Visitor Pattern

Functional Decomposition in OO via double dispatching

Before (1): *call()* and *count()* on colors

```
Class Color
{ public:
        virtual void count() = 0;
        virtual void call() = 0;

static void report_num()
{
cout << "Reds " << s_num_red << ", Blus " << s_num_blu << '\n';
}

protected:
static int s_num_red, s_num_blu;
};
int Color::s_num_red = 0;
int Color::s_num_blu = 0;
```

Before (2)

```
class Red: public Color
{ public:
void count() { ++s_num_red; }
void call() { eye(); }
void eye() { cout << "Red::eye\n"; }
};

class Blu: public Color
{ public:
void count() { ++s_num_blu; }
void call() { sky(); }
void sky() { cout << "Blu::sky\n"; }
};
```

Before (3)

```
int main()
{ Color *set[] = { new Red, new Blu, new Blu, new Red, new Red, 0 };
  for (int i = 0; set[i]; ++i)
  {       set[i]->count();
          set[i]->call();
  }
 Color::report_num();
}

Red::eye
Blu::sky
Blu::sky
Red::eye
Red::eye
Reds 3, Blus 2
```

After (1)

The Color hierarchy specifies a single "accept()" method, and then the previous "count()" and "call()" methods are implemented as Visitor derived classes. When accept() is called on a Color object, that is the first dispatch. When visit() is called on a Visitor object, that is the second dispatch; and the "right thing" can be done based on the type of both objects.

```
Class Color
{ public:
        virtual void accept() = 0;
};
```

After (2)

```
class Red: public Color
{ public:
/* virtual */ void accept(Visitor*);
void eye() { cout << "Red::eye\n"; }
};

class Blu: public Color
{ public:
/* virtual */ void accept(Visitor*);
void sky() { cout << "Blu::sky\n"; }
};
```

After (3)

```
class Visitor
{ public:
virtual void visit(Red*) = 0;
virtual void visit(Blu*) = 0;
};
class CountVisitor: public Visitor
{ public :
   CountVisitor() { m_num_red = m_num_blu = 0; }
/*virtual*/ void visit(Red*) { ++m_num_red; }
/*virtual*/vvooiidd visit(Blu*) { ++m_num_blu; }
void report_num()
  { cout << "Reds " << m_num_red << ", Blus " << m_num_blu << '\n'; }
private:
int m_num_red, m_num_blu;
};
```

After (4)

```
class CallVisitor: public Visitor
{ public:
/*virtual*/ void visit(Red *r)
  { r->eye(); }
/*virtual*/void visit(Blu *b)
  { b->sky(); }
};
void Red::accept(Visitor *v)
{ v->visit(this); }

void Blu::accept(Visitor *v)
{ v->visit(this); }
```

After (5)

```
int main()
{  Color *set[] = { new Red, new Blu, new Blu, new Red, new Red, 0};
CountVisitor count_operation;
CallVisitor call_operation;
for (int i = 0; set[i]; i++)
   {       set[i]->accept(&count_operation);
           set[i]->accept(&call_operation);
  }
count_operation.report_num();
}

Red::eye
Blu::sky
Blu::sky
Red::eye
Red::eye
Reds 3, Blus 2
```

now let's look at a Java example

Decomposition

- How to break up the functionality of program into
  - Modules
  - Packages
  - Classes
  - Methods
  - Statements
  - ….

Two Popular Types of Decomposition

- Two most prevalent
  - Object-Oriented Decomposition
    - Units of decomposition are objects that implement functions
    - Data should be encapsulated and thus easy to change
    - Designing and extending interfaces is key
  - Functional Decomposition
    - Units of decomposition are functions

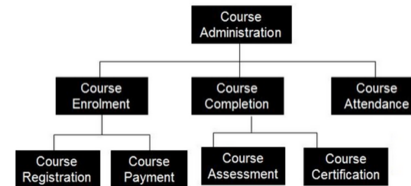
Functional Decomposition

- **Functional decomposition is probably the most pervasive design technique used for engineering systems**
- **Functional decomposition follows the transformation of inputs to outputs (under external control, if present)**
- **It's a top-down design technique**

Functional Decomposition

- **Top-down design process**
- **Recursively divide and conquer**
  - **Split a module into several submodules**
  - **Define the input, output, and behavior**
  - **Stop when you reach realizable components**
- **Decomposition is usually guided by Functional Analysis following Requirements Specification**
- **Functional Analysis**
  - **Identify top level functions (actions/objectives) the system must perform**
  - **Functions may ultimately be accomplished through the actions of the equipment, software, people**
  - **Specify "whats" (not "hows")**
  - **Iteratively keep decomposing functions onto a lower-level functions**

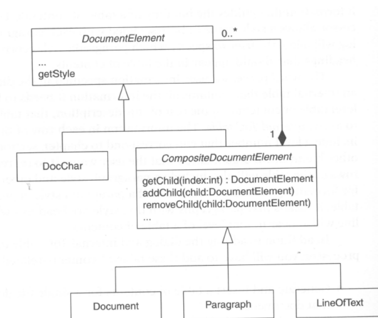
Functional Decomposition: Abstract Example



Tyranny of the Dominant Decomposition

- All decompositions favor some types of changes at the expense of others
  - This is known as "The Tyranny of the Dominant Decomposition"

Visitor Pattern: Document Example



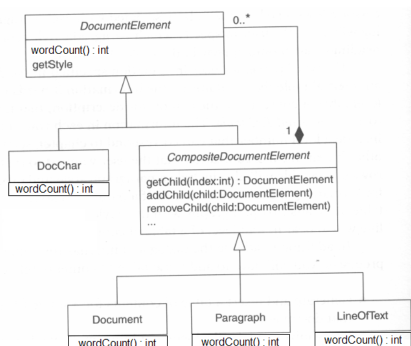
Document Example

- Want to add new data processing 'features'
  - Word Count
  - Convert document to XML format
  - Spell check
  - etc…

Two Approaches to this example

- Object-Oriented Decomposition
  - Each feature is implemented by adding a new method to each document element
- Functional Decomposition
  - Each feature is a class with methods for each kind of document element

Object-Oriented Decomposition



Visitor Pattern: Synopsis

- Implement functional decomposition in an object-oriented language
- Avoid tangling the class structure

Functional Decomposition

```
class WCVisitor implements Visitor {

//Each visit method deals with a specific concrete class
   public void visit(DocChar docChar) { … }
   public void visit(Document doc) { … }
   public void visit(Paragraph para) { … }
   public void visit(LineOfText lot) { … }
}
```

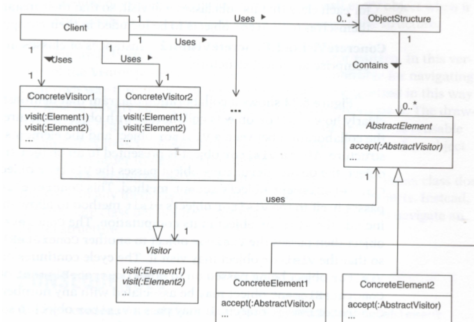
Visitor Pattern: Forces

- There are a variety of features (i.e., functions) that can be applied on an object structure
  - e.x. Word Count, XML Conversion, Spell checking, etc.
- The object structure is composed of objects that belong to different classes
  - The more classes, the more likely Visitor is the best choice

Visitor Pattern: Roles

- Roles:
  - Client
    - Creates visitor objects and applies them to object structures
  - Visitor
    - The `interface` that defines separate `visit` methods for each class to visit
  - ConcreteVisitor
    - Implements a new feature by implementing `visit` methods
  - ObjectStructure
    - Root object of an object structure
  - AbstractElement
    - Defines a method that allows objects to `accept` visitors
  - ConcreteElement
    - Implements the `accept` method
    - Implementation is mostly schematic

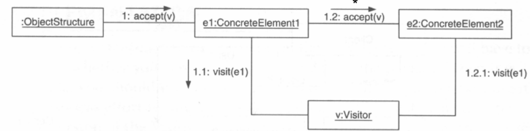
Visitor Pattern: UML class diagram



Visitor Pattern: Double Dispatch

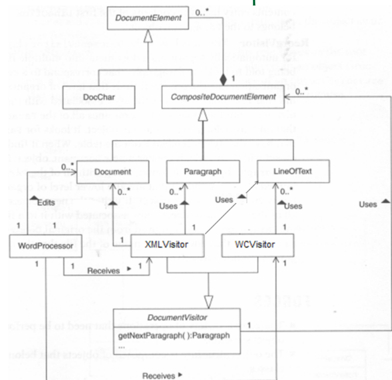
1. accept(Visitor) is called on an object
   - Dispatches to the right element class
2. visit(this) is called on the visitor
   - Dispatches to the right visitor class
3. **accept(Visitor) is called on all children (fields) of the element**
   - Causes a traversal over the object graph
     - e.g., sequence, depth first, breadth first, etc..

Visitor Pattern: Scenario



• Step 1.2 is repeated for all "children" (or successor nodes) of an element in the object graph (and repeated recursively) until some termination criteria is reached (e.x. leaves of a tree structure)

Visitor Pattern for example



accept Method Implementation

```
class Document extends CompositeDocumentElement {

    DocumentElement[ ] children;
    //Other details of class are omitted

    public void accept(Visitor v)
    {
         v.visit(this);
         for(int i=0;i<children.length;i++)
                 children[i].accept(v);
         // optionally: v.endVisit(this);   as in the XML visitors
    }

}
```

visit Method: general idea

```
class WCVisitor implements Visitor {

//Each visit method deals with a specific concrete class
    public void visit(DocChar docChar) { … }
    public void visit(Document doc) { … }
    public void visit(Paragraph para) { … }
    public void visit(LineOfText lot) { … }
}
```

visit Method Implementation

```
class WCVisitor implements Visitor {
    int wc = 0;
    boolean wordDivider= false;

    public void visit(Document d) { wordDivider= true; }
     // similar methods are omitted

    public void visit(DocChar char) {
        if(char.isWhiteSpace()) {
                wordDivider= true;
        } else {
                if(wordDivider) {
                        wc++;
                } else {
                        wordDivider = false;
                }
        }
    }
}
```

XML Visitor Example

```
class XMLVisitor implements Visitor {
    int indent = 0;
    //Other methods are omitted
    public void visit(Document doc) {
            indent();
            indent += 2;
            System.out.println("<document>");
    }

    public void endVisit(Document doc) {
            indent();
            System.out.println("</document>");
            index -= 2;
    }

    public void indent() {
            for(int i=0;i<=indent;i++)
                    System.out.print(" ");
    }
}
```

Appendix

Details of Functional Decomposition

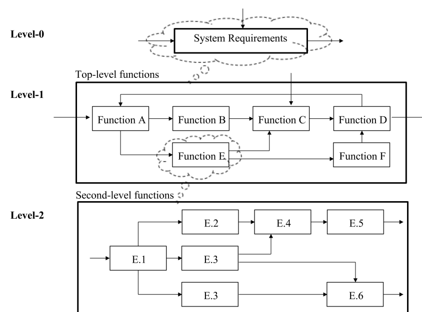
Bottom-Up vs Top-Down

- **Bottom-up**
  - The designer starts with basic components and synthesizes them to create the overall system
    - Consider designing a car from many parts – will the final product meet requirements?
  - Pros: Leads to efficient subsystem
  - Cons: Difficult to meet requirement; Complexity difficult to manage; Difficult redesign
- **Top-down**
  - The designer has an overall vision of what the final system must do, and the problem is partitioned into components, or subsystems that work together to achieve the goal
  - Pros: Highly predictable design cycle; Full utilization of requirements; Efficient development of large systems
  - Cons: More time spent in planning; May limit creativity
- **In reality, the designer must alternate between both designs**

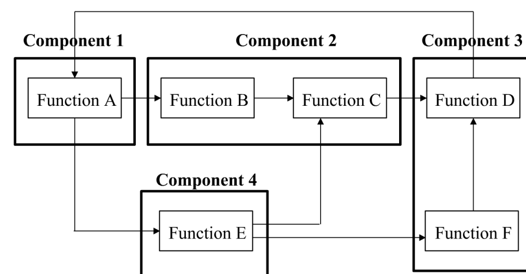
Steps of Functional Decomposition

- **Functional Flow Block Diagram**
  - End product of functional decomposition – shows sequence of system activities
  - Incrementally refine and mark inputs / outputs / controls
  - Used to illustrate system organization and major interfaces
  - Build at the later stage of Concept Generation
  - Sample system functional breakdown - see next page
- **Functional Allocation**
  - Combine or group similar functions into logical subdivisions, identifying major groups
  - Conversion of the "whats" into "hows" – system broken down into components
  - Trade-offs studies involved – implementation approaches involved

Flow Block Diagram



Functional Allocation



Levels in Functional Decomposition

- ***Level 0***
  - This is where you start – the highest level involving one block only, i.e. a block corresponding to your system
  - Define inputs, outputs and system functionality (requirements)
- ***Level 1***
  - Typically referred as main system architecture
  - Architecture means the organization and interconnection between modules. Describe the operation – how modules work together.
  - Define functional requirements for each module.
- ***Level 2***
  - Typically shows the organization of components within a single module

More Functional Decomposition

- **Decomposition process continues until you reach Detail Design**
  - This is where the problem can be decomposed no further and you identify all available components
  - Number of levels can vary – your design should have Level 0, 1 and 2, at minimum
- **Proposal requires to include Level 0 and Level 1 design**
- **Design Document requires all levels**

Example of Functional Decomposition