- Apolynomial

- The Visitor Pattern Continued

- Interfaces

Overview


```java

Example

- One interface can extend another interface.

  ```java
  public interface Colorable {
    public final int getFinish();
  }
  ```

Example

- Implementations.

  ```java
  public interface Colorable {
    public void setColor(int color);
    public void setDefaultColor();
  }
  ```

- A set of methods and constant declarations, without the method

  ```java
  public int getFinish();
  ```

```

What is an interface?

---

**Declaring Interfaces**

February 18, 2000

Comp 212
Using Interfaces

Example

- An interface is a reference type, just like a class.

```java
public class Colorable {
    public Color getColor()
    public void setColor(Color color)
}
```

```java
class Red implements Colorable {
    public Color getColor()
    return Color.RED
}
```

```java
class Green {
    public Color getColor()
    return Color.GREEN
}
```

```java
class Blue extends Point {
    public Color getColor()
    return Color.BLUE
}
```

Example

- In a class definition, we say that a class implements an interface.

```
public class Point {
    public Color getColor()
    return Color.DEFAULT
}
```

- How do you use an interface?
class PatientPoint extends ColorPoint implements Paintable

- Example #2

    ...

    }

    Interface

    class MyClass implements Interface

    - Example #1

    A class can implement one or more interfaces.
The Standard Visitor Pattern

The Cof book ("Design Patterns")

The polynomial system in homework #1 can be implemented as a

The abstract polynomial, Apolyonomial, has two concrete variants,

- The Visitor framework based on the visitor pattern described in

- The algorithms are modeled as a Java interface, IVisitor, which

- The Visitor framework is nonstandard and acts as the host to its visitors.

- The object for constmap (constmap Poly, Object input) to act

2. Objecnt for nonconstmap (Nonconstmap Poly, Object input) to act

on nonconstmap objects only.

3. Objecnt for constmap (constmap Poly, Object input) to act

has exactly two methods:

- The visitor framework has two concrete variants,

The Standard Visitor Pattern
made, reducing code complexity. * Polymorphism will ensure that, at run time, the proper calls will be
passing itself as the host.
  while NonConcrete::execute(...) will call a base::ForConcrete(...)
  passing itself as the (concrete) host,
  Concrete::execute(...) will call a Con::ForConcrete(...)
  passing Object execute (Visitor: a Ge, Object input);
  A Polymorphic can execute any algorithm that is implemented as a
concrete subclass of Visitor via the abstract „hook“ method:
The Standard Visitor Pattern (cont.)
NonConstantPoly and thus can access all package private elements.

* PolyFactory resides in the same package as ConstantPoly and
  to instantiate and initialize concrete polynomial objects.
* If checks for valid input before calling on the appropriate constructors
  NonConstantPoly objects.

- A factory class, PolyFactory, is provided to build ConstantPoly and
  are package private.

- For this reason, the constructors for ConstantPoly and NonConstantPoly

  - It is good software engineering practice to shield clients from the details

Software Engineering Issues

February 18, 2000
manipulated by any client via their public behaviors. and \texttt{PolyFactory} are all public classes, they can be directly

- Since \texttt{Apartment}, \texttt{ConcretePoly}, \texttt{NonConcretePoly}, \texttt{Visitor},

  the clients to suit their needs.

- The concrete visitors are usually in different packages created by

  system without rewriting/recompiling any of the existing code.

- Any client can develop any concrete visitors to add on to the existing

  \texttt{PolyFactory} in one package to the client.

- Deliver \texttt{Apartment}, \texttt{ConcretePoly}, \texttt{NonConcretePoly}, \texttt{Visitor}, and

  \texttt{Framework} would

  - In practice, the developer of this \texttt{Polynomial/Visitor} framework would

    public in order for any concrete visitor to use them.

  - As a consequence, \texttt{Apartment} and all of its subclasses must be

  - Each of the visitor's methods explicitly specifies what concrete subclass

Software Engineering Issues (cont.)
of its methods and thus violates this principle.

In the preceding version of the polynomial/visitor framework, the visitor

of abstraction (00pp#2: Program to the (abstract) interface).

It is good software engineering practice to program at the highest level

Software Engineering Issues (cont.)
abstract host.

by making the visitor interface depend only on the
abstraction by making the visitor interface depend only on the
abstraction by making the visitor interface depend only on the
abstraction by making the visitor interface depend only on the

We can promote the standard visitor pattern to a higher level of
We can promote the standard visitor pattern to a higher level of
We can promote the standard visitor pattern to a higher level of
We can promote the standard visitor pattern to a higher level of

- they need not know about the concrete subclasses of AbstractVisitor.
- they need not know about the concrete subclasses of AbstractVisitor.
- they need not know about the concrete subclasses of AbstractVisitor.
- they need not know about the concrete subclasses of AbstractVisitor.

And since the visitors only deal with the public methods of the host,
And since the visitors only deal with the public methods of the host,
And since the visitors only deal with the public methods of the host,
And since the visitors only deal with the public methods of the host,

- * superclasses AbstractVisitor.
- * superclasses AbstractVisitor.
- * superclasses AbstractVisitor.
- * superclasses AbstractVisitor.

and NonConcretePoyt have the same public methods as their abstract
and NonConcretePoyt have the same public methods as their abstract
and NonConcretePoyt have the same public methods as their abstract
and NonConcretePoyt have the same public methods as their abstract

- We can achieve this goal because in our current design, ConcretePo
t
- We can achieve this goal because in our current design, ConcretePo
t
- We can achieve this goal because in our current design, ConcretePo
t
- We can achieve this goal because in our current design, ConcretePo
t

- This will allow us more flexibility in modifying our implementation of
- This will allow us more flexibility in modifying our implementation of
- This will allow us more flexibility in modifying our implementation of
- This will allow us more flexibility in modifying our implementation of

- clients and made package private.
- clients and made package private.
- clients and made package private.
- clients and made package private.

- We would like to hide more of the details of the implementation from
- We would like to hide more of the details of the implementation from
- We would like to hide more of the details of the implementation from
- We would like to hide more of the details of the implementation from

Software Engineering Issues (cont.)
made by the proper concrete subclass, reducing code complexity.

- Polymorphism will ensure that, at run time, the proper calls will be

  Everything else remains the same.

- on NonConcreteObject objects only.

- 2. Object for NonConcrete(A)Polymorphic host, Object input to act

  I. Object for Concrete(A)Polymorphic host, Object input to act on

  Implementations to require APolymorphic as a host instead:

  The only change we need to make is to redefine the visitor interface

A Variant of the Visitor Pattern
with modification/recompilation as well. All client code external to the polynomial package should work as before. All the pre-existing visitors for ArrayPolynomial remain intact and private constructors for ArrayPoly. All we need to do is to create a class ArrayPoly as a subclass of existing client code. By hiding the details of implementation and exposing only the abstract class ArrayPolynomial to all of its clients, in particular its visitors, we can change the implementation of ArrayPolynomial without affecting any of the client code. Example: Array Implementation

February 18, 2000

Comp 212