The Composite Design Pattern

Object Oriented Software Engineering
Lecture 9

Dr. Graham McDonald
graham.mcdonald@glasgow.ac.uk

Learning outcomes

- Understand different problems that can be addressed using the composite design pattern.
- Understand how to apply the composite design pattern.
- Articulate the structure of the design pattern.

Introduction

Problem: Products and Boxes

Introduction

Problem: Products and Boxes

Imagine that you have two types of objects: Products and Boxes.

A Box can contain several Products as well as a number of smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.

Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes.

How would you **determine the total price of such an order**?

Introduction

Solution: Products and Boxes

Approach 1:

Unwrap all the boxes, go over all the products and then calculate the total.

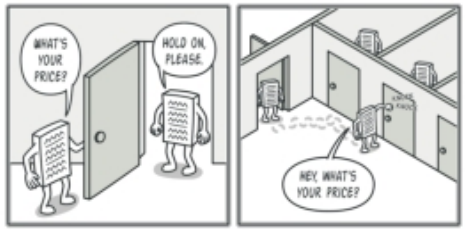
Are there any drawbacks/challenges to this implementation approach?

Introduction

Solution: Products and Boxes

Approach 2: Work with Products and Boxes through a common interface which declares a method for calculating the total price.

How would this method work?



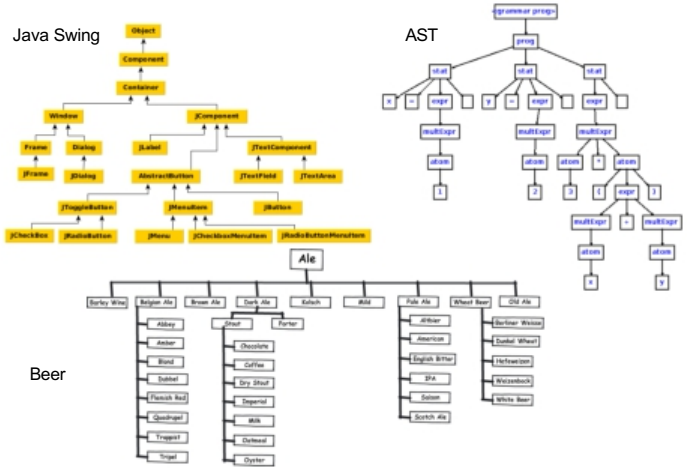
Works by running a behavior **recursively** over all **components** of an **object tree**

Introduction

Composite Design Pattern:

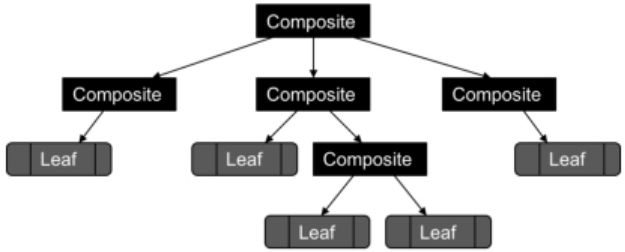
A **structural** design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Dealing with Structures and Hierarchies



The Problem

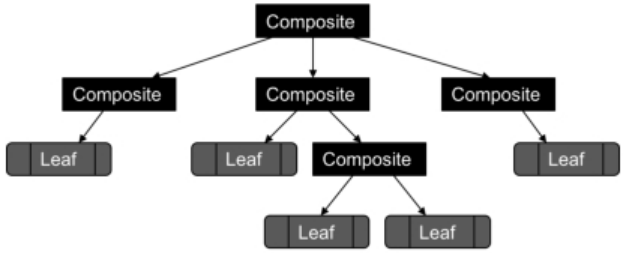
Composite Structure



Application needs to manipulate a hierarchical collection of **'primitive'** and **'composite'** objects.

The Problem

Composite Structure



Processing of a primitive object is handled one way, and processing of a composite object is handled differently.

Having to query the "type" of each object before attempting to process it is **not desirable**.

Example: The Computer File System

Directory/File Object Structure

A file system is a Folder, which is a form of *Entry*:

An *Entry* is either:

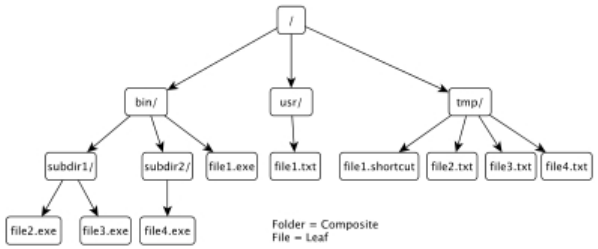
1. a *Textfile*, which holds a sequence of chars
2. a *Binaryfile*, which holds a sequence of ints
3. a *Shortcut*, which is a handle or pathname to an Entry
4. or a ***Folder***, which holds a sequence of zero or more *Entries*

Example: The Computer File System

Directory/File Object Structure

A file system consists of a start folder, i.e., root "/".

Inside the folder are "entries": textfiles, binary files, links (shortcuts/pathnames) to entries, and more folders.



Example: The Computer File System

Directory/File Object Structure

Create the tree leaves

```
public class BinaryFile {
    String name;
    public static final String TYPE = ".exe";

    public BinaryFile(String name){
          this.name = name;
    }

    public void ls() {
       System.out.println(CompositeDemo.compositeBuilder + name+TYPE);
    }
}
```

Example: The Computer File System

Directory/File Object Structure

Create the tree leaves

```
public class TextFile {
    String name;
    public static final String TYPE = ".txt";

    public TextFile(String name){
        this.name = name;
    }

    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name+TYPE);
    }
}
```

Example: The Computer File System

Directory/File Object Structure

Create the tree leaves

```
public class Shortcut {
    public Object link;
    String name;
    public static final String TYPE = ".shortcut";

    public Shortcut(String name, Object link) {
        this.name = name;
        this.link = link;
    }

    public void ls() {
        String linkType = "";
        if(link instanceof TextFile)
            linkType = TextFile.TYPE;
        else if(link instanceof BinaryFile)
            linkType = BinaryFile.TYPE;
        else if(link instanceof Folder)
            linkType = Folder.TYPE;
        //...
        System.out.println(CompositeDemo.compositeBuilder + name+linkType+TYPE);
    }
}
```

Example: The Computer File System

Directory/File Object Structure

Create the tree composite

```
public class Folder {
    String name;
    public static final String TYPE = "/";
    private ArrayList<Object> includedFiles = new ArrayList<Object>();

    public Folder(String name) {
        this.name = name;
    }
    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name+TYPE);
        CompositeDemo.compositeBuilder.append("   ");
        for (Object obj : includedFiles) {
            // Recover the type of this object
            String name = obj.getClass().getSimpleName();
            if (name.equals("BinaryFile")) {
                ((BinaryFile)obj).ls();
            }
            else if (name.equals("TextFile")) {
                ((TextFile)obj).ls();
            }
            else if (name.equals("Shortcut")) {
                ((Shortcut)obj).ls();
            }
            else if (name.equals("Folder")) {
                ((Folder)obj).ls();
            }
        }
        CompositeDemo.compositeBuilder.setLength(CompositeDemo.compositeBuilder.length() - 3);
    }
}
```

Example: The Computer File System

Directory/File Object Structure

Create the Client

```
public class CompositeDemo {

    public static StringBuilder compositeBuilder = new StringBuilder();

    public static void main(String[] args) {
        Folder root = new Folder("root");
        Folder bin = new Folder("bin");
        Folder usr = new Folder("usr");
        Folder tmp = new Folder("tmp");
        root.add(bin);
        root.add(usr);
        root.add(tmp);
        //...

        root.ls();
    }
}
```

The Composite Design Pattern

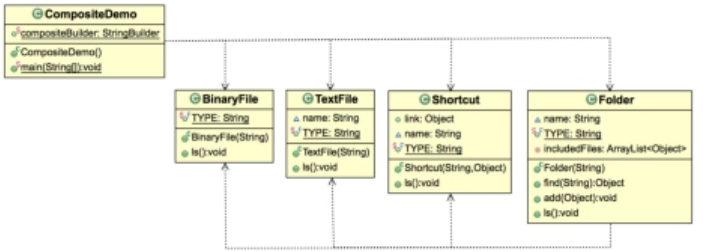
Object Oriented Software Engineering
Lecture 9: Part 2

Dr. Graham McDonald
graham.mcdonald@glasgow.ac.uk

The Problem

Each time, you have to query the file type before calling ls() function.

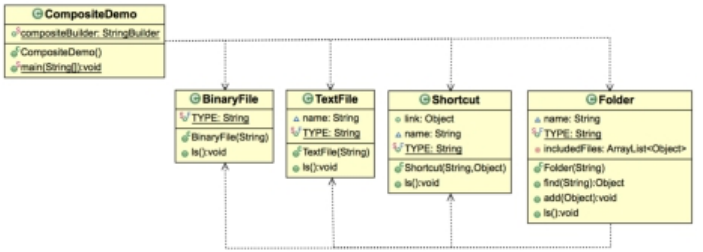
Each time you add a new component, you have to also update Folder, Shortcut and CompositeDemo classes.



The Problem : Naïve Solution

Each time, you have to query the file type before calling ls() function.

Each time you add a new component, you have to also update Folder, Shortcut and CompositeDemo classes.



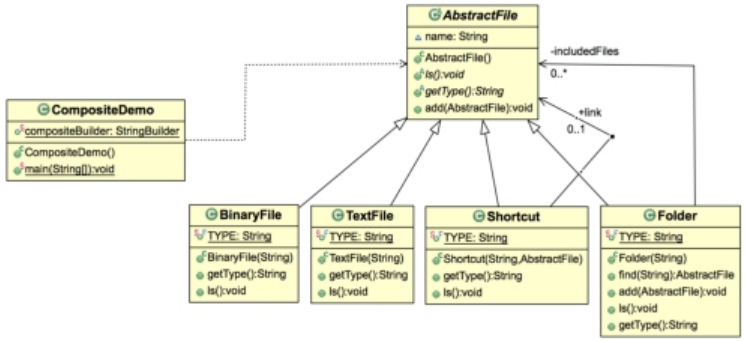
The Solution

How do we make sure that **Leaf** nodes and **Composite** nodes can be handled **uniformly**?

The Solution

How do we make sure that **Leaf** nodes and **Composite** nodes can be handled **uniformly**?

- Derive them from the same abstract base class.



The Solution

Create an abstract class

```
public abstract class AbstractFile {
    String name;

    public abstract void ls();
    public abstract String getType();
    public void add(AbstractFile f) {}
}
```

The Solution

Create the tree leaves which extends the abstract class

```
public class BinaryFile extends AbstractFile{
    public static final String TYPE = ".exe";

    public BinaryFile(String name){
          this.name = name;
    }

    @Override
    public void ls() {
       System.out.println(CompositeDemo.compositeBuilder + name+TYPE);
    }
}
```

The Solution

Create the tree leaves which extends the abstract class

```
public class TextFile extends AbstractFile{
    public static final String TYPE = ".txt";

    public TextFile(String name){
      this.name = name;
    }

    @Override
    public void ls() {
       System.out.println(CompositeDemo.compositeBuilder + name+TYPE);
    }
}
```

The Solution

Create the tree leaves which extends the abstract class

```
public class Shortcut extends AbstractFile{
public AbstractFile link;

public static final String TYPE = ".shortcut";

    public Shortcut(String name, AbstractFile link) {
        this.name = name;
        this.link = link;
    }

    @Override
    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name+link.getType()+TYPE);
    }
}
```

The Solution

Create a tree composite which extends the abstract class

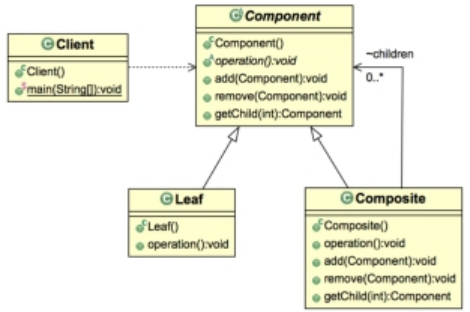
```
public class Folder extends AbstractFile{
    public static final String TYPE = "/";

    private ArrayList<AbstractFile> includedFiles = new ArrayList<AbstractFile>();

    public Folder(String name) {
        this.name = name;
    }

    @Override
    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name+TYPE);
        CompositeDemo.compositeBuilder.append("    ");
        for (AbstractFile as : includedFiles) {
            as.ls();
        }
        CompositeDemo.compositeBuilder.setLength(CompositeDemo.compositeBuilder.length() - 3);
    }
}
```

The structure of Composite Design Pattern



The structure of Composite Design Pattern

Component

- Declare the interface for objects in the composition.
- Implements default behavior.
- Declares methods for accessing and managing its child components.
- Optionally, you can define methods for accessing a component's parent in the recursive structure.

The structure of Composite Design Pattern

Leaf

- represents leaf objects in the composition.
- define behavior for primitive objects.

Composite

- defines behavior for components having children.
- stores child components.
- implements child-related operations in the component interface.

Client

- manipulates objects in the composition through the Component interface.

The structure of Composite Design Pattern

Consequences

Defines class hierarchies consisting of **primitive** objects and **composite** objects.

Makes the client simple.

Makes it easier to add new kinds of components.

(**disadvantage**) can make your design overly general.

Summary

The Composite design pattern is a **structural pattern.**

Structural patterns are concerned with **how classes and objects are composed to form larger structures**.

The Composite design pattern facilitates to **compose objects into tree structures to represent part-whole hierarchies.**