The Visitor Design Pattern

Object Oriented Software Engineering
Lecture 10

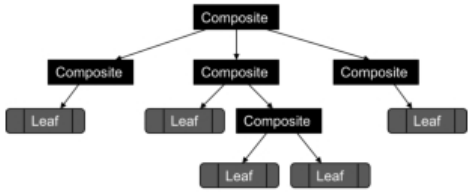
Dr. Graham McDonald
graham.mcdonald@glasgow.ac.uk

Learning outcomes

- Distinguish between single and double dispatch
- The structure of Visitors design pattern
- How the visitors pattern builds on the composite design pattern.
- Advantages, disadvantages and consequence of using visitors design pattern

Recap: Dealing with Structures and Hierarchies

Composite Structure design problem

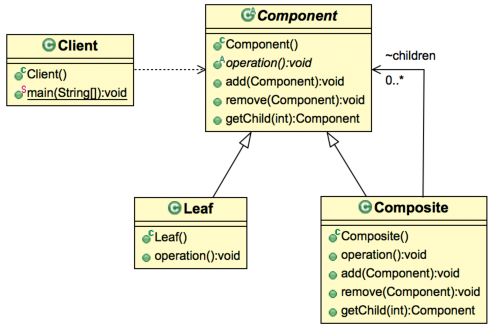


- Often times we need to manipulate a hierarchical collection of primitive and composite objects.
    - Processing of a primitive object is handled one way, and processing of a composite object is handled differently.
    - But having to query the "type" of each object before attempting to process it is not desirable.

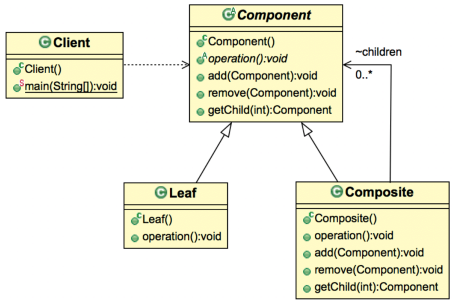
Recap: Dealing with Structures and Hierarchies

Composite Structure design solution

- How do we make sure that Leaf nodes and Composite nodes can be handled uniformly?
    - Derive both the Composite and Leaf from the same abstract base class.

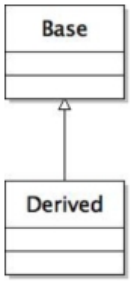


Recap: Dealing with Structures and Hierarchies



- So, abstractions are necessary in designing hierarchical composite structures.
- But each abstract type must eventually be resolved to a concrete type in order to do actual work.

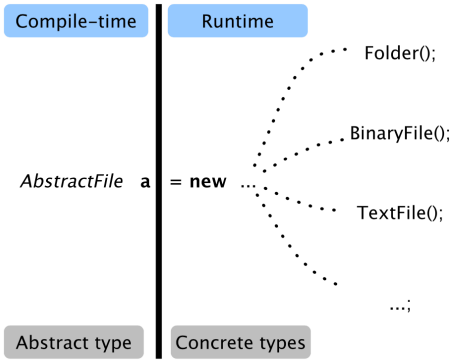
Recap: Dealing with Structures and Hierarchies



Ensure that when substituting a base class for derived class, that the Likov's Substitution Principle is adhered to.

Dealing with Structures and Hierarchies

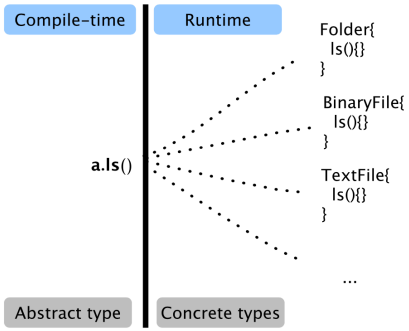
Composite:
Single-dispatch



The field AbstractFile **a** at compile time can resolve to any of the concrete types at runtime.

Recap: Dealing with Structures and Hierarchies

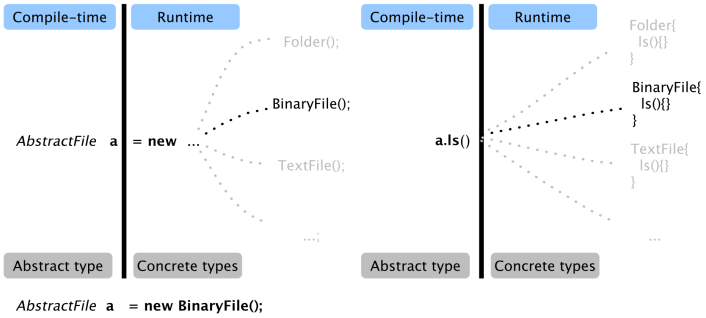
Composite:
Single-dispatch



At runtime, the program will dynamically dispatch to an object's underlying type and call that object's ls() method.

Recap: Dealing with Structures and Hierarchies

Composite: Single-dispatch



So the actual operation getting executed depends on name of the method you are calling and the type of the concrete component.

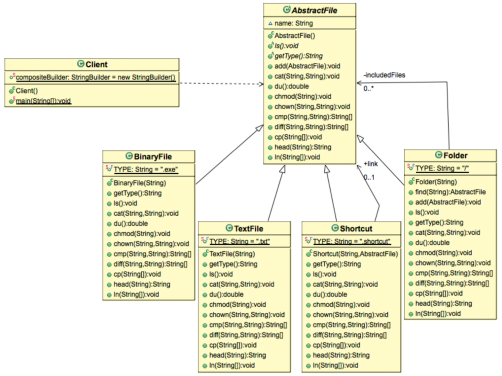
Visitors Design Pattern

Example: The Computer File System

- Assume the Client want to do arbitrarily many operations on file system objects
  - cat,
    du,
    chmod,
    chown,
    cmp,
    diff,
    cp,
    head,
    ln,
    find
    etc ...

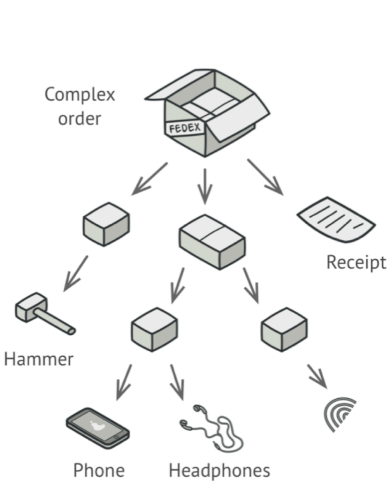
Problem

- By simply adding every new operation to the AbstractFile interface and its concrete classes:



The problem is that you end up treating AbstractFile interface as a dumping ground.

Auditing Products and Boxes



Context

- Assume you want to design operation(s) across a heterogeneous composition of objects.
- You also want the operation(s) to be defined without changing the class of any of the objects in the collection because:
    - You want to avoid "polluting" the node classes with these operations.
    - You don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Solution: Visitors Design Pattern

Intent:

- Externalise and centralise operations on an object structure so that they can vary independently but still behave polymorphically.

Solution: Visitors Design Pattern

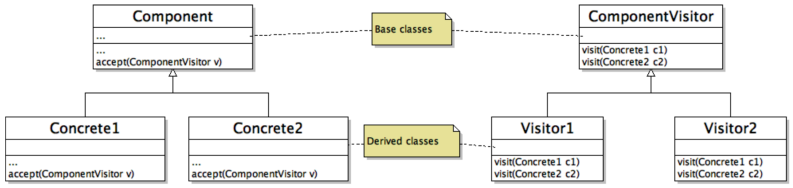
Approach:

- Encapsulate a desired operation in a separate object, called a visitor.
- The visitor object then traverses the elements of the tree.
- When a tree node "accepts" the visitor, it invokes a method on the visitor that includes the node type as an argument.
- The visitor will then execute the operation for that node

Implementation

Defining a Visitor:

1. Define the ComponentVisitor based class.
2. Add accept operation to Component base class and subclasses.
3. Define a concrete Leaf and Composite visitors as subclass of ComponentVisitor.



Implementation: The Computer File System

- Let's follow these steps to implement the find command on a file structure.

Implementation: The Computer File System

1. Defining the AbstractFileVisitor base class:

```
public interface AbstractFileVisitor {
    public void visit(BinaryFile bf);

    public void visit(Folder f);

    public void visit(Shortcut s);

    public void visit(TextFile tf);
}
```

Implementation: The Computer File System

2a. Adding accept operation to AbstractFile base class:

```
public abstract class AbstractFile {
    String name;

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

Implementation: The Computer File System

2b. Adding accept operation to BinaryFile subclass:

```
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);
    }

    @Override
    public void accept(AbstractFileVisitor afv) {
        afv.visit(this);
    }
}
```

Implementation: The Computer File System

2c. Adding accept operation to TextFile subclass:

```
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);
    }

    @Override
    public void accept(AbstractFileVisitor afv) {
        afv.visit(this);
    }
}
```

Implementation: The Computer File System

2d. Adding accept operation to Shortcut subclass:

```
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);
    }

    @Override
    public void accept(AbstractFileVisitor afv) {
        afv.visit(this);
    }
}
```

Implementation: The Computer File System

2e. Adding accept operation to Folder subclass:

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

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

    public Folder(String name) {
        this.name = name;
        includedFiles = new ArrayList<AbstractFile>();
    }

    @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);
    }

    @Override
    public void accept(AbstractFileVisitor afv) {
        afv.visit(this);
    }
}
```

Implementation: The Computer File System

- Adding accept operation to base class and subclasses:
  - Observe that the semantics and structure for accept in all subclasses are the same
    - In all cases, accept is defined to receive a single argument which is a pointer or reference to the abstract base class of the Visitor hierarchy.
    - The Leaf or Composite uses the accept method to call visit in abstract base class and passes itself as argument.

Implementation: The Computer File System

3. Defining a concrete visitor as subclass of AbstractFileVisitor:

```
public class FindVisitor implements AbstractFileVisitor{
    private String matchstring;
    public ArrayList<AbstractFile> result;
    public FindVisitor(String matchstring) {
        this.matchstring  = matchstring;
        result = new ArrayList<AbstractFile>();
    }
    @Override
    public void visit(BinaryFile bf) {
        if(bf.name.contains(matchstring))
            result.add(bf);
    }
    @Override
    public void visit(TextFile tf) {
        if(tf.name.contains(matchstring))
            result.add(tf);
    }
    @Override
    public void visit(Folder f) {}
    @Override
    public void visit(Shortcut s) {}
}
```

Implementation: The Computer File System

The Client:

- When a client needs an operation to be performed on an object structure:
  1. It creates a ConcreteVisitor object
  2. Then traverse the object structure
  3. Finally, visits each element in the object structure using the ConcreteVisitor object in (1).

Implementation: The Computer File System

1. Create a FindVisitor object

```
public static void main(String[] args) {
    AbstractFile root =  createDirStructure();

    Scanner userInput = new Scanner(System.in);
    while(true) {
        System.out.println("Enter search string or type ! to exit");
        String input = userInput.nextLine();

        if (!input.isEmpty()) {
            // Handle input
            FindVisitor visitor = new FindVisitor(input);
            traverseCompostite(root,visitor);

            for(AbstractFile f: visitor.result) {
                f.ls();
            }
        }
    }
}
```

Implementation: The Computer File System

2-3. Traverse the file system and visit each file in the system

- Who is responsible for traversing the object structure:
  1. The Composite
  2. The Client (An External class)
  3. The Visitor

Implementation: The Computer File System

2-3. Traverse the file system and visit each file in the system:

- Traversal encoded in the Composite

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

    @Override
    public void accept(AbstractFileVisitor afv) {
        afv.visit(this);
        includedFiles.forEach(af ->{
            af.accept(afv);
        });
    }
}
```

Implementation: The Computer File System

2-3. Traverse the file system and visit each file in the system:

- Traversal encoded in the Composite
  1. Simplest solution
  2. Only works if all the visitors need to visit the elements in the same order.

Implementation: The Computer File System

2-3. Traverse the file system and visit each file in the system:

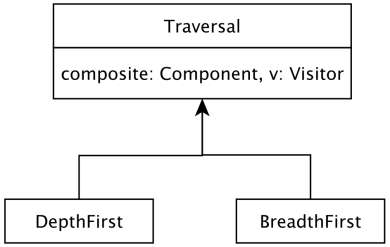
- Traversal encoded in Client

```
public class Client {
    //..
    public static void traverseCompostite(AbstractFile af, AbstractFileVisitor visitor) {
        //process composite children
        if(af instanceof Folder) {
            Folder folder = (Folder)af;
            for(AbstractFile temp:folder.getIncludedFiles()) {
                traverseCompostite(temp, visitor);
            }
        }
        else {
            af.accept(visitor);
        }
    }
}
```

Implementation: The Computer File System

2-3. Traverse the file system and visit each file in the system

- Traversal encoded in Client
  1. The client would require internal knowledge of the data structure
  2. The visitor remain generic
  3. Traversal order



Implementation: The Computer File System

2-3. Traverse the file system and visit each file in the system

- Traversal encoded in the Visitor

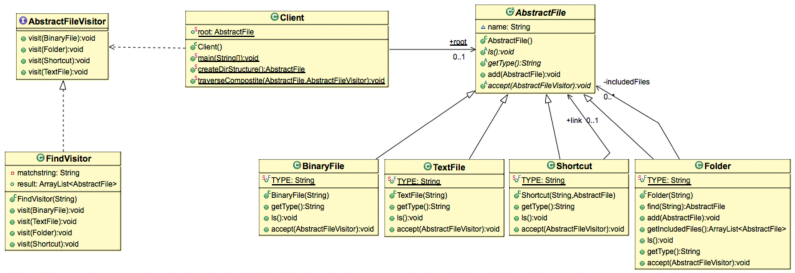
```
public class FindVisitor implements AbstractFileVisitor{
    //...
    @Override
    public void visit(Folder f) {
        f.getIncludedFiles().forEach(af ->{
            af.accept(this);
        });
    }
    //...
}
```

Implementation: The Computer File System

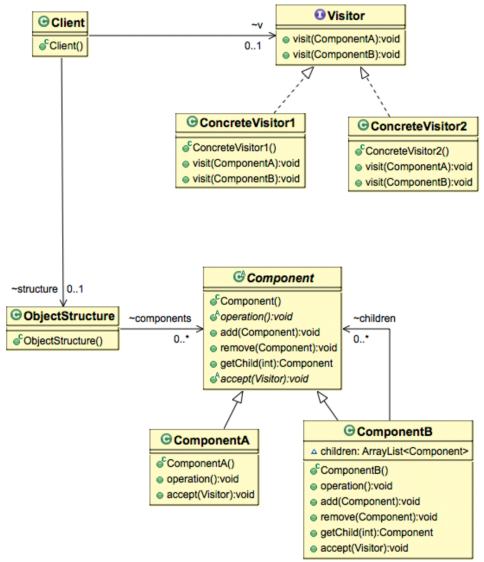
2-3. Traverse the file system and visit each file in the system

- Traversal encoded in the Visitor:
  1. Allows different traversal orders (for different visitors)
  2. This would allow each visitor to use a specific traversal.
  3. Useful for solutions that require complex traversals

Implementation: The Computer File System



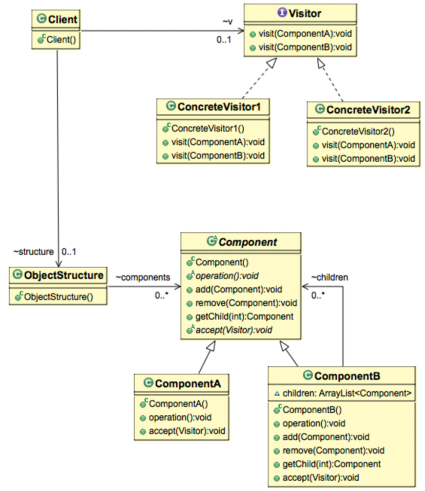
The structure of Visitors Design Pattern



The structure of Visitors Design Pattern

Double-dispatch

1. Assume you pass a parameter of type Visitor through the Component's accept method
2. Subsequently, you use a Client to call the accept method on a component instance



The structure of Visitors Design Pattern

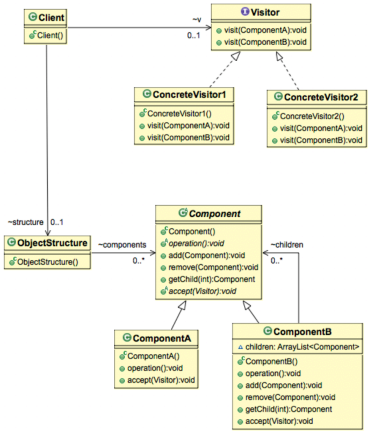
Double-dispatch

3. Then, the actual operation getting executed depends on:

   - The *type* of the component (i.e ComponentA or ComponentB).

   And

   - The *type* of the visitor instance you passed into accept (i.e ConcreteVisitor1) or ConcreteVisitor2)).



Consequences

+ The hierarchy of the structure remains intact and unpolluted.
+ Consolidates and encapsulates functionality in Visitor object.

- New concrete components may require changing the Visitor Interface.
- Circular dependency between Visitor and Component interfaces.

Applicability

- When classes define many unrelated operations
- Class relationships of objects in the structure rarely change, but the operations on then change often.
- Algorithms over the structure maintain state that's updated during traversal.