A pattern: Visitor

■ Motivation:
  ■ Have a hierarchy of classes
    ■ E.g., different kinds of queries, or query results
  ■ Want to add operations to them
    ■ E.g., translating each kind of query to string, or printing out queries or evaluating queries, or ...
  ■ But can’t (or don’t want to) modify the classes to add the operations
    ■ E.g., don’t have source access, or don’t want to pollute a shared class library with application-specific operations

Participants (class library)

abstract class Thing {
    abstract void Accept(Visitor v);
}
class A extends Thing {
    void Accept(Visitor v) { v.VisitA(this); }
}
class B extends Thing {
    void Accept(Visitor v) { v.VisitB(this); }
}

Participants (visitors)

abstract class Visitor {
    abstract void VisitA(A a);
    abstract void VisitB(B b);
}
class MyVisitor extends Visitor {
    void VisitA(A a) { ... }
    void VisitB(B b) { ... }
}
...

Some examples

■ Queries: printing, translating for back-end
■ Results: printing, displaying

Benefits

■ Allows extending class hierarchies with new operations
■ Groups methods of a single operation in one place
■ Can inherit code from one visitor class to another

Liabilities

■ Obstructs adding new subclasses of library classes
  ■ Can’t get your cake and eat it too
■ Arguments & results of all operations have to be the same
■ Can’t access private stuff from visitor
■ Must plan ahead a little
■ Somewhat tedious to program
An alternative: external methods in MultiJava!

... Thing.MyOperation(...) {
    ... // default behavior of MyOperation
}  
... A.MyOperation(...) {
    ... // behavior for A's
}  
... B.MyOperation(...) {
    ... // behavior for B's
}

Benefits of external methods

- Easy to add new operations to existing classes
- Also groups related methods together
- No need to plan ahead for visitation
- Each operation can have its own argument and result types
- No obstruction of subclassing

Liabilities of external methods

- Need a language extension
- No inheritance from one "visitor" to another
- Still can't access private stuff
- Some restrictions imposed to ensure modular safety & compilability
  - So use Relaxed MultiJava!

A pattern: Abstract Factory

- Motivation: want to decouple a client that creates objects from exactly what class is created
  - Allow changing what class is created without modifying the instantiating clients
  - Allow parameterizing clients by different implementations of some abstract interfaces (e.g. GUI elements)
  - "Virtual constructors"

Participants (items)

interface A { ... }
interface B { ... }

class MyA1 implements A { ... }
class MyB1 implements B { ... }

class YourA2 implements A { ... }
class YourB2 implements B { ... }

Participants (factories)

abstract class AbstractFactory {
    abstract A createA(...);
    abstract B createB(...); ... }

class MyFactory1 extends AbstractFactory {
    A createA(...) { return new MyA1(...); }
    B createB(...) { return new MyB1(...); } ... }

class YourFactory2 extends AbstractFactory {
    A createA(...) { return new YourA2(...); }
    B createB(...) { return new YourB2(...); } ... }
Participants (clients)

class Client {
    private AbstractFactory factory;
    public Client(..., AbstractFactory f) {
        ... factory = f; }
    ...
    A anA = factory.createA(...);
    B abB = factory.createB(...);
    ...
    Client c = new Client(..., new MyFactory1());
    // or new Client(..., new YourFactory2());
}

Some examples

- Changing visual "look and feel" of query objects
  - Without rewriting clients

- Replacing original classes with enhanced or adapted subclasses
  - Without rewriting clients

Benefits

- Can swap different implementations of interface without affecting clients

Liabilities

- More cumbersome creation protocol
- Clients must not invoke regular constructors
  - How to protect them?
- Obstructs adding new kinds of items to be created
  - Analogous to visitor limitations
    - Analogous MultiJava solution?

More design patterns

- Singleton: classes with a single instance
- Prototype: create objects by copying prototypical instances
- Proxy: a forwarding object
- Chain of Responsibility: a sequence of objects that might handle operations
- Strategy: interchangeable algorithms
- State: (appear to) change an object's class
- Mediator: a coordinator object that knows how other objects should interact