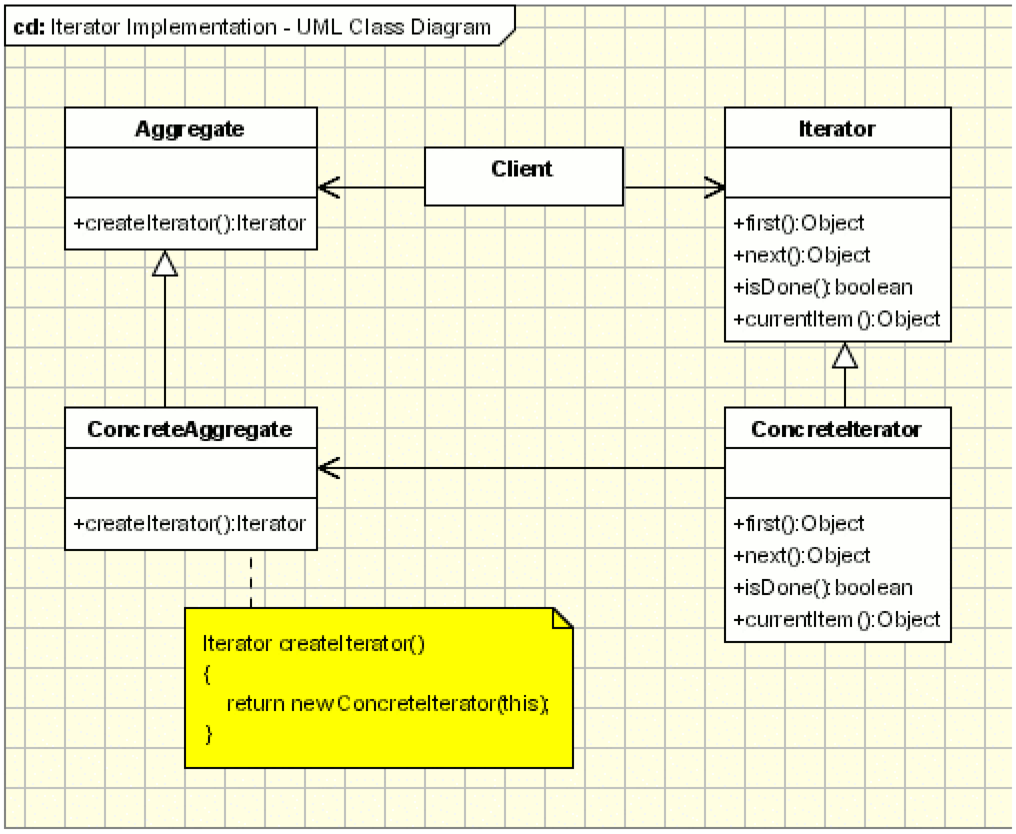
Jack Raney

Design Patterns

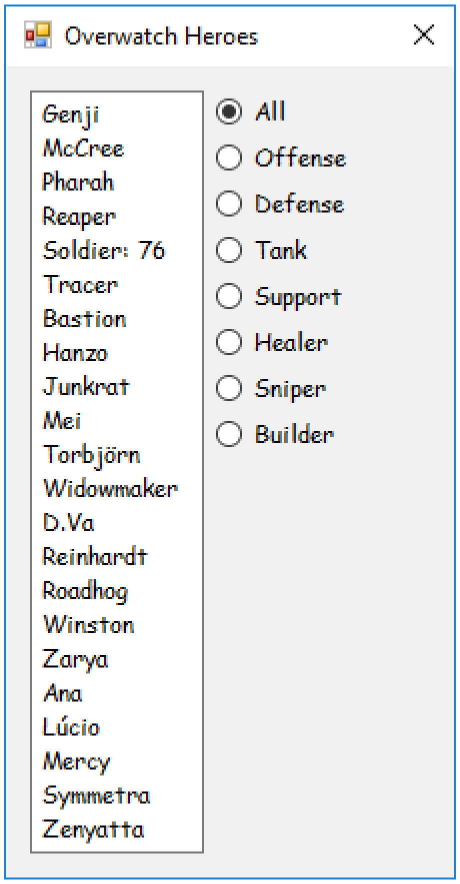
9/1/16

Iterator Pattern

The purpose of this paper it to, as Professor Retterer puts it, "do the iterator pattern. The iterator pattern's purpose, is to, well, iterate through an aggregate object (such as an array, List, or Tree object) without returning to a previous element of the aggregate and without revealing the aggregate's structure. The UML diagram, courtesy of oodesign.com, is:



To “do” this pattern, I created the program “OverwatchHeroes.” This program displays (as of this program and paper’s creation) all of Overwatch’s playable Heroes. When a radio button is pressed, it selects only the heroes that are of that class/subclass.

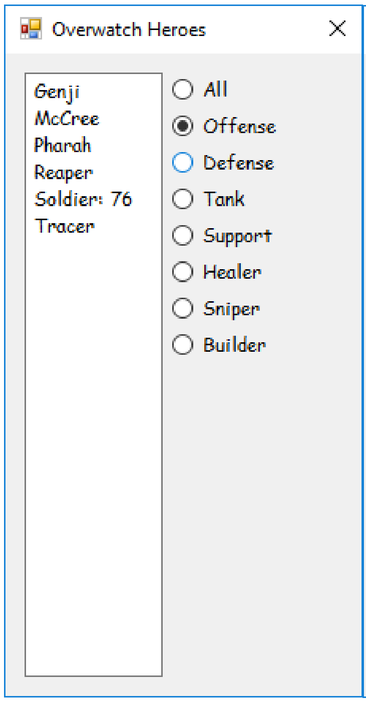


Genji
McCree
Pharah
Reaper
Soldier: 76
Tracer
Bastion
Hanzo
Junkrat
Mei
Torbjörn
Widowmaker
D.Va
Reinhardt
Roadhog
Winston
Zarya
Ana
Lúcio
Mercy
Symmetra
Zenyatta

[✓ All] Offense Defense Tank Support Healer Sniper Builder

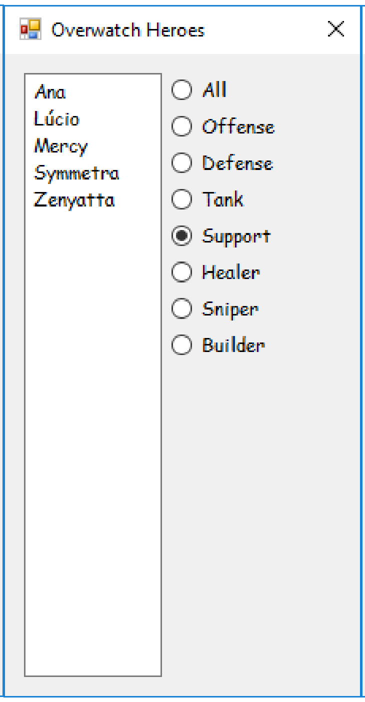
This is accomplished with the following classes:

- OverwatchHeroesForm
- Iterator
- ConcreteIterator
- *Aggregate*
- ConcreteAggregate
- Hero



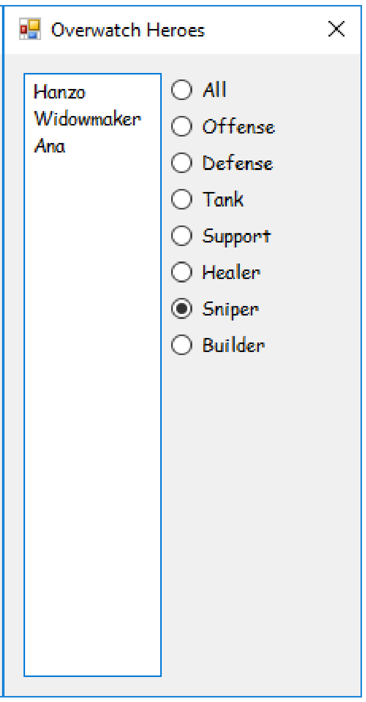
Genji
McCree
Pharah
Reaper
Soldier: 76
Tracer

All [✓ Offense] Defense Tank Support Healer Sniper Builder



Ana
Lúcio
Mercy
Symmetra
Zenyatta

All Offense Defense Tank [✓ Support] Healer Sniper Builder



Hanzo
Widowmaker
Ana

All Offense Defense Tank Support Healer [✓ Sniper] Builder

Iterator and Aggregate are abstract classes that serve as the parent classes of ConcreteIterator and ConcreteAggregate, respectively.

```
public abstract class Iterator
    {
        public abstract object first();
        public abstract object next();
        public abstract bool isDone();
        public abstract object currentItem();
    }
```

```
public abstract class Aggregate
    {
        public abstract Iterator createIterator();
    }
```

The Hero class holds 4 strings: the hero's name, his/her class, and up to 2 subclasses.

```
public class Hero
    {
        public string name;
        public string heroClass;
        public string subclass1;
        public string subclass2;
    }
```

ConcreteAggregate contains an aggregate (in this case, a List) that holds several Hero classes, and a method that creates a ConcreteIterator that refers to this aggregate:

```
public class ConcreteAggregate : Aggregate
    {
        public List<Hero> heroList = new List<Hero>()
        {
            new Hero(){name = "Genji",          heroClass = "Offense",  subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "McCree",         heroClass = "Offense",  subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Pharah",         heroClass = "Offense",  subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Reaper",         heroClass = "Offense",  subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Soldier: 76",    heroClass = "Offense",  subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Tracer",         heroClass = "Offense",  subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Bastion",        heroClass = "Defense",  subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Hanzo",          heroClass = "Defense",  subclass1 = "Sniper",   subclass2 = "none"},
            new Hero(){name = "Junkrat",        heroClass = "Defense",  subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Mei",            heroClass = "Defense",  subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Torbj\u00f6rn",  heroClass = "Defense",  subclass1 = "Builder",  subclass2 = "none"},
            new Hero(){name = "Widowmaker",     heroClass = "Defense",  subclass1 = "Sniper",   subclass2 = "none"},
            new Hero(){name = "D.Va",           heroClass = "Tank",     subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Reinhardt",      heroClass = "Tank",     subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Roadhog",        heroClass = "Tank",     subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Winston",        heroClass = "Tank",     subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Zarya",          heroClass = "Tank",     subclass1 = "none",     subclass2 = "none"},
            new Hero(){name = "Ana",            heroClass = "Support",  subclass1 = "Sniper",   subclass2 = "Healer"},
            new Hero(){name = "L\u00FAcio",     heroClass = "Support",  subclass1 = "Healer",   subclass2 = "none"},
            new Hero(){name = "Mercy",          heroClass = "Support",  subclass1 = "Healer",   subclass2 = "none"},
            new Hero(){name = "Symmetra",       heroClass = "Support",  subclass1 = "Builder",  subclass2 = "none"},
            new Hero(){name = "Zenyatta",       heroClass = "Support",  subclass1 = "Healer",   subclass2 = "none"},
        };

        public override Iterator createIterator()
        {
            return new ConcreteIterator(this);
        }
    }
```

(The weird strings of characters are Unicode for ö and ú respectively)

ConcreteIterator and OverwatchHeroesForm will be explained when what happens is explained.

What Happens

The program uses the iterator pattern as soon as the program starts to display the heroes in the textbox. The first hero in the aggregate is put into the textbox first, and then a while loop iterates through the aggregate and enters the rest of the heroes in.

```
public partial class OverwatchHeroesForm : Form
    {
        string keyword;

        ConcreteAggregate heroes = new ConcreteAggregate();

        ConcreteIterator iter;

        public OverwatchHeroesForm()
        {
            InitializeComponent();

            iter = (ConcreteIterator) heroes.createIterator();

            heroBox.Text = (string) iter.first();

            while (!iter.isDone())
            {
                heroBox.Text += iter.next();
            }
        }
```

```
public class ConcreteIterator : Iterator
    {
        private ConcreteAggregate agg;
        private int element;

        public ConcreteIterator(ConcreteAggregate aggregate)
        {
            agg = aggregate;
        }

        public override object first()
        {
            element = 0;
            return currentItem();
        }

        public override object next()
        {
            element++;
            return currentItem();
        }

        public override bool isDone()
        {
            return (element == agg.heroList.Count - 1);
        }

        public override object currentItem()
        {
            return (agg.heroList[element].name + "\r\n");
        }
    }
```

When a radio button is pressed, it changes the keyword variable shown on the last page into a specific string that is plugged into a "replaceText" function in the same form. This function uses the keyword to check which heroes have that string as a class or subclass. This check is done in the "hasKey" function, which is in ConcreteIterator. replaceText overwrites the textbox with the heroes that meet this criterion.

```
private void offenseRadio_CheckedChanged(object sender, EventArgs e)
        {
            keyword = "Offense";
            replaceText(keyword);
        }
```

```
private void replaceText(string key)
        {
            heroBox.Text = "";
            iter.first();
            if (iter.hasKey(key))
            {
                heroBox.Text = (string) iter.first();
            }

            while (!iter.isDone())
            {
                iter.next();
                if (iter.hasKey(key))
                {
                    heroBox.Text += iter.currentItem();
                }
            }
        }
```

```
public bool hasKey(string key)
        {
            return (agg.heroList[element].heroClass == key
                 || agg.heroList[element].subclass1 == key
                 || agg.heroList[element].subclass2 == key);
        }
```

Reflection

This assignment was a great one for getting to know how this course is gonna work; I had to learn how to interpret UML diagrams, how to use C# more effectively, and, of course, the Iterator Pattern. I feel like I made this paper a bit too long in my attempts to stay organized, but that shall be determined by the feedback I receive. The learning experience is constant, after all, and this is the first assignment.

I'd like to thank the Lounge of Legends, Comic Sans MS, and the Cytus soundtrack for the completion of this assignment.