	Remove the component at the given path.
`insert(path, document)`	Inserts a `Document` object at the given position. The leaf should be an integer index.
`getid(idstr)`	Search and return the element with the document id in the `id` attribute.
`search(text)`	Make a substring search in the `text` markups and return the element.
`parent()`	Return the parent of the current document if **it is** not the root element. Return `None` for the root element.
`draw()`	Draw the visual (HTML) representation of the document.
`html()`	Convert the document into an HTML string.
`json()`	Convert the document into a JSON string.
`watch(obj)`	Adds the `obj` as an observer for the document or subtree element. `obj.update(html)` will be called on document update. Observers of all elements up to the root will be notified.
`unwatch(obj)`	Remove the `obj` from the list of observers.

2. Sport Events Monitor

The project will implement a basic class library for tournament, game, and team tracking. There are 3 different classes to edit in the project: `Team`, `Cup`, and `Game`. A `Cup` is a collection of `Games`; a `Game` has two teams and score information.

`Team`:

Method	Description
`constructor(name)`	Constructs a team object with the given name.
`__setitem__(key,value)`	Sets a generic team attribute like `year`, `info`, `country`, etc.
`__getattr__(key)`	**Returns** the given attribute.
`__delattr__(key)`	**Deletes** the given attribute.
`addplayer(name, no)`	Add a player with the given name and number.
`delplayer(name)`	Delete a player with the given name.

`Game`:

Method	Description
`constructor(home, away,datetime)`	**Creates** a game between the given team objects at a specific date and time.
`id()`	Get the identifier of the game.
`home()`	Return the home team object as a property.
`away()`	Return the away team object as a property.
`start()`	Game started.
`pause()`	Game paused, break or timeout.
`resume()`	Game resumed from pause.
`end()`	Game ended.
`score(points, team, player)`	Update (add points to) the score of the team, **attributed to** an optional player name.
`watch(obj)`	Adds the `obj` as an observer for the game. `obj.update(game)` **will be called on game updates**.
`unwatch(obj)`	Remove the `obj` from list of observers.
`stats()`	Return a list of game statistics.

The game **statistics** will return a nested dictionary similar to:

```
{ "Home": { "Name": "Fenerbahçe Beko",
            "Pts": 62,
            "Players": {"Baldwin IV":16, "Biberoviç": 12, "Tucker":17, ...}
          },
  "Away":{ "Name": "Anadolu Efes",
            "Pts": 62,
            "Players": {"Larkin":20, "Osmani": 12, "Bobua":15, ...}
          }
  "Time": "33:34.2",
  "Timeline": [ ( "00:10.2", "Away", "Larkin", 2),
                ("01:02.3", "Home", "Biberoviç", 3),...]
}
```

`Time` keeps the **wall-clock** time between the start of the game and the time of the `stats()` call, with paused intervals subtracted. If the game **has ended**, `Time` will be `Full Time`.

`Cup`:

Method	Description
`constructor(teams, type, interval)`	Construct a `Cup` class competition object with the given options explained below.
`search(tname,group, between)`	Search **for** and return a list of games with the given `tname`, `group`, or `datetime` interval as a tuple. **All parameters** are optional.

Method	Description
`__getitem__(gameid)`	Return the game with the given id.
`standings()`	Return the standings in a data structure. The structure depends on the type of the competition.
`gametree()`	Return a tree of games for `ELIMINATION` and `GROUP` type tournament playoffs.
`watch(obj,**searchparams)`	**Adds** `obj` as an observer of the tournament. When a game with the given search parameters starts or ends, it calls the `update(game)` method of the object.
`unwatch(obj)`	Remove the `obj` from list of observers.

The `Cup` type can be one of `ELIMINATION`, `GROUP`, or `LEAGUE`. The constructor randomly generates the games for the tournament as **follows**:

- **ELIMINATION** Matches pairs of teams randomly. Only winners **advance** to the next round. If the number of teams is odd, one **team** passes to the next round without a game. Only the first rounds of the competition have known teams. The following round games are constructed from **previous** games as `Game(game1, game2,...)` where the actual teams depend on the results of the other games.
- **LEAGUE** All pairs of teams play a game against each other. The winner depends on **score-based** sorting.
- **GROUP** Teams are divided into N groups **named** with capital letters. Each group is executed as a `LEAGUE` and, depending on M (the number of teams in the play-offs), the first $k = \lfloor \frac{N}{M} \rfloor$ teams from each group take part in the `ELIMINATION` rounds. $M-(k \times N)$ overall best teams also take part in the `ELIMINATION`.

All types have 2 game variants, such as `ELIMINATION2`, `LEAGUE2`, etc., where each **pair of teams plays** two games, **with** home/away switched.

The `standings()` result will depend on the type of the tournament:

- **League-based tournaments** will have a list of sorted `(team, won, draw, lost, scorefor, scoreagainst, points)` tuples. Sorting is based on points and score difference. The points are calculated as winner 2, draw 1, lost 0. You are free to add extra functionality for football.
- **Elimination** standings will return a dictionary of teams with the following structure:

```
{ 'Turkey':{ 'Round': 4,
              'Won': [('Latvia',82,65), ('Serbia',82,76)],
              'Lost': None
            },
  'Serbia':{ 'Round': 3,
```

```
            'Won': [('Finland',75,73)],
            'Lost': ('Turkey',76, 82)}
  ...
}
```

`Each team will be reported with the last round it played or **is currently** playing.`

- **Group** standings will return a hybrid value: A dictionary of league standings for groups with a `Groups` key, and elimination standings **with** a `Playoffs` key.

3. Trading Monitor

This project will let users monitor and track financial markets with stock, currency, and commodity prices, **along** with financial signals. An `Asset` is either a currency, commodity, or stock **that** has selling and buying prices changing over time. A `Statistic` is a calculated value over the prices of one or more assets in a time window (e.g., the last 24 hours). A `Signal` is a condition calculated from one or more **statistic** values. People use signals to make **sell-or-buy** decisions on signal points. To make demonstrations possible, we will assume our market works on statistics and signals using *seconds* as time units.

`Asset`:

Method	Description
`constructor(description, sell, buy)`	Construct an asset instance that will have values calculated in the market.
`update(sell,buy)`	Update the sell and buy prices of the asset; the current timestamp is recorded.
`values(seconds, end='NOW')`	Return the values in the last `seconds` seconds ending at the given time. It returns the samples in the given period.
`watch(obj)`	Adds an observer object. Each time a new value is added with `update()`, `obj.update(sell, buy)` is called with the new **prices**.
`unwatch(obj)`	Remove the `obj` from list of observers.

A `Statistic` object is a wrapper (decorator) of an `Asset` object. It gets an updated value from the wrapped asset and calculates the updated statistics from the sell and buy **values**. It behaves like an `Asset` object without the `update()` member. The same statistics of sell and buy prices are calculated **separately** and returned as sell and buy values as statistics. You need to implement at least 4 **subclasses** of the abstract `Statistics` **class** as **follows**:

- `MovingAverage(a,t)` **Moving** average of the asset values in the last `t` seconds. As the asset advances, the average is updated to include only the last `t` seconds.

- `StdDev(a, t)` **Finds** the **standard** deviation of values in the last `t` seconds.
- `Correlation(a1,a2,t)` **Correlation** between prices of two assets in the last `t` seconds.
- `Beta(a,t)` **Calculates** $\beta = \rho_{im}\left(\frac{\sigma_i}{\sigma_m}\right)$, **which** needs calculation of asset/market correlation, market standard deviation, and asset standard deviation.

Note that constructors get one or two asset objects.

Similarly, `Signal` objects are defined on `Asset` or `Statistic` objects. They work as filters and generate values only **when** their base condition holds.

You need to implement at least 4 **subclasses** of the `Signal` class as **follows**:

- `Threshold(a, selval, buyval)` **Is triggered** when the **price** crosses the given threshold. For example, the previous value is less than the threshold value and the new value is greater than or **equal** to the threshold.
- `Cross(a1, a2)` Is triggered when the prices from two assets **cross each other**.
- `RSI(a, t)` **Relative** Strength Index is based on the average gain and average loss of the prices over a time period.
- `BollingerBand(a,t)` Is triggered when the price gets out of the standard deviation and moving average bands.

Each `Statistic` and `Signal` **type object is** observable, just like assets. The system should provide a factory that allows arbitrary composition of assets, statistics, **and** signals with their names as calls like:

```
createSignal("cross", createStat("average", asset("gold"), 100),
                      createStat("average", asset("gold"), 10))
```

For example, a signal **showing when the** 100-second average of "Gold" crosses **its** 10-second average can be constructed.

4. Cargo Delivery and Tracking System

CargoItem class

`CargoItem` class has the following methods:

Method	Description
`constructor(sendernam, recipnam, recipaddr, owner)`	Create a new cargo item. The item is assigned a unique tracking id; **the** state is set to `accepted`.
`trackingId()`	**Returns the** tracking id of the item.

Method	Description
`getContainer()`	**Returns** the id of the container that the item is inside. If not loaded in a container yet, returns `None`.
`setContainer(container)`	**Updates the** container of the item.
`updated()`	Called when the status of the item changes: new container is set, the container updates its position, or the state changes.
`complete()`	Called once the cargo is delivered to the recipient; the state is set to `complete`.
`track(tracker)`	**Adds** the tracker object to the tracker list so that updates of the item are sent to the tracker.
`untrack(tracker)`	**Removes** the tracker object from the tracker list.

Container class

A `Container` class can be stationary, like a cargo office or a hub (airport, depot, etc.). You can derive different behaviors as subclasses. `FrontOffice` and `Hub` subclasses can serve as stationary containers; when an item is inserted, its state will be `waiting`. Otherwise, the state will be `in transit`.

Method	Description
`constructor(cid, description, type, loc)`	Create an empty container.
`setlocation(long, latt)`	**Sets the** new location of the container.
`getState()`	Cargo items contained in this container call this function to get the state as `waiting` or `in transit`.
`move(itemlist, newcontainer)`	The items in the list are dropped from the current object and inserted into the new container.
`load(itemlist)`	The items in the list are inserted into the current object.
`unload(itemlist)`	The items in the list are removed from the current object.
`track(tracker)`	**Adds** the tracker object to the tracker list so that updates in the container are sent to the tracker.
`untrack(tracker)`	**Removes** the tracker object from the tracker list.

Tracker class

The `Tracker` class is used to track a set of `CargoItem`s. A user can construct and modify a tracker object in order to be notified **of** state changes. The tracker

may track cargo items as well as containers.

Method	Description
`constructor(tid, description, owner)`	Create a new tracker with no tracking items or containers.
`addItem(itemlist)`	**Adds** a list of cargo items to track.
`addContainer(contlist)`	**Adds** a list of containers to track.
`updated()`	Is called by tracked objects to inform that the model has been changed.
`getStatlist()`	**Returns** a list of states for the tracked items, including their locations.
`setView(top, left, bottom, right)`	The tracker generates reports for all tracked objects by default. This call restricts the report to the rectangle defined by the parameters. Status changes of objects out of this rectangle are ignored. Also, `getStatList` returns only objects with a location contained in the rectangle.

Unit Testing

You will implement unit testing scripts in addition to the class library. Tests will cover all method calls you implement. You will show the result and implementation of unit tests during the demo. You will use `pytest` module for unit tests. More information will be provided in the class.

Policies

Any kind of code sharing among the teams is not allowed. You will get zero from the phase involving the cheating. During the demo, both team members will be asked detailed questions about the code. If you fail to answer the questions, we will assume the code is not your work and cut your points. You have to show that you have the full control of your code.

Submission

You will submit all your files in an archive file on odtuclass. The demonstrations will be scheduled in the following days where you need to demonstrate the submitted files, no further contribution allowed.