Design Patterns

David Talby

Contents

- Representing Data Structures
  - Composite
  - Flyweight
  - Decorator
- Traversing Data Structures
  - Iterator
  - Visitor

Design Patterns – Reminder

- Documented Proved Design Experience
  - Finding the right classes
  - Finding them faster
  - Common design jargon
  - Consistent format
  - Coded infrastructures
- Criteria for Success
  - Open-Closed Principle
  - Single Choice Principle

1. Composite

- A program must treat simple and complex objects uniformly
- For example, a painting program has simple objects (lines, circles and texts) as well as composite ones (wheel = circle + six lines).

The Requirements

- Treat simple and complex objects uniformly in code - move, erase, rotate and set color work on all
- Some composite objects are defined statically (wheels), while others dynamically (user selection)
- Composite objects can be made of other composite objects
- We need a smart **data structure**

The Solution

- All simple objects inherit from a common interface, say *Graphic*:

```
class Graphic {
   void move(int x, int y) = 0;
   void setColor(Color c) = 0;
   void rotate(double angle) = 0;
}
```

- The classes *Line*, *Circle* and others inherit *Graphic* and add specific features (radius, length, etc.)

The Solution II

- This new class inherits it as well:

```
class CompositeGraphic
  : public Graphic,
    public list<Graphic>
{
  void rotate(double angle) {
    for (int i=0; i<count(); i++)
      item(i)->rotate();
  }
}
```


The Solution III

- Since a *CompositeGraphic* is a list, it had *add(), remove()* and *count()* methods
- Since it is also a *Graphic*, it *has rotate(), move()* and *setColor()* too
- Such operations on a composite object work using a 'forall' loop
- Works even when a composite holds other composites - results in a tree-like data structure

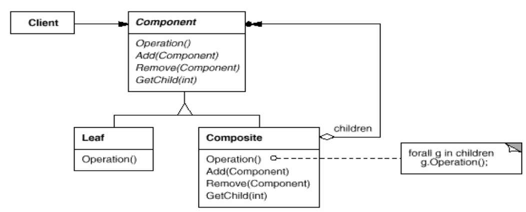
The Solution IV

- Example of creating a composite:

```
CompositeGraphic *cg;
cg = new CompositeGraphic();
cg->add(new Line(0,0,100,100));
cg->add(new Circle(50,50,100));
cg->add(t); // dynamic text graphic
cg->remove(2);
```

- Can keep order of inserted items if the program needs it

The GoF UML



- Single Inheritance
- Root has *add(), remove()* methods

The Fine Print

- Sometimes useful to let objects hold a pointer to their parent
- A composite may cache data about its children (count is an example)
- Make composites responsible for deleting their children
- Beware of circles in the graph!
- Any data structure to hold children will do (list, array, hashtable, etc.)

Known Uses

- In almost all O-O systems
- Document editing programs
- GUI (a form is a composite widget)
- Compiler parse trees (a function is composed of simpler statements or function calls, same for modules)
- Financial assets can be simple (stocks, options) or a composite portfolio

Pattern of Patterns

- Encapsulate the varying aspect
- Interfaces
- Inheritance describes variants
- Composition allows a dynamic choice between variants

Criteria for success:
Open-Closed Principle
Single Choice Principle

Contents

- Representing Data Structures
  - Composite
  - Flyweight
  - Decorator
- Traversing Data Structures
  - Iterator
  - Visitor

A Word Processor

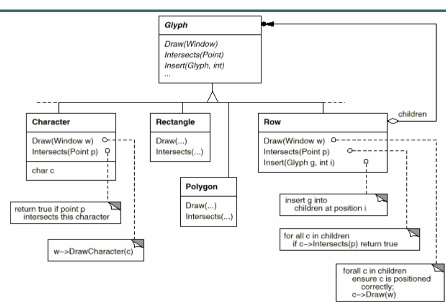
- Pages, Columns, Lines, Letters, Symbols, Tables, Images, ...
- Font and style settings per letter
- Frames, Shadows, Background, Hyperlink attached to anything
- Unlimited hierarchy: Tables with several Paragraphs containing hyper-linked images inside tables
- Should be open for additions...

A Data Structure

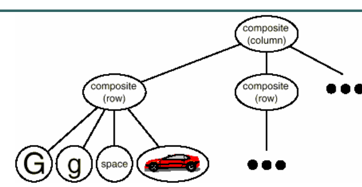
- First, a uniform interface for simple things that live in a document:

```
class Glyph
{
  void draw(Window *w) = 0;
  void move(double x, double y) = 0;
  bool intersects(Point *p) = 0;
  void insert(Glyph *g, int i) = 0;
  void remove(int i) = 0;
  Glyph* child(int i) = 0;
  Glyph* parent() = 0;
}
```


Composite Documents



At Runtime



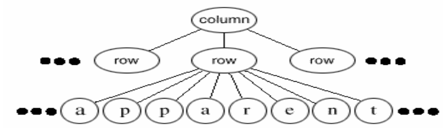
- Unlimited Hierarchy problem solved
- Dynamic selection of composites
- Open for additions

2. Flyweight

- Use sharing to support a large number of small objects efficiently
- For example, if every character holds font and style data, a long letter will require huge memory
- Even though most letters use the same font and style
- How do we make it practical to keep each character as an object?

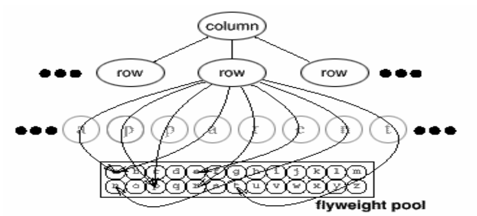
The Requirements

- Reduce the memory demands of having an object per character
- Keep the flexibility to customize each character differently



The Solution

- Intrinsic state = worth sharing
- Extrinsic state = not worth sharing



The Solution II

- Put extrinsic state in a class:

```
class CharacterContext {
   Font* font;
   bool isItalic, isBold, ...;
   int size;
   int asciiCode;
   // many others…

   draw(int x, int y) { ... }
   // other operational methods
}
```


The Solution III

- Original class holds rest of state:

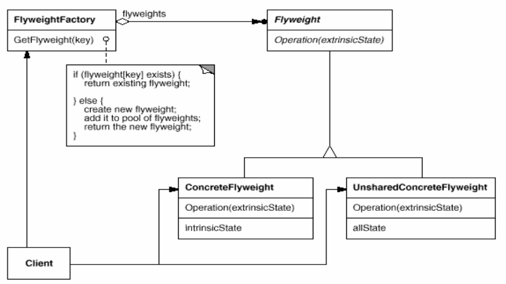
```
class Character : public Glyph {
   CharacterContext *cc;
   int x, y;

   draw() {
     cc->draw(x,y);
   }
}
```


The Solution IV

- A factory manages the shared pool
- It adds the object to the pool if it doesn't exists, and returns it
- Here's *Character*'s constructor:

```
Character(int x, int y, Font *f, …) {
 this->x = x;
 this->y = y;
 this->cc =
   factory.createCharacter(f, …);
}
```

The UML



The Fine Print

- There's a lot of tradeoff in what is defined as "extrinsic"
- Shared pool is usually a hash table
- Use reference counting to collect unused flyweights
- Don't rely on object identity
  - Different objects will seem equal

Known Uses

- Word processors
  - Average 1 flyweight per 400 letters
- Widgets
  - All data except location, value
- Strategy design pattern
- State design pattern

Contents

- Representing Data Structures
  - Composite
  - Flyweight
  - Decorator
- Traversing Data Structures
  - Iterator
  - Visitor

3. Decorator

- Attach additional features to an object dynamically
- For example, many features can be added to any glyph in a document
  - Background, Note, Hyperlink, Shading, Borders, …

The Requirements

- We can freely combine features
  - An image can have a background,
    a border, a hyper-link and a note
- Features are added and removed dynamically
- Can't afford a class per combination
- Should be easy to add new features
  - Don't put it all in *Glyph*

The Solution

- Meet Decorator, a class for adding responsibilities to another glyph:

```
class Decorator : public Glyph
{
   void draw() {
       component->draw();
   }
   // same for other features
private:
   Glyph *component;
}
```

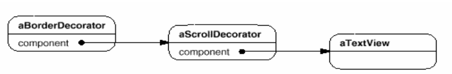

The Solution II

- Define concrete decorators:

```
class BackgroundDecorator
   : public Decorator
{
   void draw() {
     drawBackground();
     glyph->draw();
   }
}
```

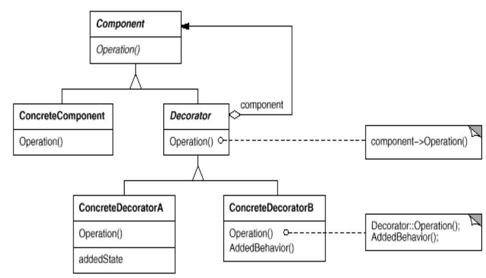

The Solution III

- Many decorators can be added and removed dynamically:



- Behavior can be added before and after calling the component
- Efficient in space
- Order of decoration can matter

The UML



The Fine Print

- The *Decorator* class can be omitted if there's only one decorator or *Glyph* is very simple
- The *Glyph* class should be lightweight and not store data

Known Uses

- Embellishing Document
  - Background, Border, Note, ...
- Communication Streams
  - Encrypted, Buffered, Compressed

Data Structure Summary

- Patterns work nicely together
  - Composite, Decorator, Flyweight don't interfere
- Data structures are not layered
  - Instead, clients work on a Glyph interface hiding structures of unknown, dynamic complexity

Saving and Loading

- Each Glyph should have "deep" *read()* and *write()* methods
- Save to disk / Send over network by simply writing the root Glyph object of a document
- All optimizations saved as well!
- Also works on subtrees
- Little coding

Cut, Copy, Paste

- Cut = Detach a subtree
- Copy = Clone a subtree
- Paste = Attach a subtree
- Also works on composite glyphs
- Glyphs should hold a reference to parents for the cut operations
- Cloning of a flyweight should only increase its reference count!

Contents

- Representing Data Structures
  - Composite
  - Flyweight
  - Decorator
- Traversing Data Structures
  - Iterator
  - Visitor

4. Iterator

- Traverse a data structure without exposing its representation
- An extremely common pattern
- For example, a list should support forward and backward traversals
  - Certainly not by exposing its internal data structure
- Adding traversal methods to *List*'s interface is a bad idea

The Requirements

- Traversal operations should be separate from *List<G>*'s interface
- Allow several ongoing traversals on the same container
- Reuse: it should be possible to write algorithms such as *findItem* that work on any kind of list

The Solution

- Define an abstract iterator class:

```
class Iterator<G> {
   void first() = 0;
   void next() = 0;
   bool isDone() = 0;
   G* item() = 0;
}
```


The Solution II

- Each data structure implementation will also implement an iterator class:
  - `ListIterator<G>`
  - `HashTableIterator<G>`
  - `FileIterator<G>`
  - `StringIterator<G>`
- Each data structure can offer more than one iterator:
  - Forward and backward iterators
  - Preorder, inorder, postorder

The Solution III

- For example:

```
class BackwardArrayIterator<G>
  : public Iterator<G>
{
    Array<G> *container;
    int pos;
 public:
    BackwardArrayIterator(Array *a)
      { container = a; first(); }
    next()
      { --pos; }
    // other methods easy
 }
```


The Solution IV

- A data structure's interface should return iterators on itself:

```
class List<G>
{
  Iterator<G>* getForwardIterator()
    { return new
         ListForwardIterator(this); }
  Iterator<G>* getBackwardIterator()
     // similarly
}
```

- Now every *LinkedList* object can have many active iterators

The Solution V

- Writing functions for containers:

```
void print(Iterator<int>* it)
{
  for (it->first();
       !it->isOver();
       it->next())
     cout << it->item();
}
```

- Using them:

```
print(myList->getBackwardIterator());
print(myTable->getColumnItr("Age"));
print(myTree->getPostOrderIterator());
```


The Solution VI

- Generic algorithms can be written:

```
G* findItem(Iterator<G>* it,
            G *element)
{
   while (!it->isOver())
   {
     if (it->item() == element)
        return element;
     it->next();
   }
   return NULL;
}
```

The Requirements II

- Some iterators are generic:
  - Traverse every *n*'th item
  - Traverse items that pass a filter
  - Traverse only first *n* items
  - Traverse a computed view of items
- Such iterators should be coded once
- It should be easy to combine such iterators and add new ones
- Their use should be transparent

The Solution

- Use the Decorator design pattern
- For example, *FilteredIterator<G>* receives another iterator and the filtering function in its constructor
- It delegates all calls to its internal iterator except *first()* and *next()*:

```
void next() {
    do it->next()
    while (!filter(it->item() &&
              !it->isOver());
}
```

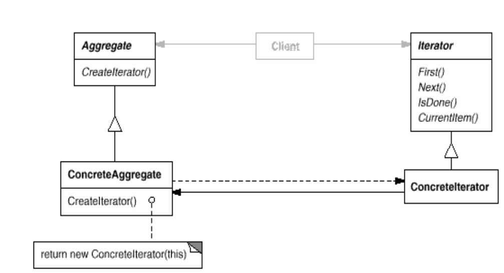

The Solution II

- It is then easy to combine such generic iterators
- Print square roots of the first 100 positive elements in a list:

```
print(new LimitedIterator(100,
      new ComputedIterator(sqrt,
      new FilteredIterator(positive,
      list->getForwardIterator()))));
```

- Adding an abstract *DecoratorIterator* reduces code size if many exist

The UML



The Fine Print

- Everything is a container
  - Character strings
  - Files, both text and records
  - Socket streams over the net
  - The result of a database query
  - The bits of an integer
  - Stream of random or prime numbers
- This allows reusing the *print*, *find* and other algorithms for all of these

The Fine Print II

- Iterators may have privileged access
  - They can encapsulate security rights
- Kinds of abstract iterators
  - Direct access iterators
  - Access the previous item
- Robustness issues
  - Is the iterator valid after insertions or removals from the container?
- Iterators and the Composite pattern

Known Uses

- All major standard libraries of popular programming languages
  - STL for C++
  - The Java Collections Framework
- New libraries for file, network and database access in C++ conform to STL's iterators as well

Contents

- Representing Data Structures
  - Composite
  - Flyweight
  - Decorator
- Traversing Data Structures
  - Iterator
  - Visitor

5. Visitor

- Separate complex algorithms on a complex data structure from the structure's representation
- For example, a document is a composite structure involved in many complex operations
  - Spell check, grammar check, hyphenation, auto-format, …
- How do we avoid cluttering Glyph subclasses with all this code?

The Requirements

- Encapsulate complex algorithms and their data in one place
- Outside the data structure
- Easily support different behavior for every kind of *Glyph*
- Easily add new tools

The Solution

- Say hello to class *Visitor*:

```
class Visitor {
 public:
   void visitImage(Image *i) { }
   void visitRow(Row *r) { }
   void visitTable(Table *t) { }
   // so on for every Glyph type
}
```

- Every tool is a subclass:

```
class SpellChecker : public Visitor
```


The Solution II

- Add to *Glyph*'s interface the ability to accept visitors:

```
void accept(Visitor *v) = 0;
```

- Every glyph subclass accepts a visitor by an appropriate callback:

```
class Image : public Glyph {
   void accept(Visitor *v)
     { v->visitImage(this); }
```

- This way the visitor is activated for the right kind of glyph, with its data

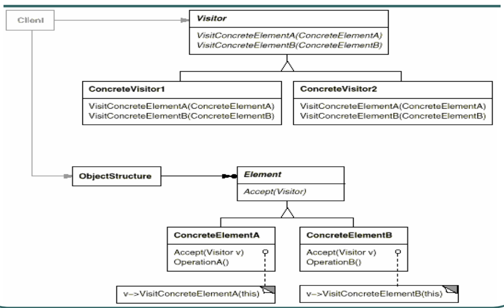
The Solution III

- Initiating a spell check (one option):
  - Create a *SpellChecker* object
  - `root->accept(sc);`
- Graphic non-text glyphs will just ignore the visit
  - This is why *Visitor* includes default empty method implementations
- Composite glyphs also do nothing
  - They can forward the visit to children. This can be coded once in *CompositeGlyph*

The Solution IV

- Easy to add operations
  - Word count on characters
  - Filters such as sharpen on images
  - Page layout changes on pages
- Works on any glyph
  - In particular, a dynamic selection as long as it's a composite glyph
- Adding a tool does not require recompilation of *Glyph* hierarchy

The UML



The Fine Print

- The big problem: adding new Glyph subclasses is hard
  - *Requires small addition to Visitor, and recompilation of all its subclasses*
- How do we traverse the structure?
  - *Using an iterator*
  - *From inside the accept() code*
  - *From inside the visitxxx() code*
- Visitors are really just a workaround due to the lack of *double dispatch*

Known Uses

- Document Editors
  - Spell Check, Auto-Format, …
- Photo Editors
  - Filters & Effects
- Compilers
  - Code production, pretty printing, tests, metrics and optimizations on the syntax tree

Summary

- Pattern of patterns
  - Encapsulate the varying aspect
  - Interfaces
  - Inheritance describes variants
  - Composition allows a dynamic choice between variants
- Design patterns are old, well known and thoroughly tested ideas
  - Well over twenty years!