HW2 – Decoupling with Observers and Extending with Decorators

Estimated time: 12-16 hours

Objectives

- Become more familiar with UML class and interaction diagrams
- Become familiar with the Observer pattern
- Become familiar with the Decorator pattern
- Improve unit testing skills
- Broaden and strengthen programming skills in related areas, including user interfaces

Overview

In this assignment, you will build a simply road-race tracking system for runners or cyclists 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 observer and decorator patterns.

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

Three key objects that need to be implemented in the race monitoring system are race events, courses, and athletes. Race events have titles and are intended to start on a particular data and time. See Figure 1. Races are run on courses, which have names and total distances. The actual route of the course is not relevant for our monitor application, because we will consider a course a 1-dimension line of some specific distance. Athletes have bib numbers that uniquely identify them and well as a registration profile, which includes first names, last name, gender, and age.

For the monitor system to working, there needs to be a data source that supplies information about the racers in real time. We will use an active-object simulator, called the *SimulatedDataSource*, that will read race data from a file and feed it you monitor system in simulated real-time. See Figure 2 and 3. As far as your program is concerned, it simply needs to create a *SimulatedDataSource*, setup up its input file name, and data handler, and then start it. At that point, the simulation will call the data handler with an *AthleteUpdate* object (a message) to create or change athlete data. As your program processes these messages, the athletes object will become available as subjects and change state, causing any observer subscribed to them to be updated. Some of this observers may be UI components, but they don't all have to be be.

Your job will be to design and build the application logic for handling all of the updates as they come from the *SimulatedDataSource* and the graphical user interface. See Figure 4.

The simulator can send your program six different kinds of updates (see Figure 1):

Registration 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.
Did-not-start Update	Sent when an athlete cannot start the race for some reason.
Started Update	Sent when an athlete official start. In large races, athletes start in small groups.
Location Update	Sent periodically to identify the position of an athlete on the course.
Did-not-finish Update	Sent when an athlete has to drop out of the race for some reason.
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.

Requirements

Build a system that allows a user to start up a race simulation for a race event on a course, and then monitor the activity of the race through a variety of observers.

When starting up your program, allow the user to specify the race course (name and total distance) and an race event (title and start date/time). You can use this information to help make you displays more meaningful (informative).

Then, allow the user to start *SimulatedDataSource*. Your program will start receiving update messages from the simulator within 1 second after starting. The first setup of update messages you will receive will be *RegistrationUpdate* objects, with maybe some *DidNotStartUpdate* objects mixed in. After all the cyclists have been registered, there will be a pause in messages. After about 100 seconds, data source will begin sending *StartedUpdates*, *LocationUpdates*, *DidNotFinishUpdated*, and *FinishedUpdates*. Note that the athletes may not all start at the same time.

Any time after starting a simulation, your program needs to allow the user to create any number of observers, of at least four different types. Below are some ideas for observers. Be creative and try to make your observers useful from a race monitoring perspective. Also, when the user creates a new observer, your program should give the user some useful tools for subscribing that observer to athletes and for configuring as needed.

Email Observer	Send an email to some address every X^{th} update about its subject location.

List Observer	Displays the current status of its subjects in a simple ordered list. This kind of observer should be able to register to multiple subjects.
Graphical Position Observer	Illustrates the course as a 1D line and places markers along the course corresponding each of its subjects. This kind of observer should be able to register to multiple subjects.
Milestone Times Observer	For a given milestone, display fastest time and the next *n* fastest time relative to that time. Note that this kind of observer may make require introduce a new subject and applying the observer pattern twice, where the new subject is also observer.
Athlete Comparison Observer	Display relative the position and time between any two athletes.

Your system needs to include at least one class that has dynamically extensible behavior. For example, one of your observers could have a number of add-on features can be dynamically added or removed.

Design and Implementation Suggestions

- Obviously, the Athlete class needs to be a subject. However, depending on the kinds of observers that you want to create, you may need to add other kinds of subjects. Note that an observer can also be a subject for another kind of observer.
- Look for opportunities to use the strategy pattern in implementing the data processing.
- Look for opportunities to make the functionality of certain classes extensible through the Decorator pattern
- Although, it should be necessary, feel free to make minor modifications to the simulator code

Instructions

To build this system, you will need to do the following:

1. Design, implement, and test the monitoring application and GUI functionality
2. Capture your designs in appropriate UML class and interaction diagrams. You may adapt the diagrams shown in Figures 1-4, but your design should add more detail for the application and GUI layers.
3. Test your core application software at the unit level with meaningful and relatively using executable test cases. You do not need to test the GUI code with executable test cases.
4. Test your software at the system level using the simulation data provide by the instructor and ad hoc testing techinques

Submission Instructions

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.

Grading Criteria

Criteria	Max Points
A clear and concise designs consisting of UML class and interaction diagrams	20
A working implement, with good encapsulation, abstractions, inheritance, and aggregation, and appropriate use of the observer, decorator, and other patterns	30
Meaningful observers that would help people track a race	20
Meaningful, executable unit test cases	30
Reasonable systems testing using the simulation data	20

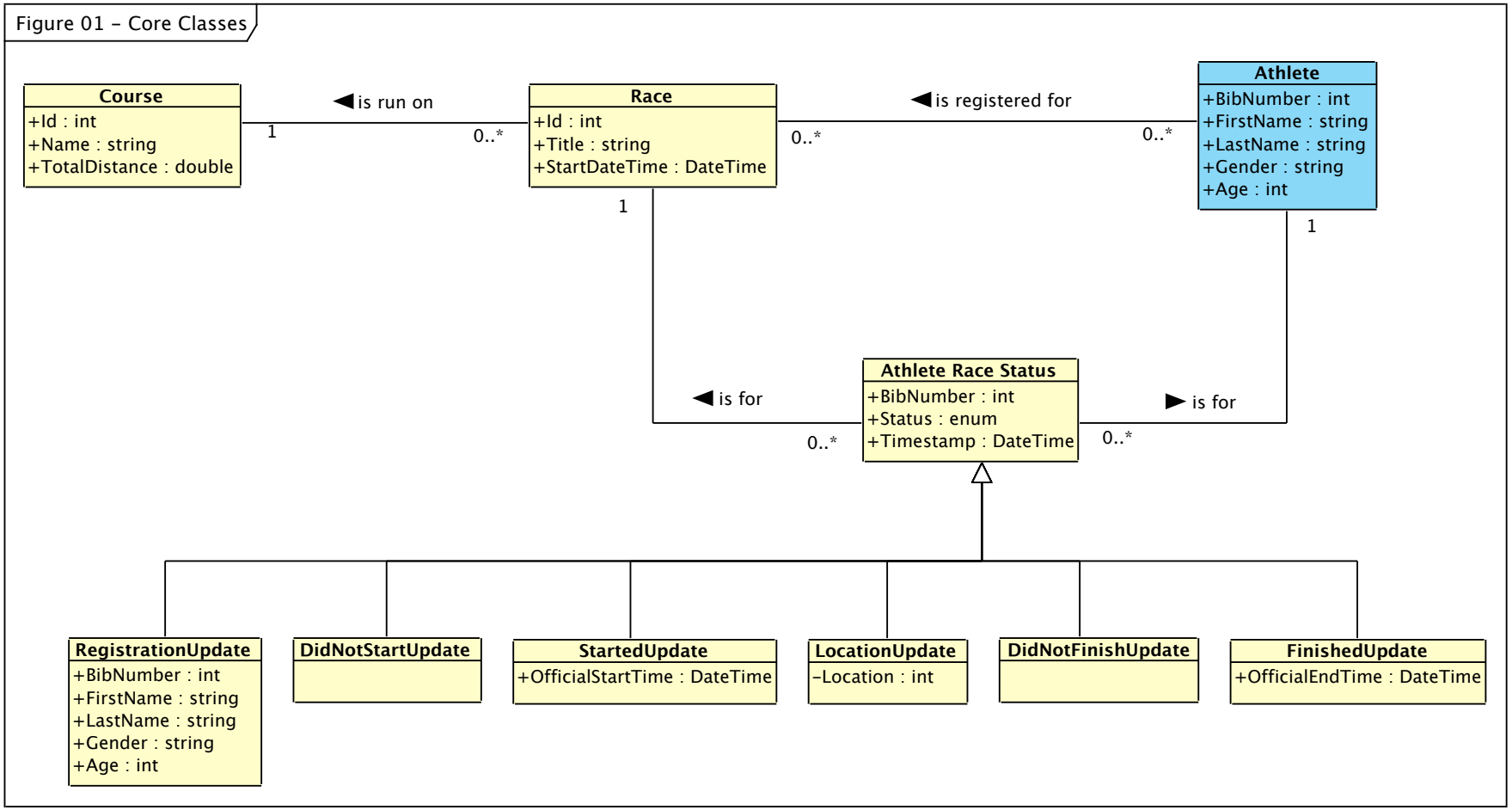


Figure 01 – Core Classes
Course
+Id : int
+Name : string
+TotalDistance : double
is run on
1
0..*
Race
+Id : int
+Title : string
+StartDateTime : DateTime
is registered for
0..*
0..*
Athlete
+BibNumber : int
+FirstName : string
+LastName : string
+Gender : string
+Age : int
1
1
is for
0..*
Athlete Race Status
+BibNumber : int
+Status : enum
+Timestamp : DateTime
is for
0..*
RegistrationUpdate
+BibNumber : int
+FirstName : string
+LastName : string
+Gender : string
+Age : int
DidNotStartUpdate
StartedUpdate
+OfficialStartTime : DateTime
LocationUpdate
-Location : int
DidNotFinishUpdate
FinishedUpdate
+OfficialEndTime : DateTime

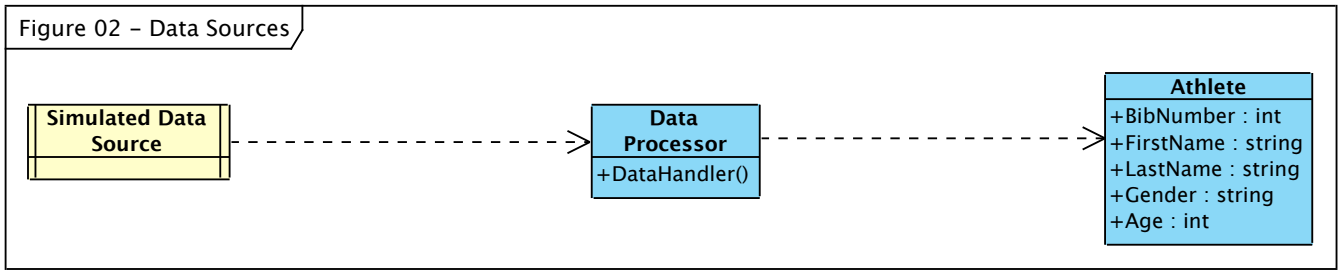
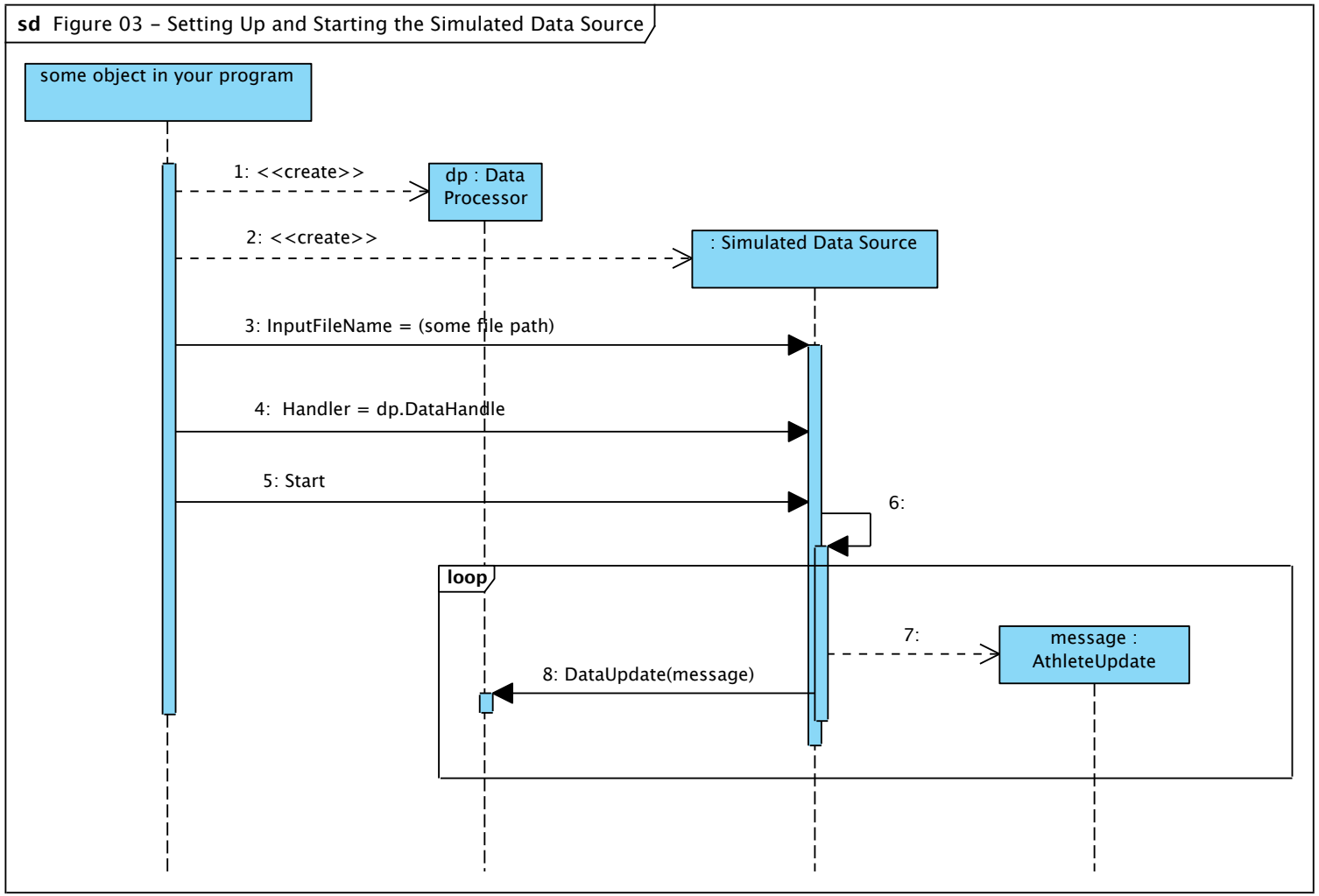


Figure 02 - Data Sources
Simulated Data Source
Data Processor
+DataHandler()
Athlete
+BibNumber : int
+FirstName : string
+LastName : string
+Gender : string
+Age : int


sd Figure 03 – Setting Up and Starting the Simulated Data Source
some object in your program
1: <<create>>
dp : Data Processor
2: <<create>>
: Simulated Data Source
3: InputFileName = (some file path)
4: Handler = dp.DataHandle
5: Start
6:
loop
7:
message : AthleteUpdate
8: DataUpdate(message)

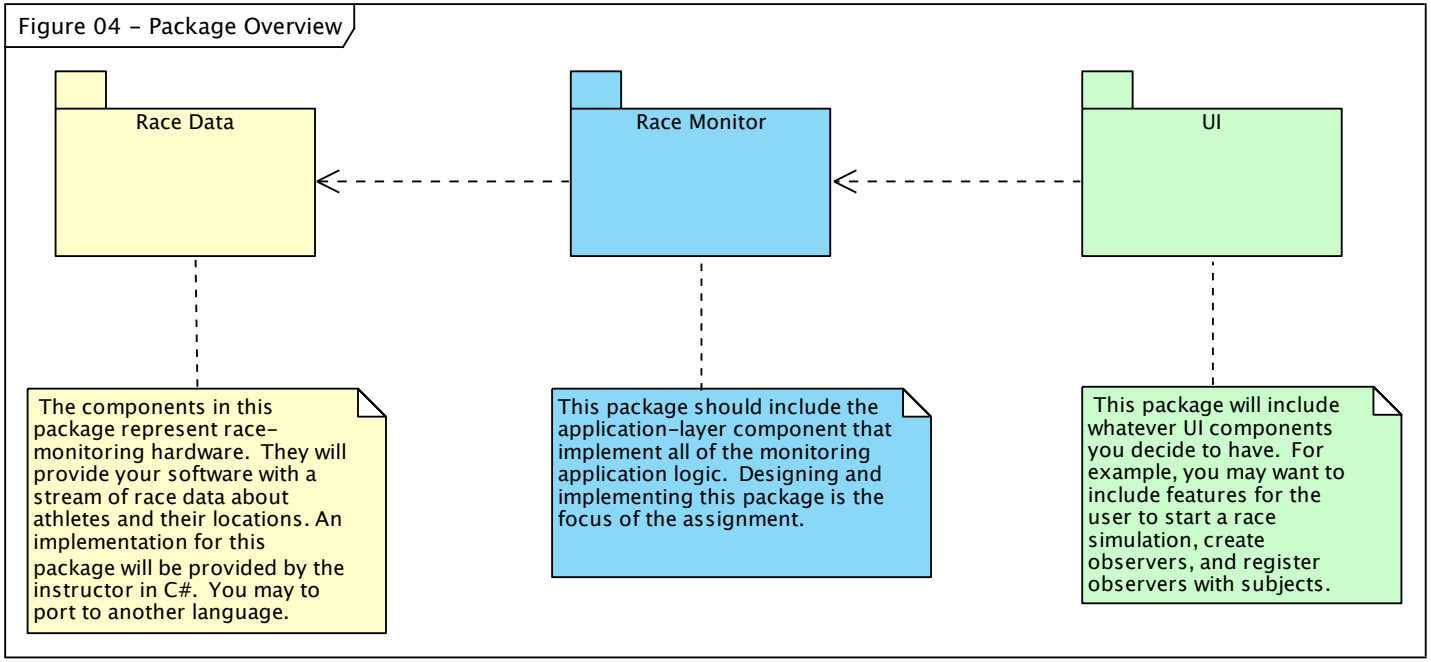


Figure 04 - Package Overview
Race Data
Race Monitor
UI
The components in this package represent race-monitoring hardware. They will provide your software with a stream of race data about athletes and their locations. An implementation for this package will be provided by the instructor in C#. You may to port to another language.
This package should include the application-layer component that implement all of the monitoring application logic. Designing and implementing this package is the focus of the assignment.
This package will include whatever UI components you decide to have. For example, you may want to include features for the user to start a race simulation, create observers, and register observers with subjects.