Overview

- Interfaces
- The Visitor Pattern Continued
  - *APolynomial*

Declaring Interfaces

- What is an interface?
  - A set of method and constant declarations, without the method implementations.
    - ∗ Example

      ```
      public interface Colorable {
        public void setColor(int color);
        public int getColor();
      }
      ```
  - One interface can *extend* another interface.
    - ∗ Example

      ```
      public interface Paintable extends Colorable {
        public static final int MATTE = 0, GLOSSY = 1;
        public void setFinish(int finish);
        public int getFinish();
      }
      ```

Using Interfaces

- How do you use an interface?
  - In a class definition, we say that a class *implements* an interface.
    - ∗ Example

```
class Point { int x, y; }

class ColoredPoint extends Point implements Colorable {
  int _color;
  public void setColor(int color) { _color = color; }
  public int getColor() { return _color; }
}
```

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

```
  Colorable widget = new ColoredPoint();
  widget.setColor(GREEN);
```

Using Interfaces (cont.)

- A class can implement one or *more* interfaces.
  - Example #1

    ```
    class MyClass implements IYourInterface1,
                             IYourInterface2 {
      . . .
    }
    ```

  - Example #2

    ```
    class PaintedPoint extends ColoredPoint implements Paintable
    {
      int _finish;
      public void setFinish(int finish) {
        _finish = finish;
      }
      public int getFinish() { return _finish; }
    }
    ```

The Standard Visitor Pattern

- The polynomial system in homework #1 can be implemented as a Polynomial/Visitor framework based on the visitor pattern described in the GoF book ("Design Patterns").
  - The abstract polynomial, `APolynomial`, has two concrete variants, `ConstPoly` and `NonConstPoly`, and acts as the *host* to its visitors.
  - The visitors implement the algorithms that operate on polynomials.
    - ∗ These algorithms are modeled as a Java interface, `IVisitor`, which has exactly two methods:
      1. `Object forConst(ConstPoly poly, Object input)` to act on `ConstPoly` objects and
      2. `Object forNonConst(NonConstPoly poly, Object input)` to act on `NonConstPoly` objects only.

The Standard Visitor Pattern (cont.)

- APolynomial can execute any algorithm that is implemented as a concrete subclass of IVisitor via the abstract "hook" method:

  ```
  Object execute (IVisitor algo, Object input).
  ```

  - ConstPoly.execute(...) will call algo.forConst(...) passing itself as the (concrete) host,
  - while NonConstPoly.execute(...) will call algo.forNonConst(...) passing itself as the host.
    - Polymorphism will ensure that, at run time, the proper calls will be made, reducing code complexity.

Software Engineering Issues

- It is good software engineering practice to shield clients from the details of correctly manufacturing concrete instances of polynomials.
  - For this reason, the constructors for `ConstPoly` and `NonConstPoly` are package private.
  - A factory class, `PolyFactory`, is provided to build `ConstPoly` and `NonConstPoly` objects.
    - It checks for valid input before calling on the appropriate constructors to instantiate and initialize concrete polynomial objects.
    - `PolyFactory` resides in the same package as `ConstPoly` and `NonConstPoly` and thus can access all package private elements.

Software Engineering Issues (cont.)

- Each of the visitor's methods explicitly prescribes what concrete subclass of `APolynomial` must be passed to it as a parameter.
  - As a consequence, `APolynomial` and all of its subclasses must be public in order for any concrete visitor to use them.
    - ∗ In practice, the developer of this polynomial/visitor framework would deliver `APolynomial`, `ConstPoly`, `NonConstPoly`, `IVisitor`, and `PolyFactory` in one package to the client.
    - ∗ Any client can develop any concrete visitors to add on to the existing system without rewriting/recompiling any of the existing code.
      - · The concrete visitors are usually in different packages created by the clients to suit their needs.
      - · Since `APolynomial`, `ConstPoly`, `NonConstPoly`, `IVisitor`, and `PolyFactory` are all public classes, they can be directly manipulated by any client via their public behaviors.

Software Engineering Issues (cont.)

- It is good software engineering practice to program at the highest level of abstraction (OOPP#2: *Program to the (abstract) interface*).

- In the preceding version of the polynomial/visitor framework, the visitor interface requires a specific concrete subclass of `APolynomial` for each of its methods and thus violates this principle.

Software Engineering Issues (cont.)

- We would like to hide more of the details of the implementation from the clients: `ConstPoly` and `NonConstPoly` should be hidden from the clients and made package private.
  - This will allow us more flexibility in modifying our implementation of `APolynomial` without changing any of the clients' code.
  - We can achieve this goal because in our current design, `ConstPoly` and `NonConstPoly` have the same public methods as their abstract superclass `APolynomial`.
    - And since the visitors only deal with the public methods of the host, they need not know about the concrete subclasses of `APolynomial`.
    - We can promote the standard visitor pattern to a higher level of abstraction by making the visitor interface depend only on the abstract host.

A Variant of the Visitor Pattern

- The only change we need to make is to redefine the visitor interface `IVisitor` and the corresponding method signatures of all of its concrete implementations to require `APolynomial` as a host instead:

  1. `Object forConst(APolynomial host, Object input)` to act on `ConstPoly` objects only, and
  2. `Object forNonConst(APolynomial host, Object input)` to act on `NonConstPoly` objects only.

- Everything else remains the same.
  - Polymorphism will ensure that, at run time, the proper calls will be made by the proper concrete subclass, reducing code complexity.

Example: Array Implementation

- By hiding the details of implementation and exposing only the abstract class `APolynomial` to all of its clients, in particular its visitors, we can change the implementation of `APolynomial` without affecting any of the existing client code.
  - For example, we can implement the polynomials using arrays.
    - ∗ All we need to do is to create a class `ArrayPoly` as a subclass `APolynomial`, and modify `PolyFactory` to make use of the package private constructors for `ArrayPoly`.
      - · All the pre-existing visitors for `APolynomial` remain intact and need not be recompiled to work with the new implementation.
      - · All client code external to the polynomial package should work with modification/recompilation as well.