Flyweight Bug

class PRleaf {
private:
    DataRec* data;
    Bool _isLeaf;
public:
    PRleaf(); // Empty leaf constructor (flyweight)
    PRleaf(int x, int y); // Full leaf constructor
    bool isLeaf() { return true }
    void getValue(DataRec* data) { return data; }
    void getChild() { return null; }
}

Problem: Constructor keeps making new empty nodes.
Flyweight Solution

- Protect the constructor
- Replace constructor with a creation method
- Keep a static instance of the flyweight

```cpp
class PRleaf {
private:
    DataRec* data;
    static PRleaf* _theEmpty = NULL;
protected:
    PRleaf(); // Empty leaf constructor (flyweight)
public:
    static PRleaf* getEmpty();
    PRleaf(int x, int y); // Full leaf constructor
    bool isLeaf() { return true; }
    void getValue(DataRec* data) { return data; }
    void getChild() { return null; }
}

static PRleaf* PRleaf::getEmpty() {
    if (_theEmpty == NULL)
        _theEmpty = PRleaf();
    return _theEmpty;
}

foo = PRleaf::getEmpty();
```
Alternate Implementation

class PREmpty {
  public:
    PREmpty();  // Empty leaf constructor (flyweight)
    bool isLeaf() { return true }
}

PREmpty TheGlobalEmpty = new PREmpty();

foo = TheGlobalEmpty;

... versus ...

class PREmpty {
  private:
    static PRleaf* _theEmpty = NULL;
  protected:
    PRleaf();  // Empty leaf constructor (flyweight)
  public:
    static PRleaf* getEmpty();
    bool isLeaf() { return true }
}

static PRleaf* PRleaf::getEmpty() {
  if (_theEmpty == NULL)
    _theEmpty = PRleaf();
  return _theEmpty();
}

foo = PREmpty::getEmpty();
Singleton Issues

An alternative to a global variable.
- Guarantees unique instance of class.
- Only need to know class, not global variable.
- Can be subclassed.
- Only created if used.
- Global variables are instantiated at start of program – might need more state (derived later) to permit instantiation.

Subclassing Singletons
- Which one(s) get instantiated? Creator method decides.
- Inform creator (in base class): Fixes choices of subclasses.
- Put creator in subclass: Link in only the preferred one.
- Registry: Store name/object pairs for all singleton classes.
Decorator Pattern

Want to add responsibilities to object, without changing interface.

Subclassing is poor if there are various interchangeable options.
  • For windows: add borders and/or scrollbars.
  • Subclassing leads to combinatorial explosion of choices.

Alternative:
  • “Wrap” the object in another class with same interface and same base class.
  • Decorator keeps a reference to the original object, and passes through calls to its methods.
  • Client need not know the object is decorated.
  • Alternatively, client can also make use of decorator’s specific capabilities.
Decorator Pattern

Decorator Pattern Structure: P. 177
Java File I/O

Makes extensive use of Decorators.

```java
public class DataInputStream
    extends FilterInputStream {
    public DataInputStream(InputStream In);
    public int read(byte[] b, int off, int len);
    public int readInt();
    public int readDouble();
    public int readChar();
    ...
}

public class FileInputStream extends InputStream {
    public FileInputStream(string name);
    public int read(byte[] b, int off, int len);
}

public class BufferedInputStream
    extends FilterInputStream {
    public BufferedInputStream(InputStream in);
    public int read(byte[] b, int off, int len);
}
```
Example

```java
static void readIt(Stringinfile) throws IOException {
    DataInputStream in = new DataInputStream(
        new BufferedInputStream(
            new FileInputStream(infile)));

    Int val = in.readInt();
    in.close();
}
```
Decorator vs. Strategy, Adaptor

Decorator is sometimes called “Wrapper.”
- Change the “skin” of the object.
- Eventually, the object gets the call.

In contrast, Strategy changes the “guts” of the object.

Adaptor is sometimes called “Wrapper.”
- Used to convert a class to another class.
Adaptor Pattern

Decorator Pattern Structure: P. 177