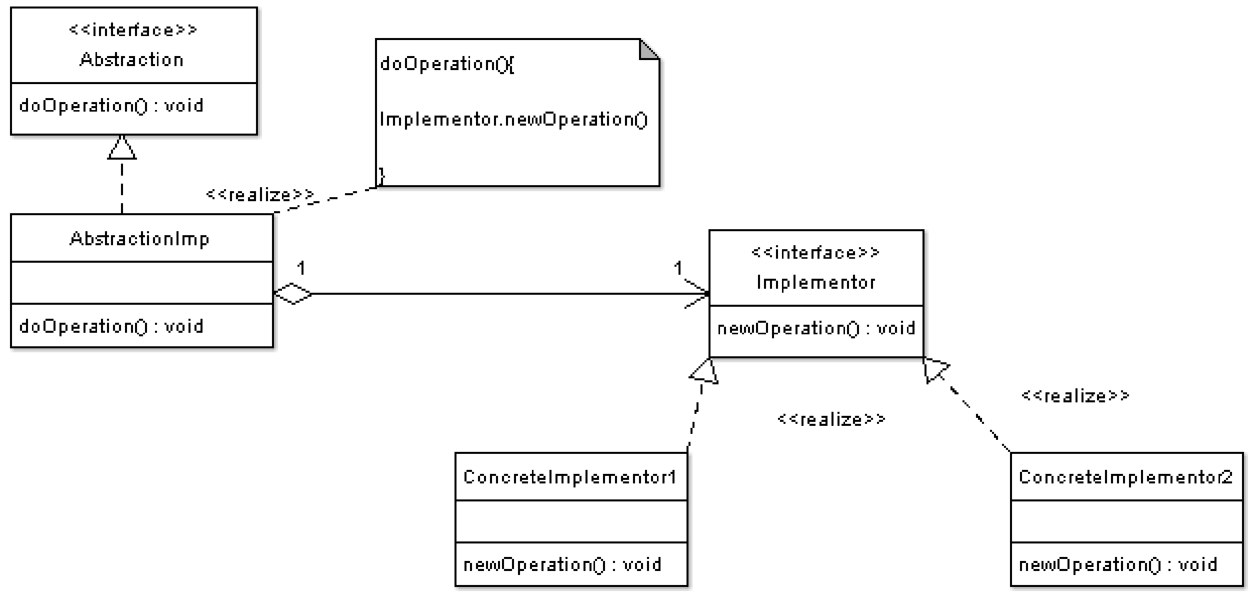
Jack Raney

Design Patterns

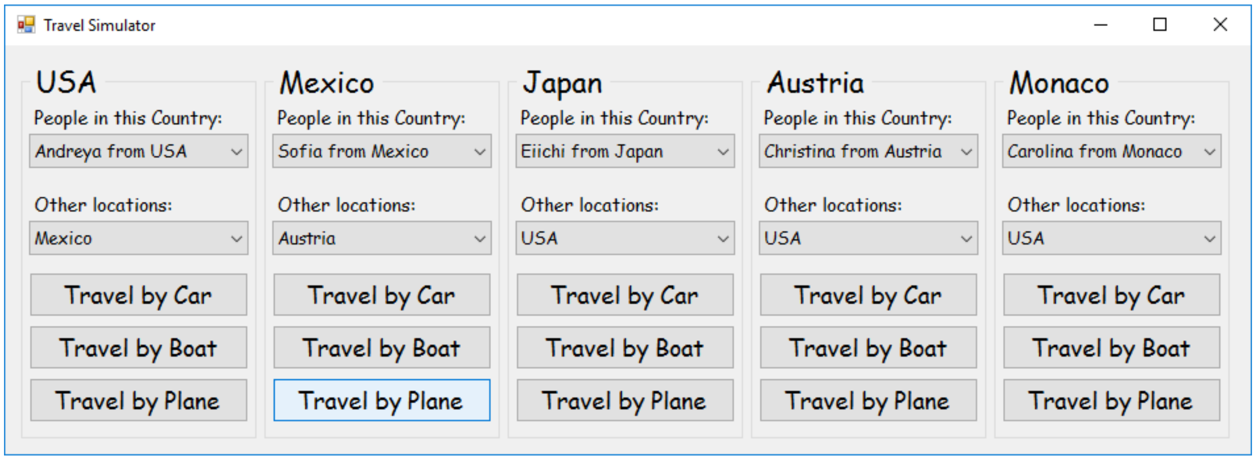
11/8/2016

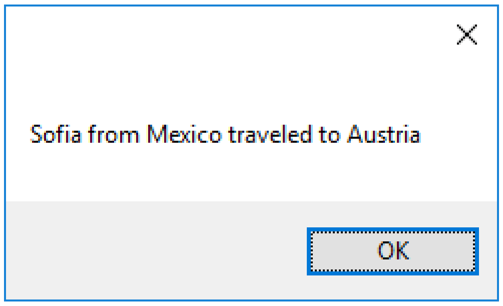
The Bridge Pattern

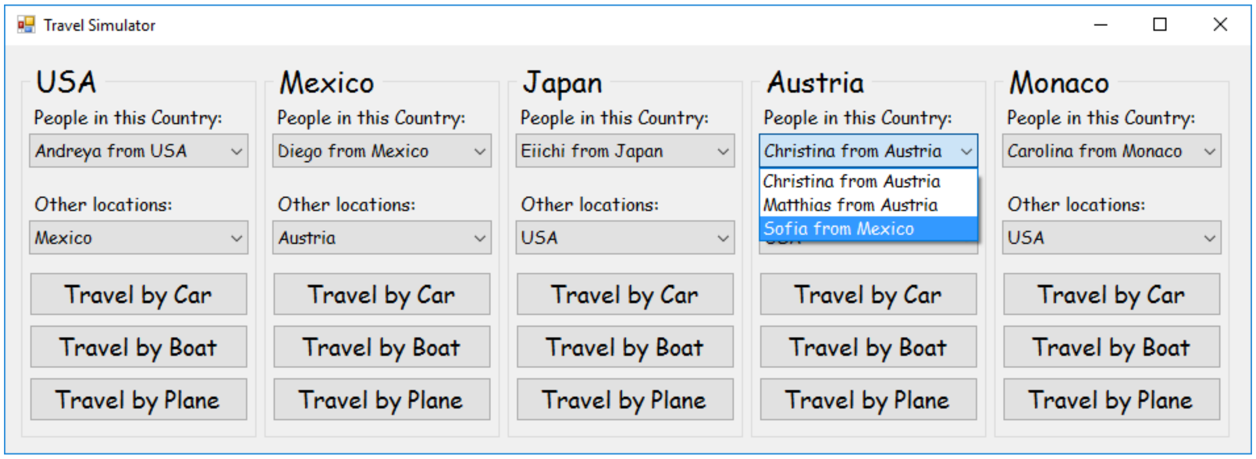
This paper was written to describe the Bridge Pattern, a design pattern whose intent is, according to oodesign, "to decouple abstraction from implementation so that the two can vary independently". This means that an object that serves as an abstraction contains an implementor object that can be interchanged to change the implementation of one or more of the abstraction object's methods. The UML diagram for the Bridge Pattern is shown below:



To demonstrate the Bridge Pattern, I created the program Travel.exe:




Sofia from Mexico traveled to Austria



Code for Travel.exe

Country.cs

```
public class Country
    {
        private string name;
        private string isle;
        private bool isCoastal;
        private bool hasAirport;

        public Country(string n, string c, bool s, bool a)
        {
            name       = n;
            isle       = c;
            isCoastal  = s;
            hasAirport = a;
        }

        public override string ToString()
        {
            return name;
        }

        public string getIsland()
        {
            return isle;
        }

        public bool getCoastal()
        {
            return isCoastal;
        }

        public bool getAerial()
        {
            return hasAirport;
        }
    }
```

The Country class is not part of the UML diagram, but the different classes that are in the UML diagram interact with Country objects.

Country objects have properties for a country's name, the island it is on, a Boolean for whether or not it borders the ocean, and a Boolean for whether or not the country contains an airport. These properties are set via the constructor and methods are available to get the values.

ToString() overrides are in every non-interface class of this program to pass names of objects into the form.

Traveler.cs

```
public interface Traveler   //This is the Abstraction interface
    {
        void travel(Country from, Country to);
        void setVehicle(Transporter v);
        Country getLocation();
    }
```

Objects that inherit the Traveler interface contain methods to travel (the method analogous to the UML diagram's doOperation() method), set the used vehicle (the Implementor), and to get the current Country the traveler is in.

Person.cs

```
public class Person : Traveler  //This is an AbstractionImp class
    {
        private Transporter vehicle;

        private string  name;
        private Country home;
        private Country location;

        public Person(string n, Country h)
        {
            name     = n;
            home     = h;
            location = h;
        }

        public override string ToString()
        {
            return name + " from " + home.ToString();
        }

        public void setVehicle(Transporter v)
        {
            vehicle = v;
        }

        public Country getLocation()
        {
            return location;
        }

        public void travel(Country from, Country to)
        {
            location = vehicle.transport(from, to);
        }
    }
```

The Person class is the child of the traveler interface. It contains properties for a person's name, home country, and current country he/she is in; these are set in the constructor.

A Transporter object is also stored in Person that determines what method of transportation the person will take.

See Traveler.cs's description for descriptions of setVehicle() and getLocation().

Because travel() is the UML diagram's doOperation() class, it uses the Transporter object's newOperation() method (transport()).

Transport.cs

```
public interface Transporter    //This is the Implementor interface
    {
        Country transport(Country from, Country to);
    }
```

Objects that inherit the Transport interface all have a transport() method that differs depending on what mode of transportation it is. They follow the same format, however:

1. The function determines if that mode of transportation is available in the country the person is coming from.
2. If it is available, return the country they intend to travel to (they were able to travel).
3. Otherwise, return the country they are coming from (they could not travel using that method of transportation.

Car objects will successfully transport if the two countries are on the same island.

Boat objects will successfully transport if both countries border the ocean.

Airplane objects will successfully transport if both countries have an airport.

Car.cs

```
public class Car : Transporter    //This is a ConcreteImplementor
    {
        public Country transport(Country from, Country to)
        {
            if (from.getIsland().Equals(to.getIsland()))
            {
                return to;
            }

            return from;
        }

        public override string ToString()
        {
            return "car";
        }
    }
```

Boat.cs

```
public class Boat : Transporter //This is a ConcreteImplementor
    {
        public Country transport(Country from, Country to)
        {
            if (from.getCoastal() && to.getCoastal())
            {
                return to;
            }

            return from;
        }

        public override string ToString()
        {
            return "boat";
        }
    }
```

Airplane.cs

```
public class Airplane : Transporter //This is a ConcreteImplementor
    {
        public Country transport(Country from, Country to)
        {
            if (from.getAerial() && to.getAerial())
            {
                return to;
            }

            return from;
        }

        public override string ToString()
        {
            return "airplane";
        }
    }
```

Form1.cs

```
public partial class Form1 : Form
    {
        List<Country> countries = new List<Country> {
            new Country("USA"     , "Americas", true , true) ,     //0
            new Country("Mexico" , "Americas", true , true) ,     //1
            new Country("Japan"   , "Japan"    , true , true) ,     //2
            new Country("Austria", "Eurasia" , false, true) ,     //3
            new Country("Monaco" , "Eurasia" , true , false),     //4
        };

        public Form1()
        {
            InitializeComponent();

            usaPeople.Items.Add(new Person("Andreya" , countries[0]));
            usaPeople.Items.Add(new Person("Dave"     , countries[0]));
            usaPeople.SelectedIndex = 0;
            usaLocs.DataSource = setList(0);

            mexPeople.Items.Add(new Person("Diego", countries[1]));
            mexPeople.Items.Add(new Person("Sofia", countries[1]));
            mexPeople.SelectedIndex = 0;
            mexLocs.DataSource = setList(1);

            japanPeople.Items.Add(new Person("Eiichi", countries[2]));
            japanPeople.Items.Add(new Person("Hanako", countries[2]));
            japanPeople.SelectedIndex = 0;
            japanLocs.DataSource = setList(2);

            ausPeople.Items.Add(new Person("Christina", countries[3]));
            ausPeople.Items.Add(new Person("Matthias" , countries[3]));
            ausPeople.SelectedIndex = 0;
            ausLocs.DataSource = setList(3);

            monPeople.Items.Add(new Person("Carolina", countries[4]));
            monPeople.Items.Add(new Person("Daniel"   , countries[4]));
            monPeople.SelectedIndex = 0;
            monLocs.DataSource = setList(4);
        }

        private List<Country> setList(int excludeIndex)
        {
            List<Country> l = new List<Country>();
            for (int i = 0; i < countries.Count; i++)
            {
                if (i != excludeIndex)
                {
                    l.Add(countries[i]);
                }
            }

            return l;
        }
```

These countries were chosen to fulfill all the different possibilities of unsuccessful travel (countries on different islands, a landlocked country, and a country that doesn't have an airport).

Upon initialization of the program, the ComboBoxes(and therefore the countries) are populated with people and vailable countries to travel to.

The setList() method adds the countries to the location ComboBoxes, excluding the country that ComboBox is part of. (no traveling to and from the same country is allowed in this demo).

```
        private void travel(ComboBox p, Country f, ComboBox t,
Transporter v)
        {
            Person person = (Person) p.SelectedItem;
            Country newLoc = (Country) t.SelectedItem;

            person.setVehicle(v);
            person.travel(f, newLoc);

            if (person.getLocation().Equals(f))
            {
                MessageBox.Show(person.ToString() + " can't travel to "
+ newLoc.ToString() + " by " + v.ToString());
            }

            else
            {
                MessageBox.Show(person.ToString() + " traveled to " +
newLoc.ToString());
                move(p, newLoc);
            }
        }

        private void move(ComboBox f, Country t)
        {
            Person p = (Person) f.SelectedItem;
            f.Items.Remove(p);

            int cIndex = 0;
            for (int i = 0; i < countries.Count; i++)
            {
                if (countries[i].Equals(t))
                    cIndex = i;
            }

            switch (cIndex)
            {
                case 0: usaPeople.Items.Add(p);
                    if (usaPeople.Items.Count == 1)
                        usaPeople.SelectedIndex++;
                    break;
                case 1: mexPeople.Items.Add(p);
                    if (mexPeople.Items.Count == 1)
                        mexPeople.SelectedIndex++;
                    break;
                case 2: japanPeople.Items.Add(p);
                    if (japanPeople.Items.Count == 1)
                        japanPeople.SelectedIndex++;
                    break;
                case 3: ausPeople.Items.Add(p);
                    if (ausPeople.Items.Count == 1)
                        ausPeople.SelectedIndex++;
                    break;
                default: monPeople.Items.Add(p);
                    if (monPeople.Items.Count == 1)
                        monPeople.SelectedIndex++;
                    break;
            }

            if (f.Items.Count > 0)
            {
                f.SelectedIndex++;
            }
        }
```

The travel() method grabs the selected person and location from their ComboBoxes, sets the selected person's mode of transportation, and then calls person's travel() function. A message box will relay to the user whether or not the transfer was successful. Upon success, the move() function is called to update the GUI.

move() updates the ComboBoxes' lists by removing the person from the old country's ComboBox and adding it to the new country's ComboBox.

```
private void usaCar_Click(object sender, EventArgs e)
{
    if (usaPeople.Items.Count > 0)
    {
        travel(usaPeople, countries[0], usaLocs, new Car());
    }

    else
    {
        MessageBox.Show("No one is in " + countries[0].ToString());
    }
}
```

Every button's function contains an if-else statement; if there are people still in the country, then people can be moved to other countries. Otherwise, the user is informed that there is no one in that country that can travel.

The only differences between button events are the ComboBoxes, countries, and modes of transportation involved.

Reflection

In my opinion, this pattern is one of the hardest to fully understand, as it requires a good understanding of abstraction and implementation. Previously, I never considered that implementation could be in an entirely different class than the abstraction. I believe creating this program and paper have allowed me to understand it well, though (after all, it basically forces me to understand so I can know what I’m talking about).