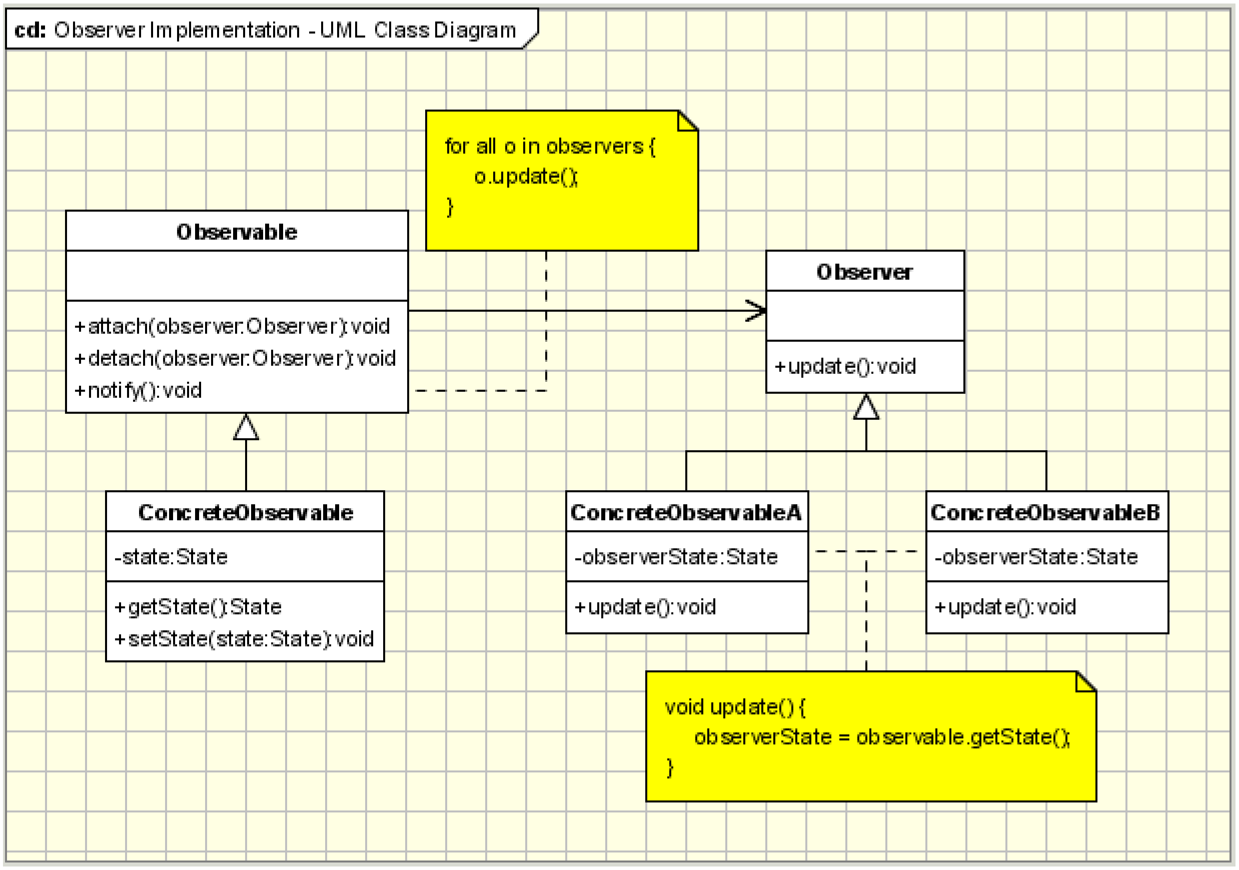
The Observer Pattern
Design Patterns
Andreya Grzegorzewski – Ohio Northern University
Fall 2016

Introduction

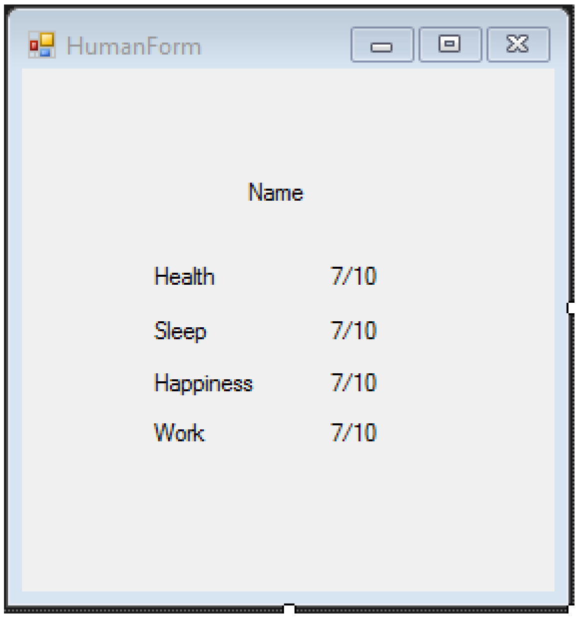
This assignment requires an application implementing the observer pattern as described in class. Instead of following the UML diagram directly, the app uses the EventArgs class and subscribes to events that I defined. My app is called “Human Simulator,” and it simulates the life of a person in a very basic manner. In the primary form, the user can input how many hours per day his human should spend on health, sleep, happiness, and work, as well as his human’s name. With these inputs, a new form is created describing the human. A timer is used to continuously update the human’s health, sleep, work, and happiness levels, which are displayed in the form of x/10. If x falls below zero at any point for any of those levels, the human dies, which fires an event. This is where the modified observer pattern is implemented. The event sends information about the human’s death to the primary form, where the death is displayed in a list box.

UML Diagram

The UML diagram for the observer pattern, shown on the next page courtesy of oodesign.com, shows the observer pattern as described by the Gang of Four. Although my app does not follow this diagram exactly, there are some similarities between the two. The primary form in my application, called Form1, could represent the Observer class shown in the diagram. Likewise, my secondary form, HumanForm, could represent the Observable class. This is because the HumanForm has its own events and event handlers, and Form1 listens for those events. Normally, the observer pattern would implement attach, detach, notify, update, getState, and setState methods, as specified in the UML diagram. However, the application I submitted captures the spirit of the pattern and properly implements the modified pattern instead.

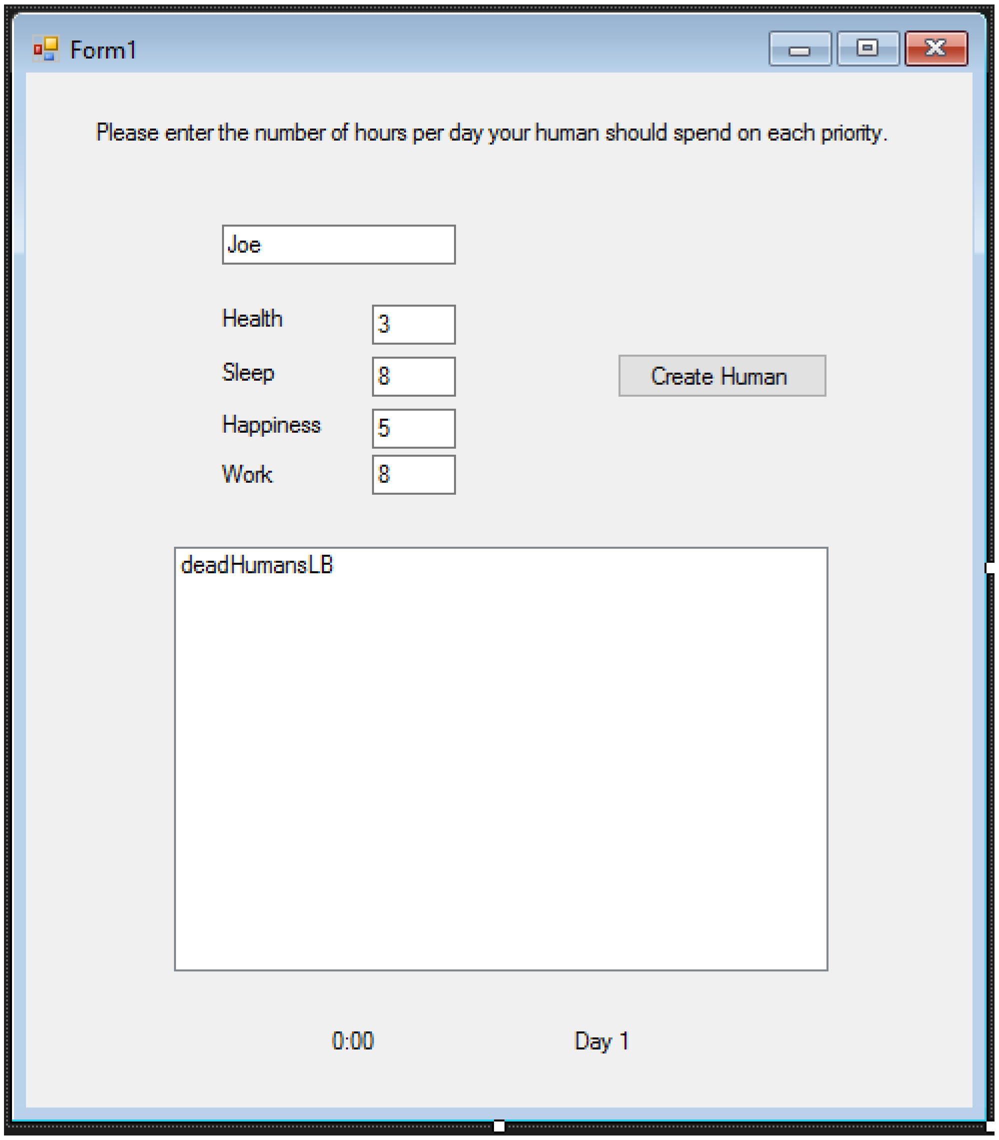


<u>Narrative and Code</u>



Health	7/10
Sleep	7/10
Happiness	7/10
Work	7/10

The only two classes that I needed to create for this app were two forms, Form1 and HumanForm. Form1 was the Observer class and HumanForm was the Observable class. I started this project by designing both forms; this provided me with a solid direction of what I needed to take care of next. To the left is a screenshot of the design view of my HumanForm, and Form1 is shown at the top of the next page.



Please enter the number of hours per day your human should spend on each priority.

Joe

Health	3
Sleep	8
Happiness	5
Work	8

deadHumansLB

0:00

Day 1

After I designed the forms, I started working on the code for the HumanForm. I started there because the Form1 class observes the HumanForm class, and therefore is dependent upon it. I had to write the code for the observable class first. I started by declaring an event and event handler for handling what to do when a human dies.

```
public delegate void HumanDeathEventHandler(object sender, HumanDeathEventArgs e);
public event HumanDeathEventHandler humanDied;
```

Then, I declared two constants which will be used later on to deal with the values of the health, sleep, happiness, and work variables:

```
const int MAX_VAL = 10;
const int STARTING_VAL = 7;
```

After this, I declared four variables to hold the number of hours per day spent on each priority, and four more to hold the human's success level at managing each priority.

```
int healthHours;
int sleepHours;
int happinessHours;
int workHours;

int happiness = STARTING_VAL;
int sleep = STARTING_VAL;
int work = STARTING_VAL;
int health = STARTING_VAL;
```

With all of my variables declared, I could start working on the code. The first thing I needed was a HumanForm constructor. This was very straightforward; it took several values as inputs and initialized my previously created variables to those values. It also set the text of the form appropriately.

```
public HumanForm(string name, int healthHours, int sleepHours, int happinessHours, int
       workHours)
{
    InitializeComponent();

    this.healthHours = healthHours;
    this.sleepHours = sleepHours;
    this.happinessHours = happinessHours;
    this.workHours = workHours;

    healthLabel.Text = STARTING_VAL + "/" + MAX_VAL;
    sleepLabel.Text = STARTING_VAL + "/" + MAX_VAL;
    happinessLabel.Text = STARTING_VAL + "/" + MAX_VAL;
    workLabel.Text = STARTING_VAL + "/" + MAX_VAL;

    nameLabel.Text = name;
}
```

Then, I created the event handling portion of the code, which this app was created to demonstrate. To do this, I created a class called HumanDeathEventArgs, which extended the EventArgs class. This class provides two arguments to event handlers when its events are fired. First, it gives the name of the dead human, and second, it gives a string describing the cause of death. The code for this class is as follows.

```
public class HumanDeathEventArgs : EventArgs
{
    private string name;

    public string deadHumanName
    {
        get { return name; }
        set { name = value; }
    }
    private string cause;

    public string causeOfDeath
    {
        get { return cause; }
        set { cause = value;}
    }

    public HumanDeathEventArgs(string name, string cause)
    {
        deadHumanName = name;
        causeOfDeath = cause;
    }
}
```

After I finished handling events, I moved on to the biggest method in the HumanForm class: the method that updates the human's health, sleep, happiness, and work variables every "hours." The method is passed an integer representing the hour of the day in military time and an integer representing the day of the week, where 0 is Sunday, 1 is Monday, and so on. Comments should help the reader follow along with the logic. The code is as follows:

```
public void updateHuman(int hour, int day)
{
    // First, update human based on how much time per day the human spends doing each
       item.
    // The first x hours of the day, starting at midnight, are devoted to sleep
    if (hour < sleepHours && sleep < MAX_VAL)
        sleep++;
    // Then the human goes to work
    else if (hour < sleepHours + workHours && work < MAX_VAL)
        work++;
    // Then the human takes care of health concerns like eating, exercising, etc.
    else if (hour < sleepHours + workHours + healthHours && health < MAX_VAL)
         health++;
    // Finally, the human relaxes until bedtime
    else if (happiness < MAX_VAL)
        happiness++;

    // Then, update the human based on the stresses of everyday life, determined based
       on the hour of the day.

    Random rng = new Random();

    // During the week (Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday = 5)
    if (day >= 1 && day <= 5)
    {
        int thingToChange = rng.Next(4);    // 0 means change health, 1 means sleep, 2
            happiness, and 3 work
        int amtToChange = rng.Next(3);      // Change it by zero, one, or two points.

        if (thingToChange == 0)
            health -= amtToChange;
        else if (thingToChange == 1)
```

```
            sleep -= amtToChange;
        else if (thingToChange == 2)
            happiness -= amtToChange;
        else
            work -= amtToChange;
    }

    // It's the weekend, so things change differently.
    else
    {
        int thingToChange = rng.Next(4);    // 0 means change health, 1 means sleep, 2
              happiness, and 3 work
        int amtToChange = rng.Next(2);      // Change it by zero or one points; the
              weekend is less stressful

        if (thingToChange == 0)
            health -= amtToChange;
        else if (thingToChange == 1)
            sleep -= amtToChange;
        else if (thingToChange == 2)
            happiness -= amtToChange;
        else
            work -= amtToChange;
    }

    // Update the form to display the human's new stats
    changeHealth(health);
    changeSleep(sleep);
    changeHappiness(happiness);
    changeWork(work);

    // Check for death and raise an event if necessary
    if (health < 0)
        humanDied(this, new HumanDeathEventArgs(nameLabel.Text, "undisclosed medical
              problems"));
    else if (sleep < 0)
         humanDied(this, new HumanDeathEventArgs(nameLabel.Text, "fatigue"));
    else if (happiness < 0)
         humanDied(this, new HumanDeathEventArgs(nameLabel.Text, "sadness"));
    else if (work < 0)
        humanDied(this, new HumanDeathEventArgs(nameLabel.Text, "extreme poverty"));
}
```

The functions changeHealth, changeSleep, changeHappiness, and changeWork, as well as three more similar ones in the Form1 class, were my solution to a problem I was having where I couldn't update the labels in my forms. These functions are all important to the functionality of the program, but unimportant to the understanding of the code, so they will not be included here. However, the full source code for this project is available at https://github.com/andreya-grzegorzewski/ in the Design-Patterns repository.

After this was done, I had only one more function to write. I discovered while writing the code for Form1 that I needed this function. It simply calculates whether or not a human is dead. A human dies when one of its levels falls below zero. The function is as follows:

```
public bool isDead()
{
    return (health < 0 || sleep < 0 || happiness < 0 || work < 0);
}
```

This was the last function I had to write in the HumanForm class. With that class finished, I moved on to Form1. I started again with variable declarations: a timer used to simulate days passing, two variables to keep track of the current hour and day, an array of humans, and a human counter.

```
System.Timers.Timer clock = new System.Timers.Timer(1000);

private static int day;
private static int hour;

HumanForm[] humans = new HumanForm[50]; // Can keep track of 50 humans at once
int numHumans = 0;
```

Next, I had to define what would happen every time the timer ticked. This updates the hour and day, then calls the updateHuman method from the HumanForm on each human.

```
private void clockEvent(Object source, ElapsedEventArgs e)
{
    hour++;
    if (hour == 24)
        hour = 0;

    changeHour(hour);

    if (hour == 0)
    {
        day++;
        changeDay(day);
    }

    for (int i = 0; i < numHumans; i++)
         humans[i].updateHuman(hour, day);
}
```

After that, I created the event handler. This code is called every time a HumanDeathEvent is fired. It uses the variables in HumanDeathEventArgs to display a string in the form letting the user know that a human died. Then, it updates the array of humans so that the dead human will not still be updated every time a timer event is fired.

```
void humanFormDeath(object sender, HumanForm.HumanDeathEventArgs e)
{
    string deathString = "Oops! " + e.deadHumanName + " died of " + e.causeOfDeath + ".";
    changeDeaths(deathString);
    for (int i = 0; i < numHumans; i++)
    {
        if (humans[i].isDead())
        {
            humans[i] = null;

            for (int j = i; j < numHumans - 1; j++)
            {
                humans[j] = humans[j + 1];
            }

            numHumans--;
            humans[numHumans] = null;
            break;
        }
    }
}
```

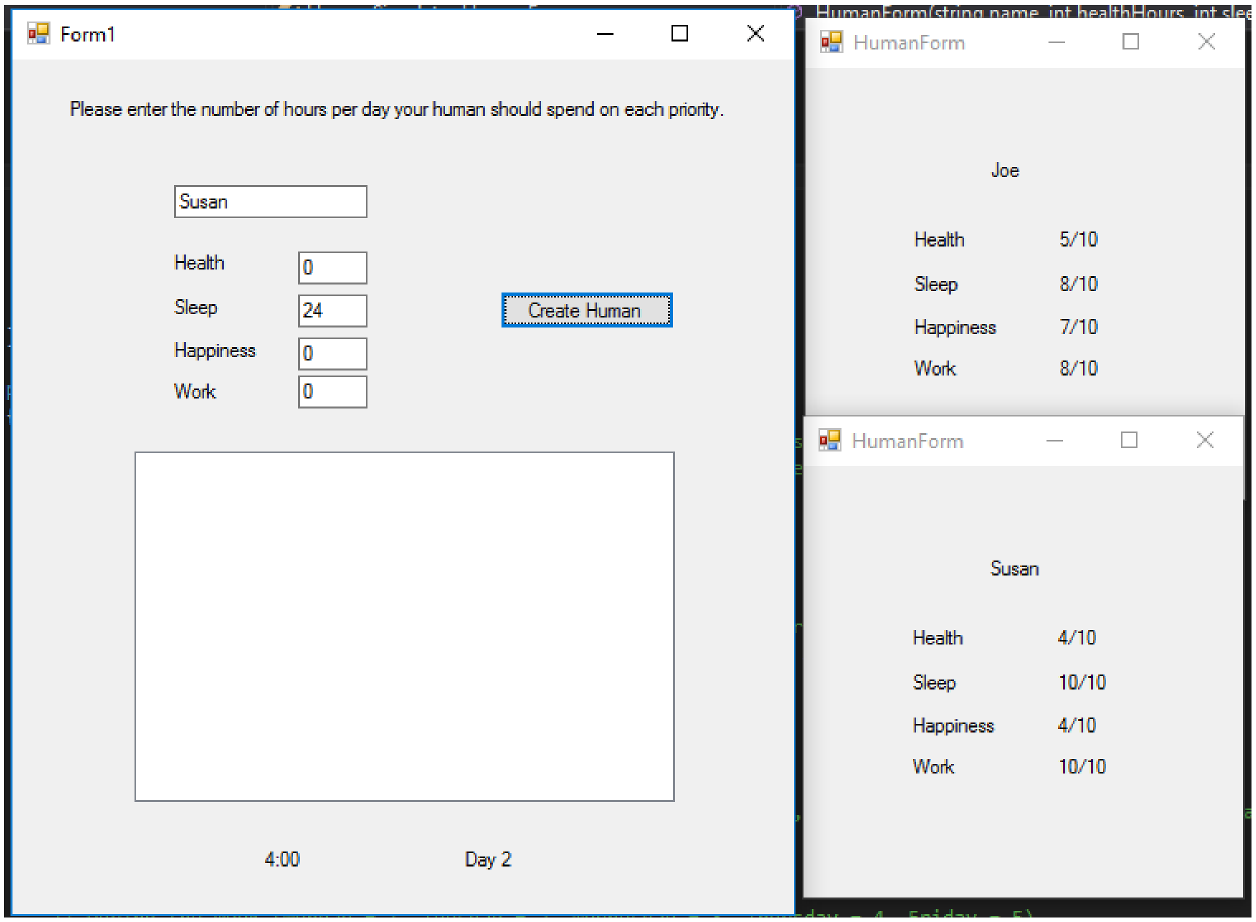
Finally, I could write the last portion of code in this class: the method that is called when the Create Human button is pressed. This method makes a new HumanForm and gives it an event handler, and with it, the code for this project was complete.

```
private void createHumanButton_Click(object sender, EventArgs e)
{
    // Gets the variables entered by the user
    int healthHours = Convert.ToInt32(healthTB.Text);
    int sleepHours = Convert.ToInt32(sleepTB.Text);
    int happinessHours = Convert.ToInt32(happinessTB.Text);
    int workHours = Convert.ToInt32(workTB.Text);
    string name = nameTB.Text;

    // Makes sure the input is valid
    if (healthHours + sleepHours + happinessHours + workHours != 24)
        MessageBox.Show("Please make sure that the numbers in the boxes add up to 24.");

    // If the input is valid...
    else
    {
        // Start the timer if it hasn't already been started
        if (!clock.Enabled)
        {
            clock.Elapsed += clockEvent;
            clock.Start();

            day = 1;
            hour = 0;
        }

        // Create a new human with the variables entered by the user
        HumanForm newHuman = new HumanForm(name, healthHours, sleepHours, happinessHours,
              workHours);

        // Add the human to the list
        humans[numHumans] = newHuman;
        numHumans++;
```

```
            // Add an event handler
            newHuman.humanDied += new HumanForm.HumanDeathEventHandler(humanFormDeath);

            // Prepare and show the new human form
            newHuman.Location = new Point(Location.X + Size.Width + 10, Location.Y);
            newHuman.Show();
        }
    }
```

The Deliverable

Below are screenshots of the program in action, with captions to explain them.



Form1

Please enter the number of hours per day your human should spend on each priority.

	Susan
Health	0
Sleep	24
Happiness	0
Work	0

4:00 Day 2

HumanForm

Joe

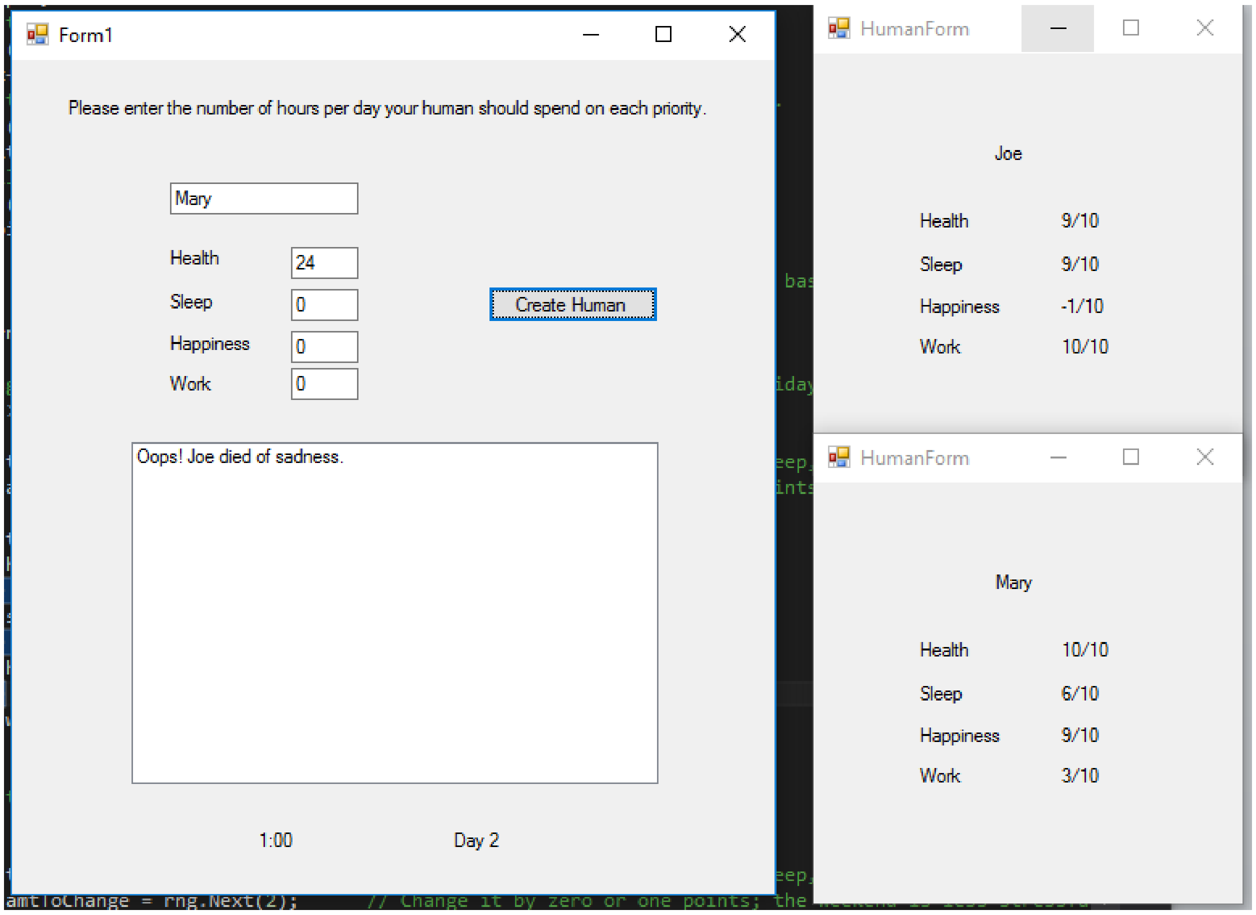
Health	5/10
Sleep	8/10
Happiness	7/10
Work	8/10

HumanForm

Susan

Health	4/10
Sleep	10/10
Happiness	4/10
Work	10/10

The app just after two humans were created (before either of them had a chance to die)



Form1

Please enter the number of hours per day your human should spend on each priority.

Mary

Health	24
Sleep	0
Happiness	0
Work	0

```
Oops! Joe died of sadness.
```

1:00 Day 2

HumanForm

Joe

Health	9/10
Sleep	9/10
Happiness	-1/10
Work	10/10

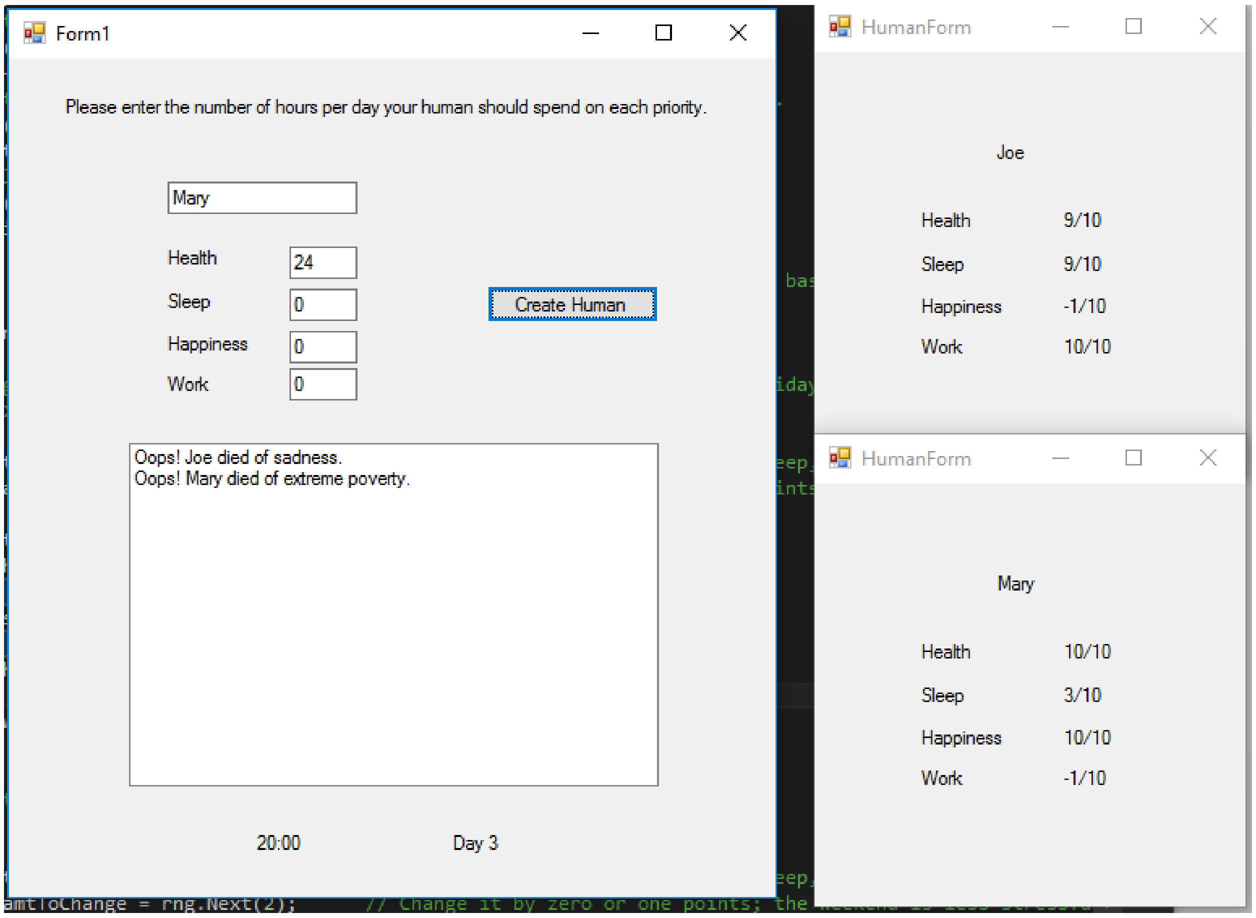
HumanForm

Mary

Health	10/10
Sleep	6/10
Happiness	9/10
Work	3/10

```
amtToChange = rng.Next(2);          // Change it by zero or one points; the
```

The app after one human died. The list box shows the death event, and we can see in Joe’s form that he did indeed die of sadness.



Please enter the number of hours per day your human should spend on each priority.

Mary

Health	24
Sleep	0
Happiness	0
Work	0

```
Oops! Joe died of sadness.
Oops! Mary died of extreme poverty.
```

20:00 Day 3

Joe

Health	9/10
Sleep	9/10
Happiness	-1/10
Work	10/10

Mary

Health	10/10
Sleep	3/10
Happiness	10/10
Work	-1/10

```
amtToChange = rng.Next(2);     // Change it by zero or one points; the
```

The app after both humans died. We can see that Joe’s stats were no longer updated after he died.

Observations

While this modified pattern was fairly simple and straightforward, I ran into a lot of unforeseen difficulties with updating my forms. Once these were overcome, however, I found that I had no problems at all implementing the observer pattern. My events worked as planned the first time with no tweaking or bug-squashing necessary. I like that this pattern gives forms a way to communicate with one another, and I could see myself using it in more complex programs in the future. In fact, this pattern would have been very helpful to me when I was working on my educational program during Programming 2 last semester, which had one main form and a number of additional forms that were created by the main form. The observer pattern could have greatly enhanced the functionality of the main form by helping to implement scoring and other components of gamification.