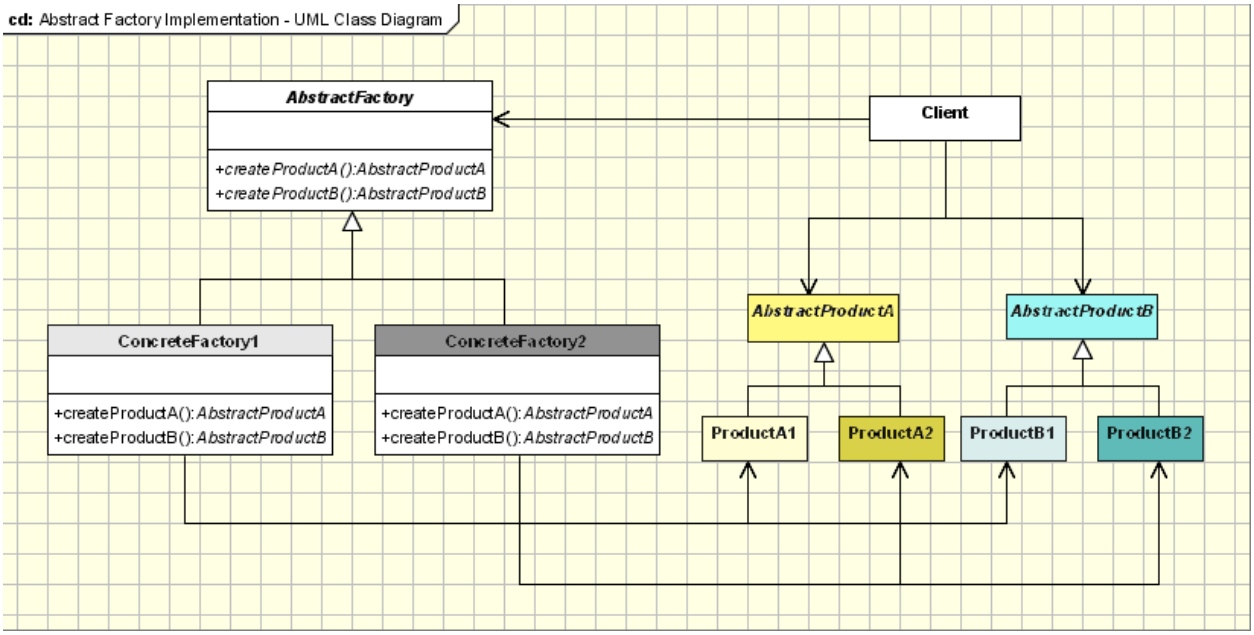
Jack Raney

Design Patterns

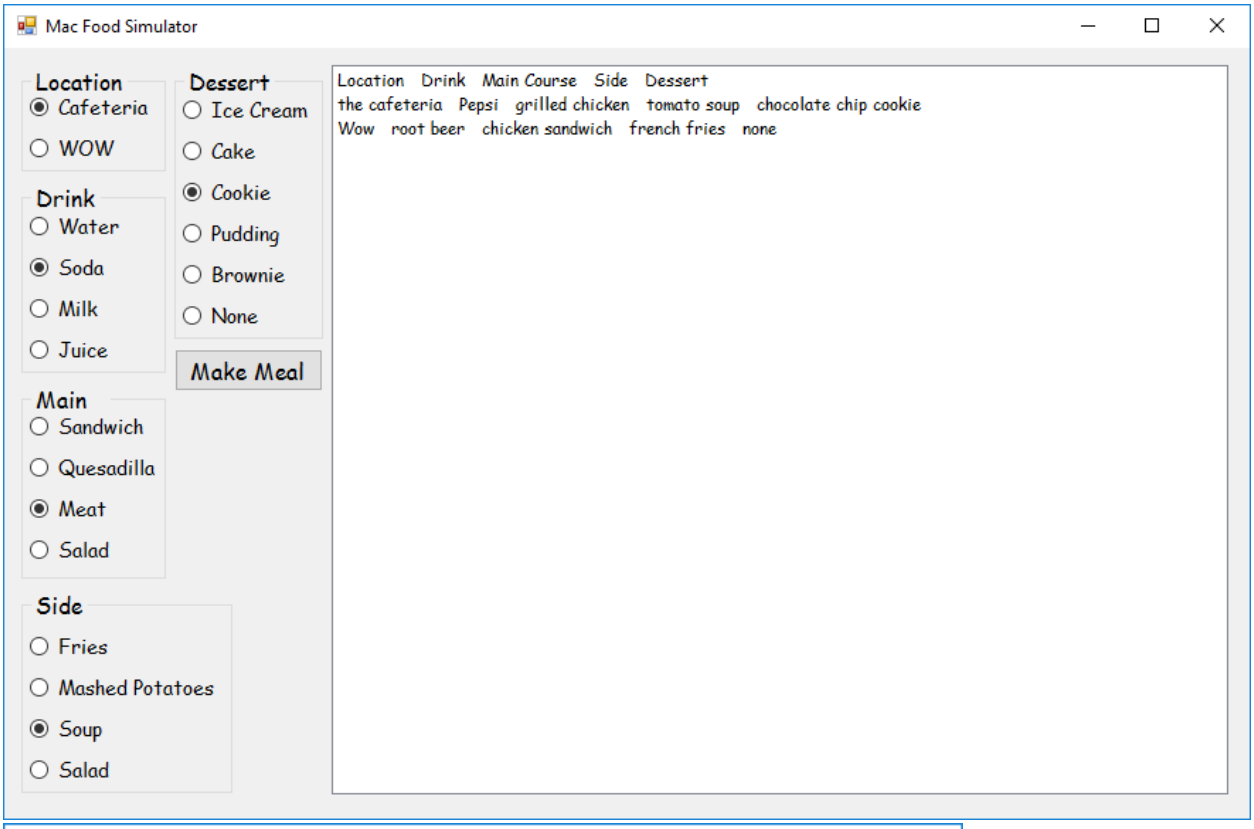
9/27/2016

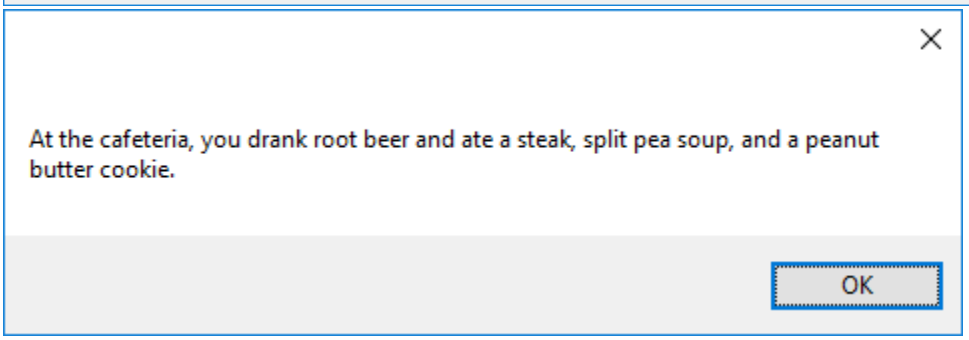
The Abstract Factory Pattern

This paper is about the Abstract Factory Pattern, a programming design pattern that, according to oodesign.com, is intended to "[offer] the interface for creating a family of related objects, without explicitly specifying their classes". This means that an abstract factory class (that other factories extend) details a method of creating several product objects; the objects differ depending on what factory instantiated it. The UML diagram for this pattern is shown below:



For the Abstract Factory Pattern, I created the application MacFood.exe, which simulates the options for meal swipes in the Macintosh building at Ohio Northern:





The "AbstractFactory" for this application is the Cook class. This class creates abstract methods for instantiating four different "ConcreteProducts" derived from the Food class.

```
public abstract class Cook  //This is the Abstract Facotry
    {
        public abstract Food pourDrink(int choice);
        public abstract Food cookMain(int choice);
        public abstract Food cookSide(int choice);
        public abstract Food cookDessert(int choice);
    }
```

The Food class is an "AbstractProduct" class in the UML diagram. It contains a protected Random object for random selection utilized by the child objects of Food. Food also contains an integer that changes based on what options the user chooses and a Boolean that is based off of whether the food is coming from the cafeteria of Wow. The method getFood will return a string that tells the user what food they received from that category (class).

```
public abstract class Food  //This is an Abstract Product
    {
        protected Random rand = new Random();
        protected int choice;
        protected bool isWow;

        public abstract string getFood();
    }
```

All four child objects (Drink, MainCourse, Side, and Dessert) have the following structure to their code:

1. String arrays that the code randomly selects from for meal options
2. A constructor to instantiate the object with the selected choice and, in MainCourse's case, location
3. An override for getFood that returns a random food via a switch-case statement
4. An override of ToString that returns the result of getFood

```
public class Drink : Food   //This is a Product
    {
        private string[] SODA =
            {"Pepsi", "Sprite", "Dr. Pepper", "root beer"};
        private string[] MILK =
            {"white", "chocolate", "skim"};
        private string[] JUICE =
            {"apple", "orange", "pineapple", "grape"};

        public Drink(int theChoice)
        {
            choice = theChoice;
        }

        private override string getFood()
        {
            switch (choice)
            {
                case 1:  return "water";
                case 2:  return SODA[rand.Next() % SODA.Length];
                case 3:  return MILK[rand.Next() % MILK.Length] + " milk";
                default: return JUICE[rand.Next() % JUICE.Length] + " juice";
            }
        }

        public override string ToString()
        {
            return getFood();
        }
    }
```

The two "factory" classes that extend Cook, CafeCook and WowCook, have one simple difference: MainCourse's Boolean in its constructor is made false for CafeCook and true for WowCook (The Boolean is for whether or not the food is from Wow).

```
public class CafeCook : Cook    //This is a factory
    {
        public override Food pourDrink(int choice)
        {
            return new Drink(choice);
        }

        public override Food cookMain(int choice)
        {
            return new MainCourse(choice, false);
        }

        public override Food cookSide(int choice)
        {
            return new Side(choice);
        }

        public override Food cookDessert(int choice)
        {
            return new Dessert(choice);
        }
    }
```

The Form for this application instantiates a Boolean for whether or not the location is Wow (a common variable in this program), four integers that are changed by the radio buttons to be passed into their respective Food objects' getFood methods, a CafeCook, and a WowCook. On the program's initialization, the first radio buttons of each group are checked and the values that represent them are placed into the getFood integers. A header string is also added to the MealList (the ListBox) that describes what each segment of the items added later correlate with.

```
public partial class Form1 : Form
    {
        private bool isWow;
        private int drink;
        private int main;
        private int side;
        private int dessert;

        CafeCook cafe = new CafeCook();
        WowCook wow = new WowCook();

        public Form1()
        {
            InitializeComponent();

            isWow = false;
            drink = 1;
            main = 1;
            side = 1;
            dessert = 6;

            cafeRadio.Checked = true;
            waterRadio.Checked = true;
            sandwichRadio.Checked = true;
            friesRadio.Checked = true;
            noneRadio.Checked = true;

            mealList.Items.Add(
                "Location   Drink   Main Course   Side   Dessert");
        }
```

The Location radio buttons (cafeRadio and wowRadio) set the isWow Boolean to their respective values. wowRadio also disables radio buttons that don't apply to it, like milkRadio and cakeRadio (you can't get milk or cake from Wow). cafeRadio enables them when selected.

```
private void cafeRadio_CheckedChanged(object sender, EventArgs e)
        {
            isWow = false;
            milkRadio.Enabled = true;
            juiceRadio.Enabled = true;
            mashRadio.Enabled = true;
            soupRadio.Enabled = true;
            if (!saladMainRadio.Checked)
                saladSideRadio.Enabled = true;
            iceCreamRadio.Enabled = true;
            cakeRadio.Enabled = true;
            cookieRadio.Enabled = true;
            puddingRadio.Enabled = true;
            brownieRadio.Enabled = true;
        }
```

Each radio button simply changes the value of the getFood integer that correlates with its food category; for example, the water, soda, milk, and juice radio buttons change the drink integer to 1 through 4, depending on which button is pressed.

```
private void waterRadio_CheckedChanged(object sender, EventArgs e)
        {
            drink = 1;
        }

        private void sodaRadio_CheckedChanged(object sender,
EventArgs e)
        {
            drink = 2;
        }

        private void milkRadio_CheckedChanged(object sender,
EventArgs e)
        {
            drink = 3;
        }

        private void juiceRadio_CheckedChanged(object sender,
EventArgs e)
        {
            drink = 4;
        }
```

mealButton, the button labeled "Make Meal" in the Form, starts by declaring strings for the location, drink, main course, side, and dessert. Next, the location is set, and depending on whether or not isWow is true, either the WowCook object or the CafeCook object is used to create the Food objects and stores their ToString results (received from their getFood methods) into the strings. Then, the meal is added to mealList by concatenating the strings together with space in-between. Finally, a MessageBox pops up that tells the user in a complete sentence what meal they received.

```
private void mealButton_Click(object sender, EventArgs e)
        {
            string location;
            string theDrink;
            string theMain;
            string theSide;
            string theDessert;

            if (isWow)
            {
                location = "Wow";
                theDrink  = wow.pourDrink(drink).ToString();
                theMain = wow.cookMain(main).ToString();
                theSide = wow.cookSide(side).ToString();
                theDessert = wow.cookDessert(dessert).ToString();
            }
            else
            {
                location = "the cafeteria";
                theDrink  = cafe.pourDrink(drink).ToString();
                theMain = cafe.cookMain(main).ToString();
                theSide = cafe.cookSide(side).ToString();
                theDessert = cafe.cookDessert(dessert).ToString();
            }

            mealList.Items.Add(location + "   " + theDrink + "   " + theMain
                + "   " + theSide + "   " + theDessert);
            if(!theDessert.Equals("none"))
                MessageBox.Show("At " + location + ", you drank " + theDrink
                    + " and ate a " + theMain + ", " + theSide + ", and a "
                    + theDessert + ".");
            else
                MessageBox.Show("At " + location + ", you drank " + theDrink
                    + " and ate a " + theMain + " and " + theSide + ".");
        }
```

Reflection

This program was not incredibly difficult to make, but it contained several classes and Form objects, which made its creation time-consuming. I was also able to understand the Abstract Factory Pattern better than I did the Factory Method Pattern, fortunately for the program and the paper.