Debra Cooperman
cs5010 Assignment 4: Design Patterns, Recursion, and Halloween

For this assignment, I implemented both the visitor pattern and the interpreter pattern.

HalloweenNeighborhoodTraversal: Visitor Pattern

For the visitor pattern program, have 14 classes (including the main class, **HalloweenNeighborhoodTraversal**) and two interfaces, **Visitor** and **Visitable**.

The program uses the abstract classes **NeighborhoodMemberLeaf** and **NeighborhoodMember** to create the structure of the neighborhood. **NeighborhoodMemberLeaf** has a name, a parent, and a visited state, and implements the **Visitable** interface (see below). **NeighborhoodMember** extends **NeighborhoodMemberLeaf** with the difference being **NeighborhoodMember** has children.

Classes that extend these abstract classes are used to construct a tree that has a **Neighborhood** at the root, with **Households** (abstract class) as children, and **NeighborhoodCandy**s, which are the children of the **Household**s. Four different classes extend **Household**: **Mansion**, **Duplex**, **DetachedHouse**, and **Townhome**.

The **Visitable** interface contains accept methods for classes that implement the **Visitor** specifically the two **Visitor** classes the program uses:

- **PopulateNeighborhoodVisitor**: populates an empty **Neighborhood** with **Households** and **NeighborhoodCandy**s
- **NeighborhoodCandyTraversalVisitor**: traverses the **Neighborhood** to find matching candies and determine whether there exists a traversal path through the **Neighborhood** that contains all the desired candies.

The **CandyParser** class takes the contents of the "DreamCandyX" file that contains the list of desired candies, validates the filename, and extracts each desired candy name and size to be compared against the candies in the **Neighborhood**. The **CandyCategories** class contains a set of enum classes that contain the candy name and size categories for each of the **Household**s: **Mansion, Duplex, Detached House, and Townhome**, with methods for generating the candy list for each specific household type.

The program starts by first creating a new empty **Neighborhood**. The populateNeighborhood method creates a list of households, sets that list as the household list for the Neighborhood, then creates a **PopulateNeighborhoodVisitor**, which then visits the Neighborhood. When **PopulateNeighborhoodVisitor** visits the

Neighborhood, there is a separate visit method for each type of Household. When a Mansion is visited, PopulateNeighborhoodVisitor populates it with **NeighborhoodCandy**s that are specific to a **Mansion**, same behavior with **Duplex**, **Townhome**, etc.

Once the Neighborhood is populated, the program goes through each file line by line and uses the **CandyParser** to validate each file name and extract a list of candy names and sizes for each file. For each candy name and size, the NeighborhoodCandyTraversalVisitor's *desiredCandyName* and *desiredCandySize* attributes are set with those values, and then **NeighborhoodCandyTraversalVisitor** visits the **Neighborhood**, visiting each Household and **NeighborhoodCandy** within each Household. The special visit method for **NeighborhoodCandy** will attempt to match the candy name and size from the DreamCandyX file with the name and size of the **NeighborhoodCandy**.

If the NeighborhoodCandyTraversalVisitor traverses the entire Neighborhood without finding a match, the program prints out a message that no traversal path can be found, and then goes on to the next file if there is one. Otherwise if a match is found for that candy, then the candy name, candy size, and household type are written to the candyPath list that the **NeighborhoodCandyTraversalVisitor** maintains. Then *desiredCandyName* and *desiredCandySize* are set in the **NeighborhoodCandyTraversalVisitor** with the next candy name and size from the DreamCandyX, and the **Neighborhood** is visited again. If the **NeighborhoodCandyTraversalVisitor** is able to find a match for all the candies in a given DreamCandyX file, then the candyPath list is iterated through and the contents written to the DreamTraversalX file.

Advantages/Disadvantages of Visitor Pattern for HalloweenNeighborhoodTraversal

Advantages:

- Easily able to traverse the Neighborhood, its children (Households) and its children's children by recursing visit methods.
- By creating classes for specific Household types (Mansion, DetachedHouse, Duplex, Townhome) whose visit methods were different, if I wanted to change the types of Candies, for example that each household types had, or to add some behavior around when the candies were given out for each household type, I could change all that without having to change the interface, and avoiding lengthy "if/else" statements.

Disadvantages:

- **Proliferation of classes:** For the magic of Visitor pattern to work, by virtue of being a specific class (like Mansion vs. DetachedHouse), special visit behavior is inferred. But in order for that model to work, you have to create many

different classes depending on the type of behavior you want. For example, if I had created a separate class for each type of candy, it would have been easy to apply the Visitor pattern to determine whether, for example "TwixCandy" existed in the Neighborhood, because all I would have to do is visit it and then by virtue of the special visit method I would know it was a Twix. But then I would need separate classes for KitKat, Snickers, etc.

- **Having to maintain external state in the Visitor classes.** The way I got around the class proliferation (although I still had a lot) was to create attributes in the NeighborhoodCandyTraversalVisitor to represent the name and size of each desired candy. It felt odd because the NeighborhoodCandyTraversalVisitor doesn't "own" that state, it gets updated every time a new candy is read from the file, and NeighborhoodCandyTraversalVisitor uses that information to compare against the candies in the Neighborhood.

HalloweenNeighborhoodTraversal2: Interpreter Pattern

Next I implemented the same program using the Interpreter Pattern. The CandyParser and HalloweenNeighborhoodTraversal(2) program itself were roughly the same—take candy names and sizes from the file, evaluate each against information about the candies that exist in the Neighborhood, write to a file if a traversal path is found, etc. But the way that each desired candy name and file are evaluated is very different. This program has only 7 classes and 1 interface, called **IExpression**. **IExpression** has one method, *interpret,* that takes a String context, checks that string against some kind of expression (e.g., "if Twix and super size or Kit Kat and fun size, etc., the candy belongs to a mansion), and returns true if the expression evaluates to true, and false if it evaluates to false. There are three classes that implement IExpression:

- **TerminalExpression** has one attribute: a String *candyInfo* that represents the candy name and size. **TerminalExpression**'s *interpret* method takes a String *context* and simply determines whether the context String contains candyInfo, returning true if it does and false if it doesn't.
- **AndExp** has two attributes: **IExpression** *expr1* and **IExpression** *expr2*. It's *interpret* method simply takes a String context and evaluates whether both expr1's and expr2's *interpret* methods evaluate to true when passed the context String.
- **OrExp** also has **IExpression** *expr1* and **IExpression** *expr2*, except that it evaluates whether one or the other or both interpret methods evaluate to true.

The **NeighborhoodCandyExpression** class does most of the work to evaluate whether a given candy exists in the Neighborhood. It essentially puts together And, Or and Terminal Expressions to come to a conclusion about a given candy. For example, the getMansionSuperSizeCandyExpression() evaluates to true when there

is a candy that is “super size” and (twix or snickers or mars). The getMansionCandyExpression() evaluates to true when the candy satisfied either the getMansionFunSizeCandyExpression() or the getMansionKingSizeCandyExpression() or the getMansionSuperSizeCandyExpression(). And when you combine all the expressions for each type of household (Mansion, DetachedHouse, Duplex, Townhome), it adds up to getNeighborhoodCandyExpression, i.e., whether the candy exists in the Neighborhood.

Advantages/Disadvantages of Interpreter Pattern for HalloweenNeighborhoodTraversal

Advantages:

- Requires far fewer specialized classes, and far fewer lines of code to determine whether a candy exists in the Neighborhood and what household type it exists in. Once the evaluation logic is known, easy to build this program in interpreter pattern.
- Using **IExpression** interface has same benefits as **Visitor** interface in that the interpret method can mean different things for different classes that implement or use **IExpression**, so a bunch of the logic can be changed without effecting the interface implementers.
- The top-level methods are very easy to understand (e.g, getNeighborhoodCandyExpression() just evaluates getMansionCandyExpression() or getDuplexCandyExpression() or getTownhomeCandyExpression() or getDetachedCandyExpression()).

Disadvantages:

- **Proliferation of expressions:** Each high-level expression is made up of multiple smaller expressions, that are in turn made up of even smaller expressions. The logic can get very complicated and difficult to accurately test. If you’ve made a mistake in a lower-level expression (substituting and and for an or, for example), it can be hard to figure out how that affects the higher-level expressions.