HW2 –Modularity and Encapsulation

Estimated time: 16-20 hours

Objectives

- Practice in designing software with good modularity and encapsulation
- Become familiar with using the Strategy and Observer patterns
- Become more familiar with UML class and interaction diagrams
- Improve unit testing skills

Overview

In this assignment, you will build the *Tracking Server* for simply road-race tracking system that will help race officials, support teams, and spectators monitor where athletes currently are on a race course. In the building this system, you will look for and take advantage of opportunities to apply the Strategy and Observer patterns.

Description

The challenge in watching a road race is there is no practical way for people to see the whole course all the time. So, we need some kind of tracking system that can help spectators, support personnel, and officials view athlete's status and position.

Two key types of objects in the system are race events and athletes. A race event has a title and course length. In the real-world, there is a lot more information about race events and the courses on which they are held, but this minimal information is sufficient for our simply tracking application. For example, we don't need to know about the actual course route because we can consider every course to be a 1-dimension line of the specified length (in meters). Each athlete has a unique number, called bib number, as well as a first name, last name, gender, and age.

For the tracking system to working, there needs to be a data source that supplies information about the racers in real time. In a real-world system, this would be another system with position sensors and transmitters. We will use a *Simulator* that you can run as a separate program and that will send simulated data to your *Tracking Server*. The *Simulator* will read an input file to simulate a real road-race, but potentially much faster than the actual event would take in real life. You will be provided with a functioning *Simulator* that you can run on a Window platform directly or on Linux and MacOS using Mono.

Clients (spectators, support personnel, and officials) will subscribe with the server to receive tracking information about specific athletes and will display information about athletes and their status. You will be provided with a functioning *Client* that can be run on Windows or with Mono.

You will also be provided with the starting source code (in C# and Java) for a server-side class, called *Communicator*, that can receive messages from the *Simulator* or *Client* and send messages to a *Client*. Your *Tracking Server* should create an instance of this class. It can be used as either an "active object" or "passive object".

If you use *Communicator* as an active object, create instance and call the *Start* method. This will cause the Run method to execute on its own thread. The Run method will check for incoming messages and delegate the processing of the method to some functionality that you provide. If you are working in C# you must register a delegate with the *IncomingMessage* event of the *Communicator* class. If you are working in Java you must create a class that implements *IMessageProcessor*, create an instance of that classes, and give that instance to the *Communicator* using the *setProcessor* method.

If you use Communicator as a passive object, your *Tracking Server* must contain the logic that repeated tries to get a message from the Communicator. It can call *GetMessage* to try to retrieve the message within a certain amount of time. With the C# version of the *Communicator*, it can also call *IsMessageAvailable* to check to see if there is a message to receive.

You are welcome to change the *Communicator* class but shouldn't have to do so to build a well-designed *Tracking Server*.

Your *Tracking Server* will include other classes for manage data about races and athletes. As you design the system, look for opportunities to use the Strategy and Observer patterns effectively. For example, when a *Client* first communicates with the server, consider creating an object within the server that represents that *Client* and its communication end point, and then let this object become a subscriber to athlete objects.

Each message received by or sent to the *Tracking Server* is a comma-delimiter Unicode string, where the commas separate various field values. The value of the first field indicates the type of message. The rest of the fields vary by message type. The *Tracking Server* can receive seven different kinds of messages from the *Simulator*; it can receive three different kinds of requests from client, and it can send three different kinds of messages to a *Client*. See Tables 1-2 below.

Table 1 – Types of messages sent from the *Simulator* to the *Tracking Server*.

Message Type	Description	Message Format
Race Started	Sent from at the beginning of a race	Race,<race name>,<course length in meters>
Registered Update	Sent when a new athlete registers for the race. All of these kinds of updates will come before first start update. For each of these kinds of updates, your system needs to create an athlete.	Registered,<bib number>,<time>,<first name>, <last name>,<gender>,<age>
Did-Not-Start Update	Sent when an athlete cannot start the race for some reason.	DidNotStart,<bib number>,<time>
Started Update	Sent when an athlete official start. In large races, athletes start in small groups.	Started,<bib number>,<time>
On Course	Sent periodically to identify the position of an	OnCourse,<bib number>,<time>,<distance covered in

Update	athlete on the course.	meters>
Did-not-Finish Update	Sent when an athlete has to drop out of the race for some reason.	DidNotFinish,<bib number>,<time>
Finished Update	Sent when an athlete courses the finish line. The different between athlete's finish time and start time is that athlete's race time.	Finished,<bib number>,<time>

Table 2 – Types of messages sent from the *Client* to the *Tracking Server*.

Message Type	Description	Message Format
Hello	Sent whenever a client startup	Hello
Subscribe	Sent when client wants to start tracking an athlete	Subscribe,<bib number>
Unsubscribe	Sent when client wants to stop tracking an athlete	Unsubscribe,<bib number>

Table 3 – Types of messages sent from the server to the *Client*.

Message Type	Description	Message Format
Race Started	Sent from the server to the client after to client subscribes to an athlete for the first time	Race,<race name>,<course length in meters>
New Athlete	Sent from the server to every client whenever a new athlete is registered.	Athlete,<bib number>,<first name>,<last name>,<gender>,<age>
Athlete Status	Sent from the server to the client anytime the athlete status changes and the client is tracking that athlete.	Status,<bib number>,<status>,<start time>,<distance covered in meters>,<last updated time>, <finished time>

Note all times are integers that represent seconds relative to the beginning of the race and all distances are doubles that represent meters.

Requirements

Build a *Tracking Server* for a road-race tracking system that satisfies the following functional requirements:

1. Allows a user start up the *Tracking Server* from the command-line.
2. After starting *Tracking Server*, the user should be able to start up a *Simulator* and any number of *Client* processes. To do this easily, you may want to have the *Tracking Server* display its communication end points. You may also want to have your *Communicator* always use the same port number, e.g., 12000.
3. An athlete's state should include bib number, first name, last name, gender, age, status, distance covered, start time, last-updated time, finish time.

4. When the *Tracking Server* receives a *Race Started* messages, it needs to initialize all race and athlete data and send *Race Started* messages to all known clients.
5. When the *Tracking Server* receives a *Registered Update* message, it must create a new athlete using the racer information in the message and send a *New Athlete* message for that athlete to all clients.
6. When the *Tracking Server* receives a *Did-Not-Start Update* message, it should update the specified athlete accordingly by changing the athlete's status to "did not start" and time. This state change should cause an *Athlete Status* message to be sent to all clients that are tracking the athlete.
7. When the *Tracking Server* receives an *On-Course Update* message, it should update the specified athlete accordingly by distance covered and time. This state change should cause an *Athlete Status* message to be sent to all clients that are tracking the athlete.
8. When the *Tracking Server* receives a *Did-Not-Finish Update* message, it should update the specified athlete accordingly by changing the athlete's status to "did not finish" and time. This state change should cause an *Athlete Status* message to be sent to all clients that are tracking the athlete.
9. When the *Tracking Server* receives a *Finished Update* message, it should update the specified athlete accordingly by changing the athlete's time and the status to "Finished". This state change should cause an *Athlete Status* message to be sent to all clients that are tracking the athlete.
10. When the *Tracking Server* receives a *Hello* message from a *Client*, it should create an object that represents that client and the communication end point that client is using. Besides the communication end point, the client object may need to encapsulate some information about what's been sent to the client, like whether race information about been sent already.
11. The first message a *Tracking Server* should send a *Client* is a *Race Started* message.
    a. If the Tracking Server receives a *Hello* message from some *Client X* after it has received a *Race Started* messages from the Simulator, then the Tracking Server should send *X* a Race Started messages immediately, along with *New Athlete* messages for all known athletes to that point.
    b. If the *Tracking Server* receives a *Hello* message from some *Client* prior to receiving a *Race Started* message for the *Simulator*, it needs to postpone the sending the *Race Started* message until it receives one from the *Simulator*.
12. After the *Tracking Server* receives a *Subscribe* message from a client, it must send an *Athlete Status* message back to the client anytime the state of the athlete changes.
13. After the *Tracking Server* receives an *Unsubscribe* message from a client, it should stop sending *Athlete Status* message for the specified athlete to that client.
14. The user should be able to stop the *Tracking Server* in a simple way.

Instructions

1. Design *Tracking Server*
    1.1. Your designs should include appropriate UML class and interaction diagrams.

2. Implement all the component for your Tracking Server with appropriate unit-test cases
3. Complete some ad hoc integration and ad hoc system testing using the *Simulator* and *Client* programs that are provided to you.
4. Zip up your entire solution, including test cases and sample input files, in an archive file called CS5700_hw2_<*fullname*>.zip, where fullname is your first and last names. Then, submit the zip file to the Canvas system.
5. Complete a code review.

Grading Criteria

Criteria	Max Points
A clear and concise design documented with UML class and interaction diagrams	20
A working implementation, with good abstraction modularity, encapsulation	25
Effective use of the Strategy pattern	20
Effective use of the Observer pattern	20
Reasonable unit-test cases	25
Reasonable integration and system testing using the Simulator and Client	10