prof-welch-psu-ist / asg1-KathrynSimpson Private

generated from dtwelch/racetrack

asg1-KathrynSimpson created by GitHub Classroom

0 stars 0 forks

Star Watch

Code Issues Pull requests Actions Projects Security Insights

main

github-classroom Initial commit ... 11 days ago 1

View code

README.md

Racetrack: A Cli-Based Competitive Running App

In this assignment you will implement a small Command Line Interface (CLI) based Java application. In doing so, you will practice reading and interpreting partial requirements documentation, working with basic Java objects and collections, as well as begin to familiarize yourself with the version control system git and its associated workflow.

The Problem

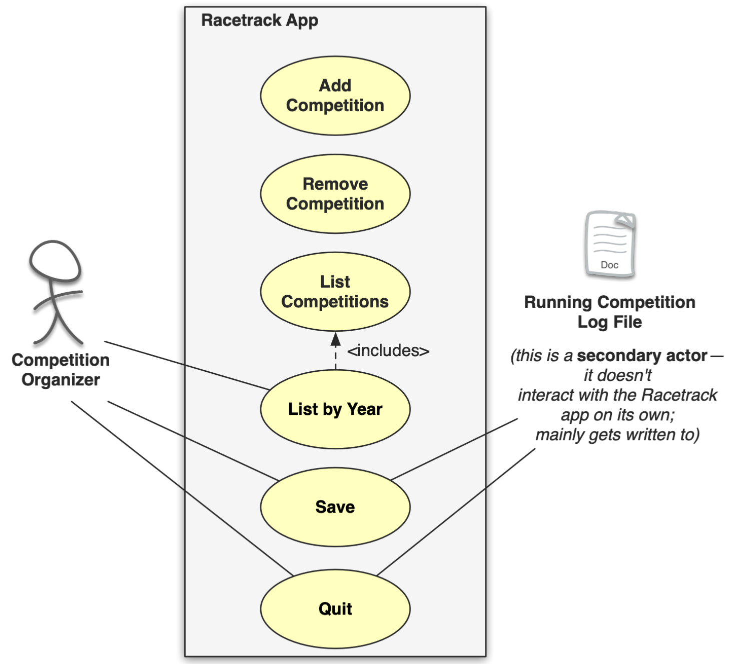
Your small development outfit has been approached by the organizer of a local runners club with a request for an application that can help track, view, and update information associated with various competitive running events the club has participated in over the years.

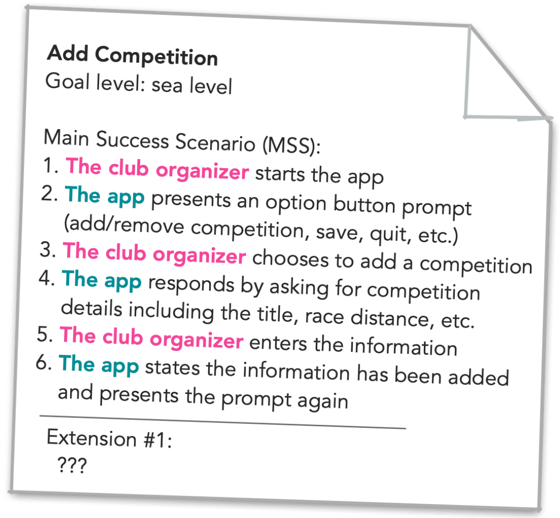
The club's mechanism for storing physical records of races has become increasingly unwieldy. The club organizer has asked your outfit to to build a specialized application that allows users to query past competitions, add/remove competitions, list all the competitions and their details out (or list by a specified year), save (persistence), and be able to quickly observe the number of runners/participants in a given event along with the top three runners and their finishing times.

*The app should be user friendly and sensitive to the users time and effort (e.g., asking to save before closing **if** the app detects changes, additions, removals of competitions, etc.)*.

Part 1: Use Cases

Your (fictional) partner has started specifying the the functional requirements for the app (tentatively titled: **Racetrack**), and has even provided some rough UML class digrams to help pin down a tentative design. Your partner's use case diagram and (partially developed) textual counterparts for the identified use cases below:


Racetrack App
Add Competition
Remove Competition
List Competitions
<includes>
List by Year
Save
Quit
Competition Organizer
Doc
Running Competition Log File
(this is a **secondary actor** — it doesn't interact with the Racetrack app on its own; mainly gets written to)



Add Competition
Goal level: sea level

Main Success Scenario (MSS):
1. The club organizer starts the app
2. The app presents an option button prompt (add/remove competition, save, quit, etc.)
3. The club organizer chooses to add a competition
4. The app responds by asking for competition details including the title, race distance, etc.
5. The club organizer enters the information
6. The app states the information has been added and presents the prompt again

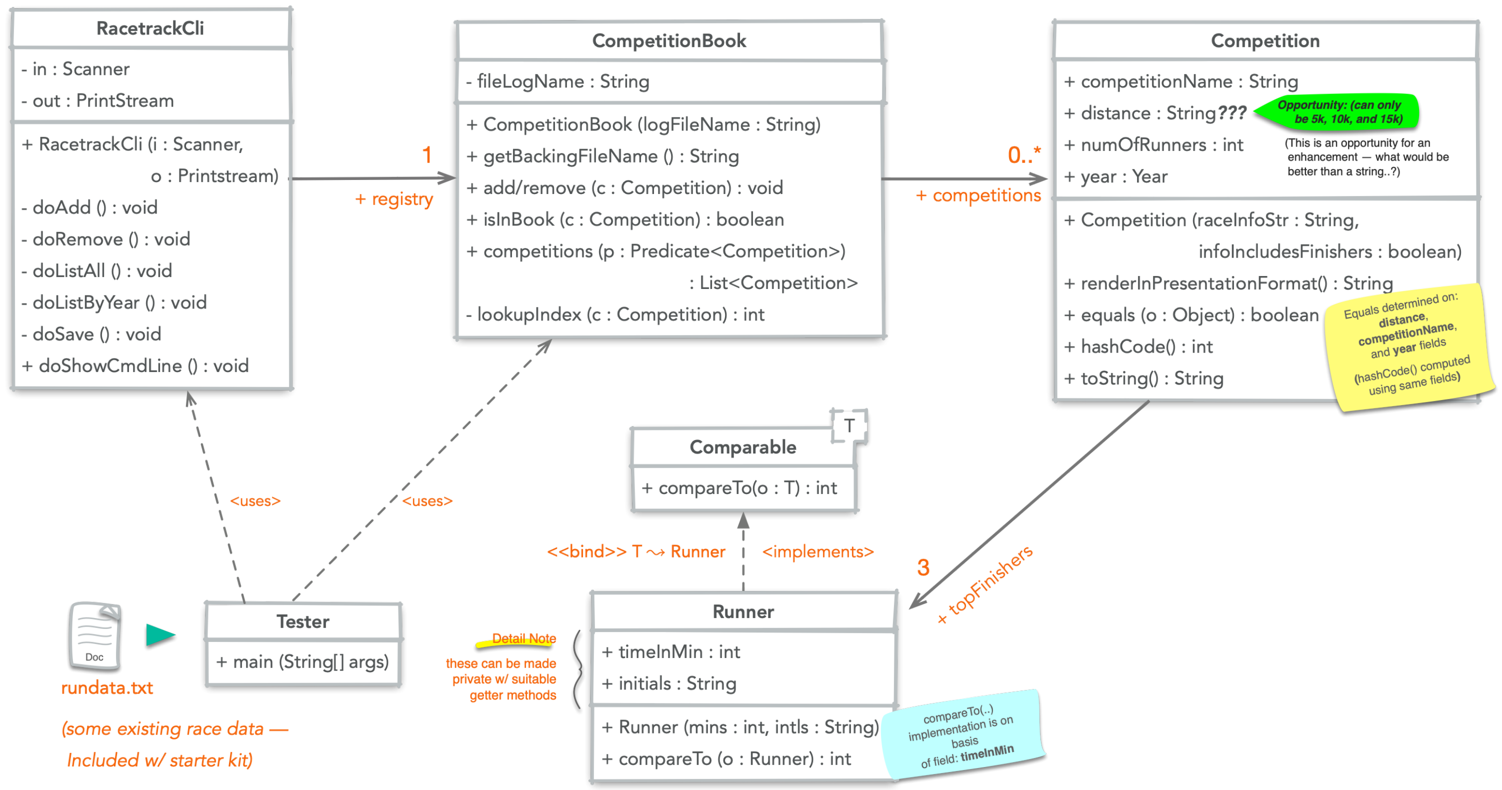
Extension #1:
???

As Fowler suggests in UML Distilled, think of the use case diagram on the left as a "graphical table of contents" for all the apps use cases.

You must finish your partner's work by supplying the rememaining textual descriptions and variant scenarios for each use case depicted (**Add Competition** has been done for you, but not its extension). All the use cases should be described using Fowler's textual notation in a text `.txt` or markdown `.md` file. If you wish to do it in github flavored markdown (gfm), consult Github's mastering markdown overview. This is worth learning, as it can then be used to render nice looking documentation for your own projects.

Part 2: Prototype Implementation of a (Tentative) Design

The following is a class diagram your parter devised.


RacetrackCli
- in : Scanner
- out : PrintStream
+ RacetrackCli (i : Scanner, o : Printstream)
- doAdd () : void
- doRemove () : void
- doListAll () : void
- doListByYear () : void
- doSave () : void
+ doShowCmdLine () : void
1
+ registry
CompetitionBook
- fileLogName : String
+ CompetitionBook (logFileName : String)
+ getBackingFileName () : String
+ add/remove (c : Competition) : void
+ isInBook (c : Competition) : boolean
+ competitions (p : Predicate<Competition>) : List<Competition>
- lookupIndex (c : Competition) : int
0..*
+ competitions
Competition
+ competitionName : String
+ distance : String???
Opportunity: (can only be 5k, 10k, and 15k)
+ numOfRunners : int
(This is an opportunity for an enhancement — what would be better than a string..?)
+ year : Year
+ Competition (raceInfoStr : String, infoIncludesFinishers : boolean)
+ renderInPresentationFormat() : String
+ equals (o : Object) : boolean
+ hashCode() : int
+ toString() : String
Equals determined on: distance, competitionName, and year fields (hashCode() computed using same fields)
<uses>
<uses>
T
Comparable
+ compareTo(o : T) : int
<<bind>> T ⤳ Runner
<implements>
3
+ topFinishers
Doc
rundata.txt
(some existing race data — Included w/ starter kit)
Tester
+ main (String[] args)
Detail Note
these can be made private w/ suitable getter methods
Runner
+ timeInMin : int
+ initials : String
+ Runner (mins : int, intls : String)
+ compareTo (o : Runner) : int
compareTo(..) implementation is on basis of field: timeInMin

You must implement a working prototype of this system. Your partner, as shown in the diagram, has left a few notes on the design of the various classes that make up the system. Here are more details on the various responsibilities of each individual class:

`RacetrackCli`

This class uses the `doShowCmdLine()` method to repeatedly present users (via a 'busy' while loop) with a selection of options that correspond to the various use cases of the system:

> Welcome to the running competition record system
>
> Enter A)dd, R)emove, LY) List by Year, S)ave, L)ist all, or Q)uit

Note that the class holds references to `Scanner` and `PrintStream` objects, which are used to input user command strings and output messages from the system, respectively. So don't write `System.out.println(..)` anywere, but rather: call `out.println(..)` . The main intent of this class is to serve as an intermediary between the user, IO, and the business logic found in the other classes. Thus, it de-couples the output and input streams from the logic inside the `CompetitionBook` class.

The high level structure of your busy loop might look something like the following in Java:

```java
public void doShowCmdLine() {
   out.println("Welcome to the running competition record system");
   out.println("------------------------------------------------")
   while (true) {
      out.println("Enter A)dd, R)emove, LY) List by Year, S)ave, L)ist all, or Q)uit")
      try {
        String cmd = input.nextLine().trim().toUpperCase();

        if (cmd.equals("A")) { doAdd(); }
        /* additional actions here */
        else { out.println("Invalid command"); }

      } catch (Exception e) {
         out.println("An error occurred ..  " + e.getMessage() + " .. try again");
      }
   }
}
```

`CompetitionBook`

This class acts as a repository for storing all existing `Competition` objects. The data structure you use to store these is up to you, though you should think about which data structure might be leveraged to make searching the stored collection of competitions faster/more scalable. That said, since we're just prototyping this app, feel free to use whichever would be the most straightforward.

The constructor for this class should accept a file name of existing races, load the file, and pass each line of text read to the `Competition` class constructor for parsing. After the constructor for `Competition` completes, the resulting object should be stored in the `competitions` collection. Handle all exceptions that arise while parsing data in the constructor.

`Competition`

This class represents an individual race/competition. The constructor should accept a string of the following form for describing a competition:

> [comp. name] [5k|10k|15k] ([year]) [number-of-runners] [top 3 finishers (initials:minutes)]

A concrete example matching this schema can be found in the `rundata.txt` file that your partner has put together for testing purposes:

> Sparrow Pond Trail Run 5k (2012) 12 (h.o.s:26) (p.k.t:30) (p.c.v:49)

The constructor for class `Competition` will need to parse this string into the following fields:

- the name of the competition as a string (i.e.: "Sparrow Pond Trail Run")
- the distance (5k, 10k, or 15k) -- I'll leave how you model this type-wise up to you
- the year (utilize `java.time.Year` for this)
- an integer holding the total number of runners in the competition
- the top three `Runner` s (note that they might not always be given in order in terms of finish time); these should be stored in the `topFinishers` list within the `Competition` class

I'll leave the parsing to you, though I suggest you consult API docs and examples of `java.util.StringTokenizer` for ideas on how to break up the string up into individual tokens that can then be converted and stored into the appropriate fields of the class.

Note that the constructor also accepts a boolean flag indicating whether or not the `raceInfoStr` fed into the constructor contains information about the top runners/finishers. *Whether this flag is true or not will affect how the string is parsed. This is mainly to ease removal of `Competition` objects (so the users can just type the name, distance, and year of the race they want to delete).*

Accordingly, you should override the `equals(..)` method for this class to consider two competition objects equal if and only if their name, year, and distances (5k, 10k, 15k) match exactly; false should be returned otherwise.

The `renderInLogfileFormat()` method merely produces a string suitable for writing new entries to the `rundata.txt` log file that comes with the project. In other words, it should render output that can be written to that file and reparsed when necessary (i.e.: when/if the app is restarted).

Here's some code for the `toString()` method that will help render the entries of the logfile in a nicer format (e.g., when the races are listed via the cli):

```
@Override
public String toString() {
    String result = competitionName + " " + distance.asString + " (" + year + ") ";

    if (!topFinishers.isEmpty()) {
        result = result + "runners: " + numberOfRunners + " top time: " +
                    topFinishers.get(0).getTimeInMinutes() + "mins";
    }
    return result;
}
```

Runner

The runner class should implement the `Comparable<Runner>` interface to allow sorting of top finishers in the `Competition` class constructor (using the `using the` Collections.sort(..) `method). Note that you'll need to override the` compareTo(..)` method and give it the following logic:

```
@Override
public int compareTo(Runner o) {
   return this.timeInMins - o.timeInMins
}
```

You should also override `toString()` to render instances of the class like so:

> (initials:timeInMins)

Imagined (Example) Execution

Here's a hypothetical sample run of the application:

```
Welcome to the running competition record system
-------------------------------------------------
Enter A)dd, R)emove, LY) List by Year, S)ave, L)ist all, or Q)uit
A
Enter: [comp. name] [5k|10k|15k] ([year]) [number-of-runners] [top 3 finishers (initials:minutes)]
Example Run 5k (2008) 205 (a.k.s:30) (h.k:31) (j.k:32)
Added competition Example Run 5k (2008) runners: 205 top time: 30mins

Enter A)dd, R)emove, LY) List by Year, S)ave, L)ist all, or Q)uit
X
Invalid command

Enter A)dd, R)emove, LY) List by Year, S)ave, L)ist all, or Q)uit
LY
Enter the year: 2008

Competitions held in 2008:
        St. Patricks Distance Festival 10k (2008) runners: 10 top time: 49mins
        Albion Fair 5k (2008) runners: 205 top time: 36mins
        4 Seasons Mud Run 10k (2008) runners: 57 top time: 58mins
        Example Run 5k (2008) runners: 205 top time: 30mins
```

```
Enter A)dd, R)emove, LY) List by Year, S)ave, L)ist all, or Q)uit
Q
Would you like to save (Y/N):
Y
Saved!
Goodbye!
```

Handin and Grading

When you are ready to submit, commit your work by typing:

> git commit -am "message goes here"

then follow this up with a

> git push origin main

You will be graded on the quality of your requirements analysis (use of fowler notation) and the quality and clarity of your implementation (including comments).

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

- ● **Java** 100.0%