ELEC 372
Engineering Software Systems

Week 7
Design Patterns Continued

T.C. Nicholas Graham
Diane Kelly
http://www.cs.queensu.ca/~elec372

Behavioural Patterns

- The patterns we have seen so far address *structural* problems in programs
  - *Adapter* -- structures programs to aid reuse
  - *Façade* -- localizes references to a subsystem, aiding modifiability
- Also interesting are
  - *Creational patterns* -- help in object creation
  - *Behavioural patterns* -- help manage runtime behaviour of program

Abstract Factory

[Gamma et al., pp. 87-95]

- Motivation
  - Imagine you have a book ordering system in which you need to maintain a list of books
    - *Book* is a class containing a title, author, ISBN, …
    - A *BookList* is a list of books, including a quantity >= 1 for each book
  - Initially, you wish to maintain a simple *Vector*
  - However, after the system has been delivered, you decide you want a data structure that also permits books to be represented in sorted order

```
import java.util.Vector;
public class BookListClassAdapter
             extends Vector implements BookList {
    public int getNumBooks() {
        return size();
    }

    public Book getBook(int n) {
        return (Book) elementAt(n);
    }
    public void addBook(Book newBook) {
        for (int i=0; i<getNumBooks(); i++) {
            Book b=getBook(i);
            if (b.isSameBook (newBook)) {
                b.setQuantity (b.getQuantity()
                    + newBook.getQuantity());
                return;
            }
        }
        add (newBook);
    }
}
```

Recall our initial implementation of a BookList

Abstract Factory

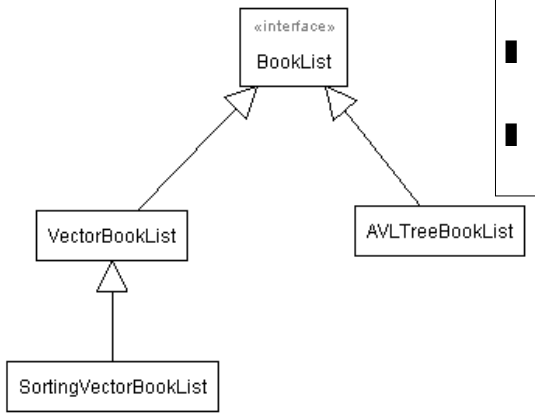
- Adding sorting functionality to the *BookList* class
  - Option 1: Modify *BookList* so that every time a book is entered, the Vector is sorted
    - Pros
      - Localizes change to *BookList* class itself
    - Cons
      - Source to *BookList* may not be available
      - *BookList* may be used in other programs, other subsystems of same program, so that change in class functionality could have wider impact than planned
      - Danger of version proliferation, where different versions of *BookList* with different functionality appear in different programs -- increases problems with maintenance

Abstract Factory

- Option 2:
  - Create a *BookList* interface with the required functionality of maintaining a list of books
  - Create an abstract factory class that creates an instance of the correct kind of *BookList*

BookList variants

- *BookList* interface specifies the required operations for a list of books
- *VectorBookList* is the simple vector implementation
- *SortingVectorBookList* adds sorting functionality
- *AVLTreeBookList* provides faster lookups, sortings

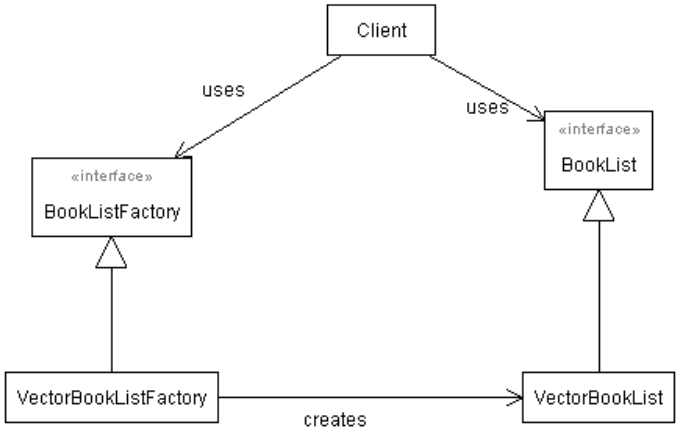


Problems with Changing Implementations

- Problem: If we change from the *VectorBookList* to *SortingVectorBookList* implementation, need to ensure that wherever we create a book list, we replace **"new VectorBookList()**" with "**new SortingVectorBookList()**"
- This could be a large change
  - What if source to classes creating book lists is not available?

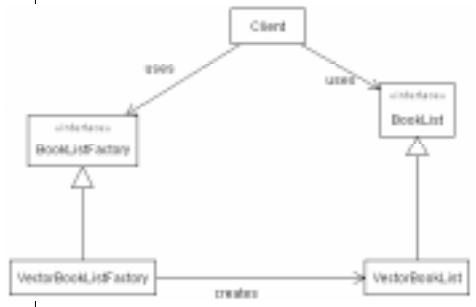


Abstract Factory



Abstract Factory

- Whenever a client object wishes to create an instance of a *BookList*, it calls the *createBookList()* method of the *BookListFactory*
- This is an object that returns a value of type *BookList*
- The *BookListFactory* is specialized to a factory capable of returning the correct kind of *BookList*
- Here, a *VectorBookListFactory* creates instances of *VectorBookList*



BookListFactory

BookListFactory.java

```
public interface BookListFactory {
    public BookList createBookList ();
}
```

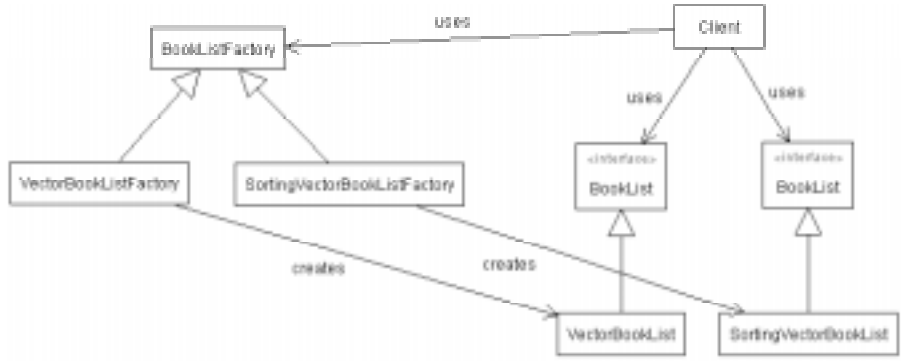
```
class VectorBookListFactory
        implements BookListFactory {
    public BookList createBookList () {
        return new VectorBookList ();
    }
}
```

VectorBookListFactory.java

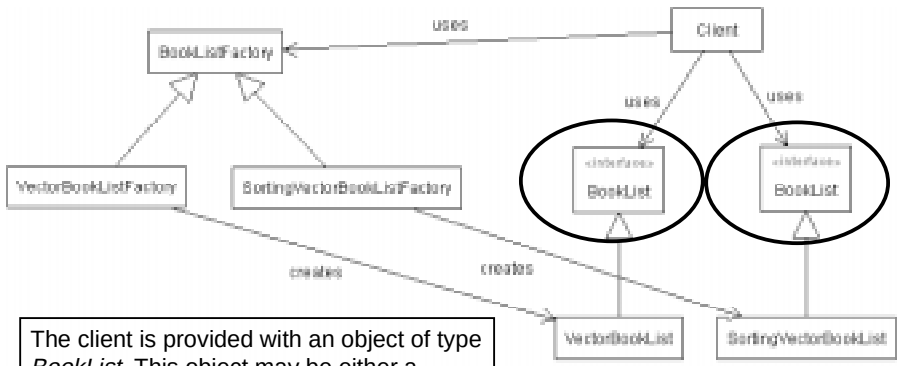
Client.java

```
class Client {
   …
   BookListFactory blf;
   …
   BookList b = blf.createBookList();
}
```

Extending to Multiple Factories

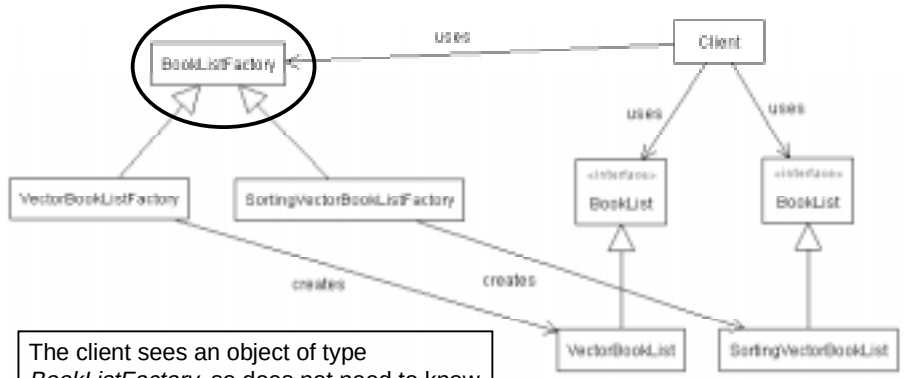


Extending to Multiple Factories



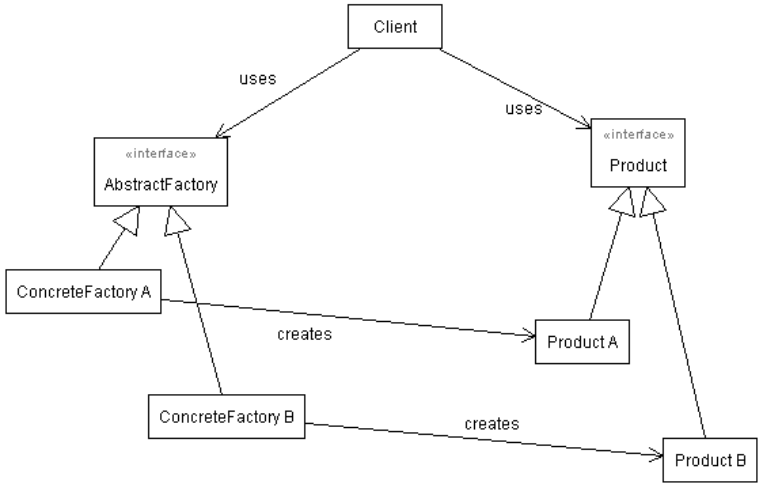
The client is provided with an object of type *BookList*. This object may be either a *VectorBookList* or a *SortingVectorBookList*, depending on which factory was used.

Extending to Multiple Factories



The client sees an object of type *BookListFactory*, so does not need to know whether it is using a *VectorBookListFactory* or a *SortingVectorBookListFactory*

General Form of Abstract Factory



Relating to Quality Attributes

- Modifiability
  - Increased through use of indirection in creation process
  - Creator does not need to know exactly what type of object is being created

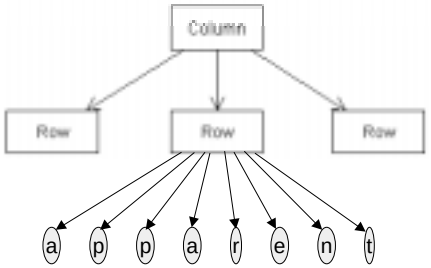
The Flyweight Pattern

[Gamma et al., pp 195-206]

- Sometimes objects too expensive for applications where large numbers of objects required
- Flyweight objects provide cheaper object mechanism
- Illustrates abstract factory pattern

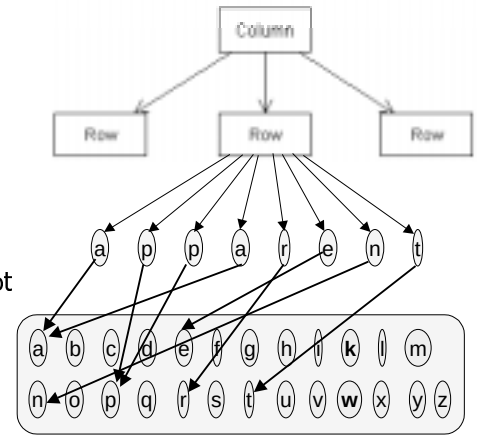
Flyweight

- In a text editor, wish to represent a set of rows, and within each row a set of columns
- Each row in turn contains a set of characters
- Logically, each character is an object
- However, a document of tens of thousands of characters would require tens of thousands of objects

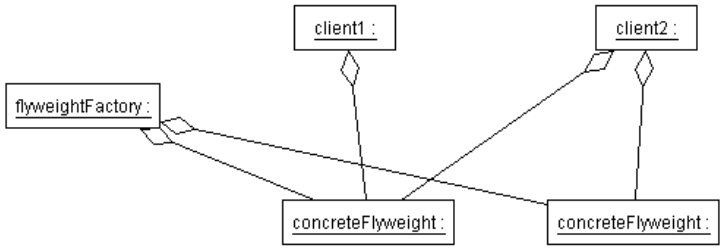


Flyweight

- Instead, create one object per character
- Each of these is called a flyweight object
- Flyweight objects can contain only data shared amongst all instances of objects
  - E.g., character name, but not position, font
- Extrinsic state must be passed as a parameter to operations on flyweight



Flyweight Implementation



Relation to Quality Attributes

- Performance
  - Systems with tens/hundreds of thousands or more of objects can be prohibitively expensive in memory use
  - Flyweight objects reduce this expense
- Modifiability
  - Encapsulation of system functionality no longer as clear
  - Separation of intrinsic/extrinsic state into different objects

The Iterator Pattern

[Gamma et al, pp 257-271]

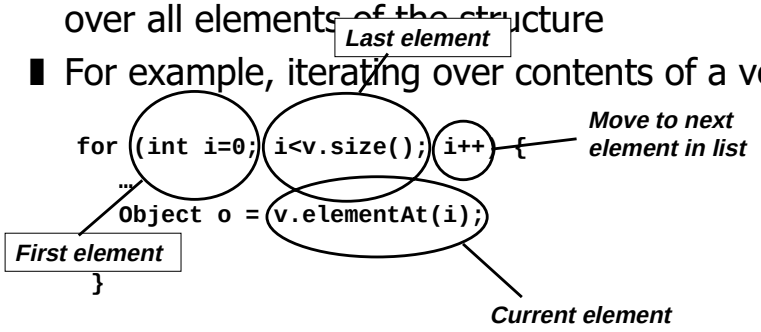
- Many structures require support for iterating over all elements of the structure
- For example, iterating over contents of a vector:

```
for (int i=0; i<v.size(); i++) {
   …
   Object o = v.elementAt(i);
   …
 }
```

The Iterator Pattern

[Gamma et al, pp 257-271]

- Many structures require support for iterating over all elements of the structure
- For example, iterating over contents of a vector:



The Iterator Pattern

- **Iteration allows a program to successively visit each element in a container class**
  - **I.e. over any class that contains a collection of elements**
- **List (Vector), Tree, Hash Table, Graph, …**

Iterators in Java

- **An enumeration provides iteration functionality that abstracts the underlying container type**

```
public interface Enumeration {

   // Returns true as long as there are more
   // elements to visit
   boolean hasMoreElements();

   // Returns next element to visit
   Object nextElement();
}
```

- **Each element will be returned by *nextElement* exactly once**
- **Order of visitation not specified**

Iterators in Java

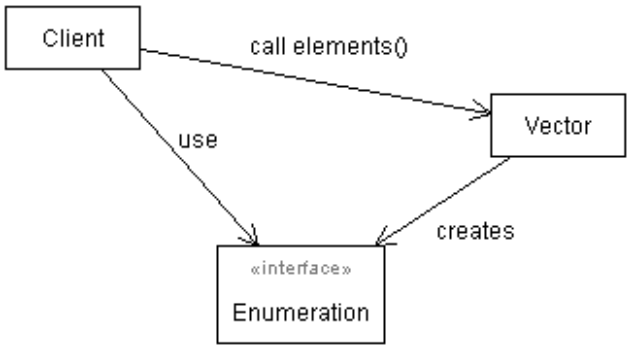
- **In *Vector*:**

```
// Returns all elements in the Vector
public Enumeration elements();
```

- **In *Hashtable*:**

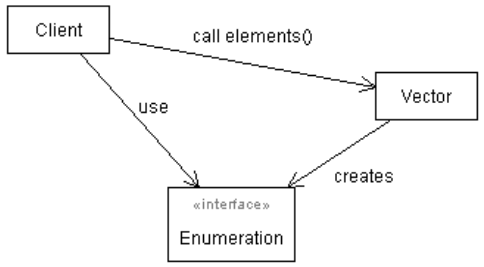
```
// Returns all elements in the Hashtable.
// That is, returns all values that have been
// entered into the hashtable via put
public Enumeration elements();
```

Example: Using an Iterator with a Vector



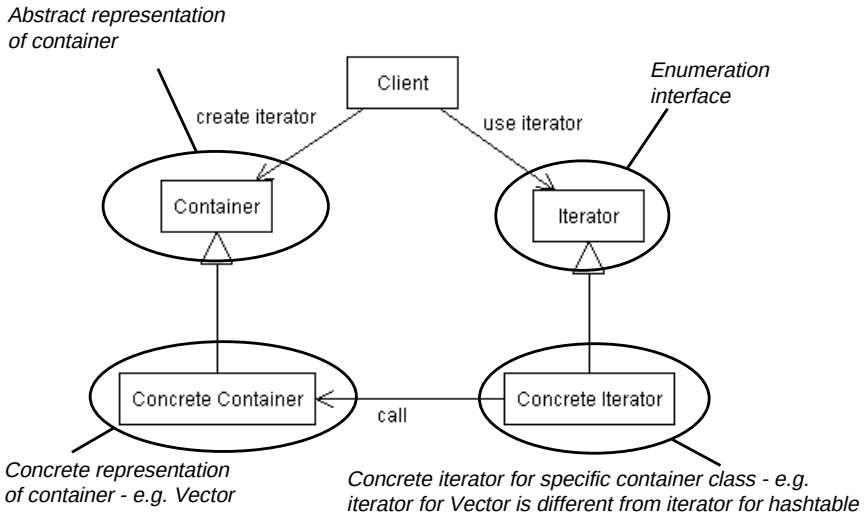
Using an Iterator with a Vector

- Client creates an iterator by calling the Vector's *elements* method
  - *Vector* acts like an iterator factory
- Client then accesses elements of Vector via Enumeration's methods




```
for (Enumeration e = v.elements(); e.hasMoreElements();) {
   Object o = e.nextElement();
   …
}
```

General Form of Iterator



Relation to Quality Attributes

- Modifiability
  - Separates iteration behaviour from specifics of container class
  - Therefore can change container class without impacting code that iterates over it
- Performance
  - May be some minor performance penalty through indirection in iterator