Proxy Pattern Assignment

Design Patterns

Jacob Hartman

1 November 2016

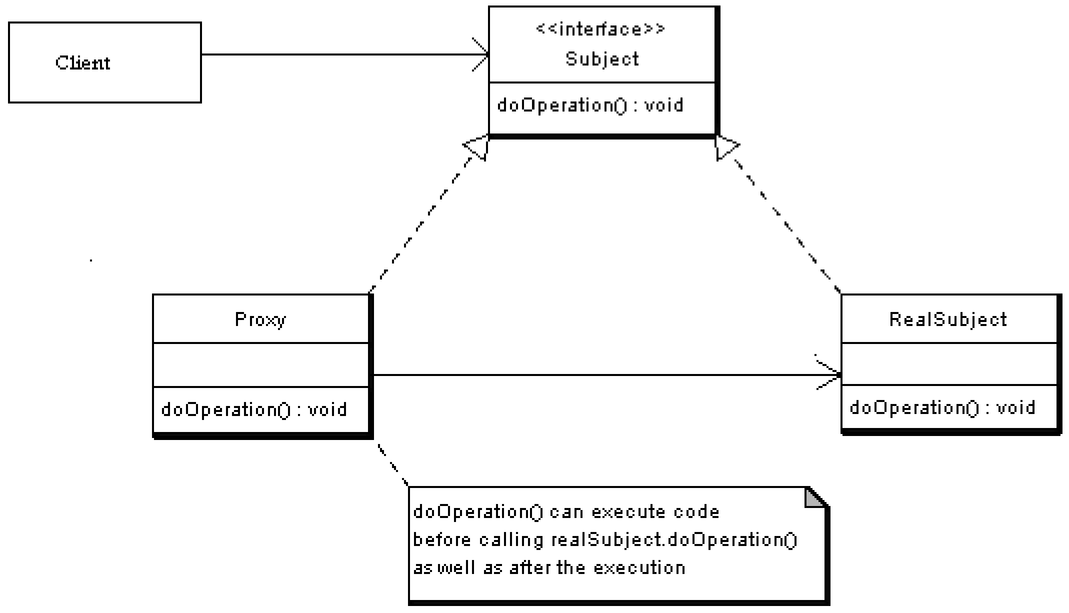
Introduction

This assignment required the creation of a program demonstrating the use of the proxy pattern. The proxy pattern creates a buffer between a client and a subsystem, allowing additional code to be executed when using the subsystem.

The application I created for this assignment is designed to simulate system management of a starship. The user takes the role of the captain, with the engineer acting as the proxy for the ship's systems. The captain may ask the engineer for information about the ship, and the engineer will turn the ship's raw information into an appropriate message for the captain.

UML Diagram

image taken from http://www.oodesign.com/proxy-pattern.html



Above is the UML diagram for the proxy pattern. It shows that the pattern requires two classes and an interface aside from the client. Both the Proxy class and the RealSubject class implement the Subject interface and its doOperation method. The Proxy also instantiates a RealSubject object within it, and the Proxy's doOperation method calls the RealSubject's doOperation method.

I created an interface called Subject, with a long list of doOperation methods. These methods are getShields, getHull, getPower, getWeapons, getSensors, getComms, jamFrequencies, detonate, fireWeapons, hullIntegrity, and torpCount. The Starship class takes the role of the RealSubject class, implementing the interface and its methods. I then created the Engineer class, which acts as the Proxy class. The client, Form1, communicates with the Engineer class, which in turn communicates with the Starship class.

Narrative

First, I wrote the Subject interface.

```
public interface Subject // this is the subject interface
{
    string getShields();
    string getHull();
    string getPower();
    string getWeapons();
    string getSensors();
    string getComms();
    void jamFrequencies();
    void detonate();
    void fireWeapons();
    int hullIntegrity();
    int torpCount();
}
```

The interface creates a list of signatures for the methods that later classes must implement. This makes it much simpler to create the subclasses. The next class I wrote was the Starship class.

```
public class Starship : Subject // this is the RealSubject class
{
    int shields;
    int hull;
    int power;
    int torpedoCount;
    bool phasers;
    bool sensors;
    bool comms;
    public Starship()
    {
        shields = 100;
        hull = 100;
        power = 100;
        torpedoCount = 10;
        phasers = true;
        sensors = true;
        comms = true;
    }
```

```
        public string getShields()
        {
            return shields + "%";
        }

        public string getHull()
        {
            return hull + "%";
        }

        public string getPower()
        {
            return power + "%";
        }

        public string getWeapons()
        {
            return torpedoCount + " torpedoes";
        }

        public string getSensors()
        {
            if (sensors)
            {
                return "active";
            }
            else
            {
                return "being jammed";
            }
        }

        public string getComms()
        {
            if (comms)
            {
                return "functioning";
            }
            else
            {
                return "being jammed";
            }
        }
```

The getter methods return values of the starship's attributes as strings. Numeric percentage values are returned with the % sign at the end. The boolean values are returned as appropriate terms rather than raw values.

```
        public void jamFrequencies()
        {
            sensors = false;
            comms   = false;
        }

        public void detonate()
        {
            if (shields > 0)
            {
                shields -= 50;
            }
            else
            {
                shields = 0;
            }
            if (hull > 29)
            {
                hull -= 30;
            }
            else
            {
                hull = 0;
            }
        }

        public void fireWeapons()
        {
            if (torpedoCount > 0){
                torpedoCount--;
            }
            power -= 5;
        }
```

The next three methods all change some attributes of the ship. The first jams all the ship's frequencies to demonstrate that the systems can be disabled. The detonate method damages the ship, and can potentially destroy it. The fireWeapons method fires a torpedo to show that the weapons are limited.

```
        public int hullIntegrity()
        {
            return hull;
        }

        public int torpCount()
        {
            return torpedoCount;
        }
    }
```

The remaining methods return raw data values from the ship for use in if-else statements, so that no impossible actions are taken. This Starship class is instantiated by the Engineer class.

```
    public class Engineer : Subject // this is the Proxy class
    {
        Starship ship;
        public Engineer()
        {
            ship = new Starship();
        }
        public string getShields()
        {
            return "Our shields are at " + ship.getShields() + ", Captain.";
        }

        public string getHull()
        {
            return "Our hull integrity is at " + ship.getHull() + ", Captain.";
        }
        public string getPower()
        {
            return "Our power level is at " + ship.getPower() + ", Captain.";
        }

        public string getWeapons()
        {
            return "We currently have " + ship.getWeapons() + ", Captain.";
        }

        public string getSensors()
        {
            return "Our sensors are " + ship.getSensors() + ", Captain.";
        }

        public string getComms()
        {
            return "Our communication systems are " + ship.getComms() + ", Captain.";
        }
```
The getter methods pull in the values from the ship, which is an instance of the Starship class. The methods then add some text so that the returned string is an actual message that is meaningful to a user.
```
        public void jamFrequencies()
        {
            ship.jamFrequencies();
        }

        public void detonate()
        {
            ship.detonate();
        }

        public void fireWeapons()
        {
            ship.fireWeapons();
        }
```
The next three methods merely execute the ship's methods of the same names.
```
        public int hullIntegrity()
        {
            return ship.hullIntegrity();
        }

        public int torpCount()
        {
            return ship.torpCount();
        }
    }
```
The last two methods return the value of the ship's methods, providing access to raw data for use in if-else statements. The remaining code is found in the client, Form1.
```
    public partial class Form1 : Form
    {
        Subject joe;
        public Form1()
        {
            InitializeComponent();
            joe = new Engineer();
        }
        public int getIndex()
        {
            int s;
            if (rb_shields.Checked)
            {
                s = 0;
            }
            else if (rb_hull.Checked)
            {
                s = 1;
            }
            else if (rb_power.Checked)
            {
                s = 2;
            }
            else if (rb_weapons.Checked)
            {
                s = 3;
            }
            else if (rb_sensors.Checked)
            {
                s = 4;
            }
            else if (rb_comms.Checked)
            {
                s = 5;
            }
            else
            {
                s = -1;
            }
            return s;
        }
```

There is a set of radio buttons on the main form. These radio buttons allow the user, who takes on the role of captain, to ask the engineer for information about the ship's systems. The getIndex method provides a simple way of getting the selected radio button's index as an integer. This is used later for identifying which option the user has selected.

```
        private void btn_ask_Click(object sender, EventArgs e)
        {
            if (joe.hullIntegrity() == 0)
                return;
            int s = getIndex();
            string message = "error";
            if (s == 0)
            {
                message = joe.getShields();
            }
            else if (s == 1)
            {
                message = joe.getHull();
            }
            else if (s == 2)
            {
                message = joe.getPower();
            }
            else if (s == 3)
            {
                message = joe.getWeapons();
            }
            else if (s == 4)
            {
                message = joe.getSensors();
            }
            else if (s == 5)
            {
                message = joe.getComms();
            }
            lb_status.Items.Add(message);
            lb_status.TopIndex = lb_status.Items.Count - 1;
        }
```

When the ask button is clicked, the getIndex method is called to identify the selected option. The appropriate method is then called to get the message the user has asked for and display it.

```
        private void btn_fire_Click(object sender, EventArgs e)
        {
            if (joe.torpCount() > 0)
            {
                joe.fireWeapons();
                lb_status.Items.Add("Firing now, Captain.");
                lb_status.TopIndex = lb_status.Items.Count - 1;
            }
            else
            {
                lb_status.Items.Add("We are out of weapons, Captain.");
                lb_status.TopIndex = lb_status.Items.Count - 1;
            }
        }
```

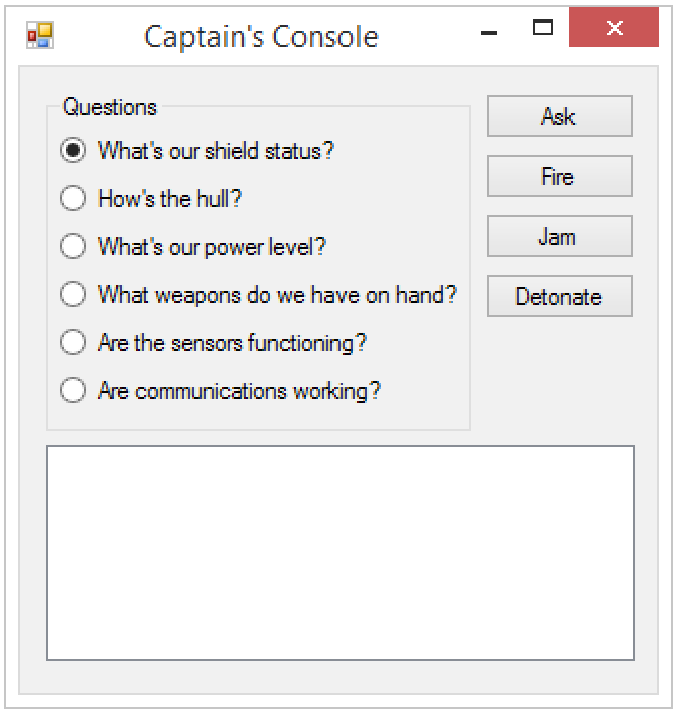
The fire button first checks if there are any remaining torpedoes when clicked. If there are, the engineer fires one as a test. Otherwise, the user is informed that the ship is out of torpedoes.

```
        private void btn_jam_Click(object sender, EventArgs e)
        {
            joe.jamFrequencies();
            lb_status.Items.Add("Captain, something is jamming us.");
            lb_status.TopIndex = lb_status.Items.Count - 1;
            btn_jam.Enabled = false;
        }
```

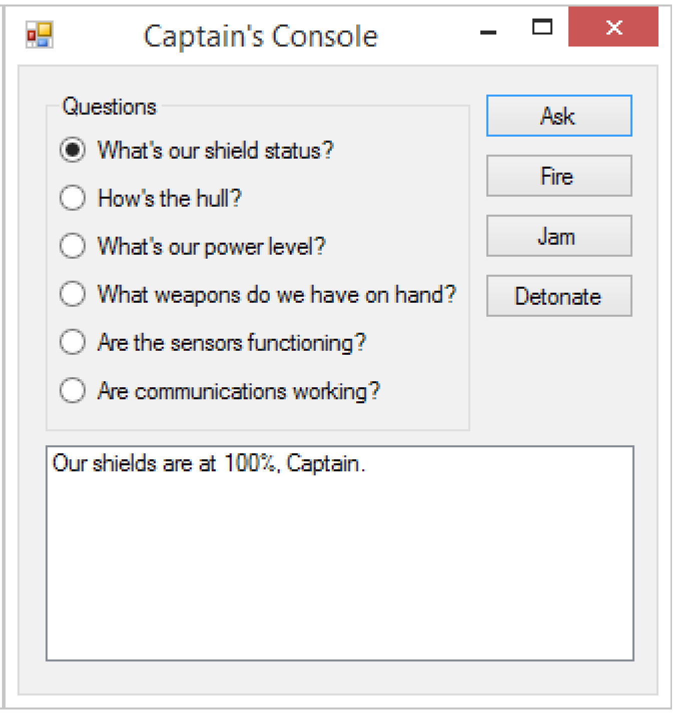
The jam button causes the ship's frequencies to be jammed. The user is informed of this, and the appropriate systems are disabled.

```
        private void btn_explode_Click(object sender, EventArgs e)
        {
            joe.detonate();
            if (joe.hullIntegrity() > 0)
            {
                lb_status.Items.Add("Captain, an explosive has gone off within the ship.");
                lb_status.TopIndex = lb_status.Items.Count - 1;
            }
            else
            {
                lb_status.Items.Add("Notice: the ship has been destroyed.");
                lb_status.TopIndex = lb_status.Items.Count - 1;
                btn_ask.Enabled = false;
                btn_fire.Enabled = false;
                btn_jam.Enabled = false;
                btn_explode.Enabled = false;
            }
        }
    }
```

When the detonate button is clicked, an explosive is set off on the ship. This causes damage to the ship, and if the damage is too great, the ship is destroyed. Destroying the ship immediately disables all commands. Below are some screenshots of the program in use.

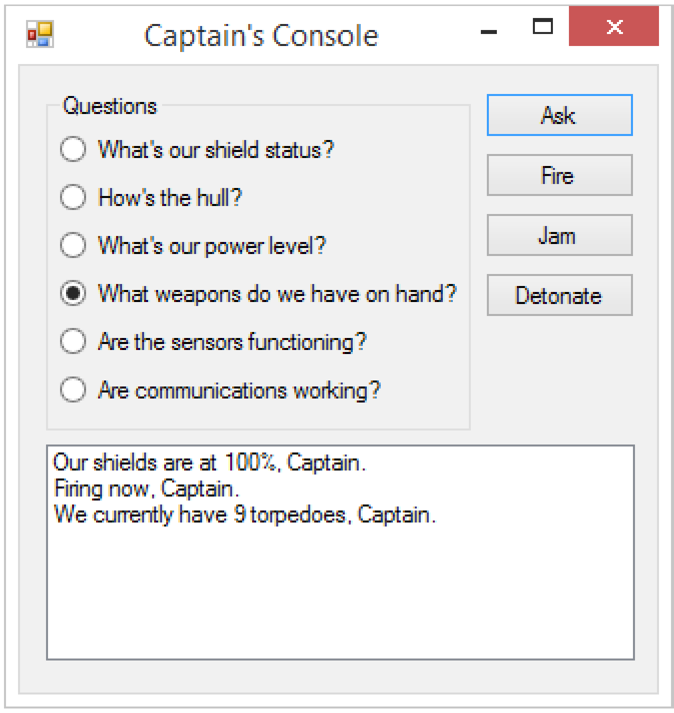

Captain's Console
Questions
What's our shield status?
How's the hull?
What's our power level?
What weapons do we have on hand?
Are the sensors functioning?
Are communications working?
Ask
Fire
Jam
Detonate

Initial view of the application.

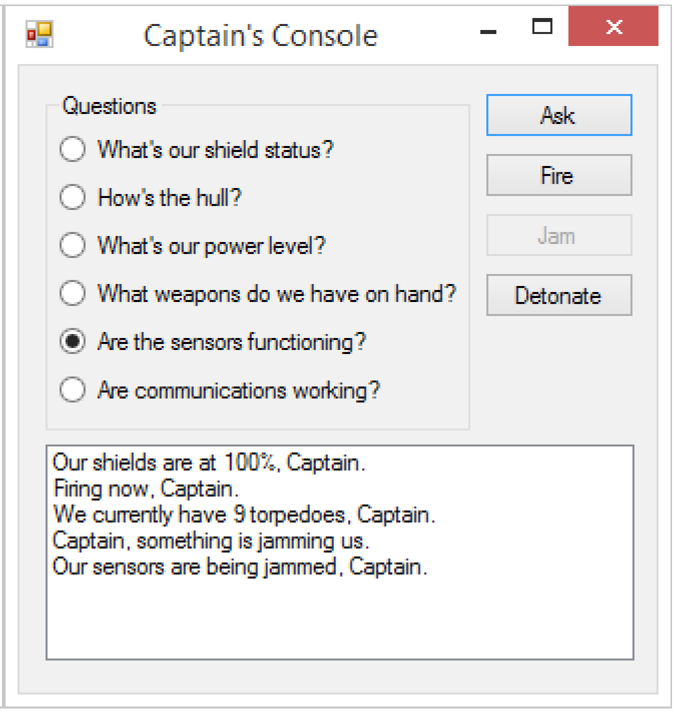

Captain's Console
Questions
What's our shield status?
How's the hull?
What's our power level?
What weapons do we have on hand?
Are the sensors functioning?
Are communications working?
Ask
Fire
Jam
Detonate

```
Our shields are at 100%, Captain.
```

The user can select a radio button and hit ask. The response appears in the box below.

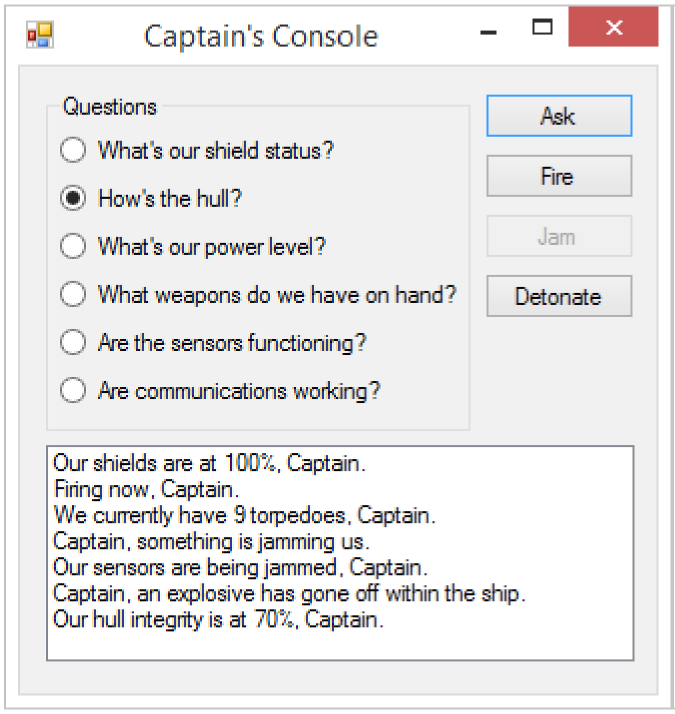

Captain's Console
Questions
What's our shield status?
How's the hull?
What's our power level?
What weapons do we have on hand?
Are the sensors functioning?
Are communications working?
Ask
Fire
Jam
Detonate

```
Our shields are at 100%, Captain.
Firing now, Captain.
We currently have 9 torpedoes, Captain.
```


Captain's Console
Questions
What's our shield status?
How's the hull?
What's our power level?
What weapons do we have on hand?
Are the sensors functioning?
Are communications working?
Ask
Fire
Jam
Detonate

```
Our shields are at 100%, Captain.
Firing now, Captain.
We currently have 9 torpedoes, Captain.
Captain, something is jamming us.
Our sensors are being jammed, Captain.
```

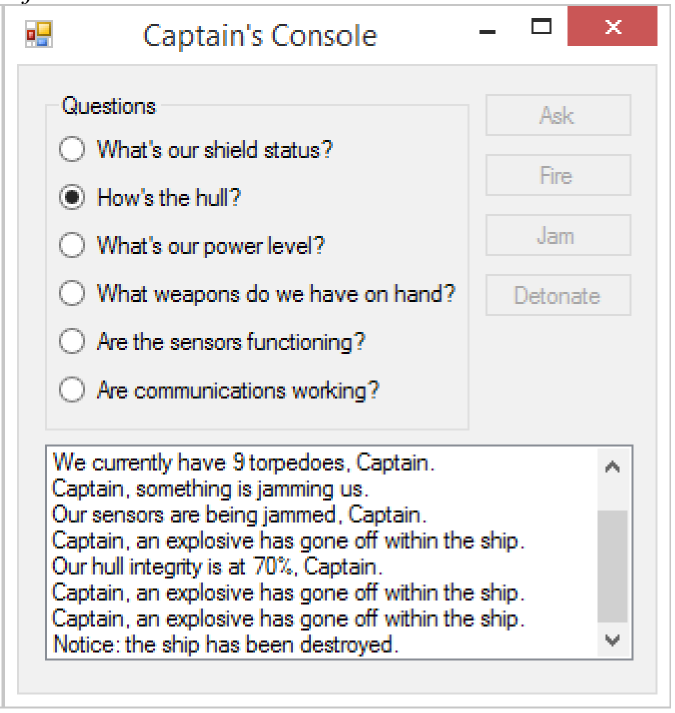
Hitting fire uses up a torpedo.


Captain's Console
Questions
What's our shield status?
How's the hull?
What's our power level?
What weapons do we have on hand?
Are the sensors functioning?
Are communications working?
Ask
Fire
Jam
Detonate

```
Our shields are at 100%, Captain.
Firing now, Captain.
We currently have 9 torpedoes, Captain.
Captain, something is jamming us.
Our sensors are being jammed, Captain.
Captain, an explosive has gone off within the ship.
Our hull integrity is at 70%, Captain.
```

The detonate command damages the ship.

The jam command can only be activated once, but it jams sensors and communications.


Captain's Console
Questions
What's our shield status?
How's the hull?
What's our power level?
What weapons do we have on hand?
Are the sensors functioning?
Are communications working?
Ask
Fire
Jam
Detonate

```
We currently have 9 torpedoes, Captain.
Captain, something is jamming us.
Our sensors are being jammed, Captain.
Captain, an explosive has gone off within the ship.
Our hull integrity is at 70%, Captain.
Captain, an explosive has gone off within the ship.
Captain, an explosive has gone off within the ship.
Notice: the ship has been destroyed.
```

If the detonate command is used too much, the ship is destroyed. All commands are disabled when this happens.

<u>Observations</u>

The proxy pattern is rather interesting. It allows two systems to communicate indirectly without ever actually exchanging any information. It could be useful in data processing. I also believe it could be used effectively when implementing legacy code. I feel that my submission fits the assignment well. It implements the proxy pattern as specified, and it is fairly simple for the user to understand the underlying concept. This assignment provided me with a new possible approach to handling legacy code. I believe it will be useful in the future.