

The Expression Problem Revisited

Mads Torgersen
University of Aarhus

Overview

- The problem
- Terminological Framework
- Solutions with Java and C# generics

The Expression Problem

	class1	class2	class3
op1	X	X	X
op2	X	X	X
op3	X	X	X

The Expression Problem

	class1	class2	class3	class4
op1	X	X	X	X
op2	X	X	X	X
op3	X	X	X	X

The Expression Problem

	class1	class2	class3	class4
op1	X	X	X	X
op2	X	X	X	X
op3	X	X	X	X
op4	X	X	X	?

Data-centered approach

```
interface Exp {  
    void print();  
}  
  
class Lit implements Exp {  
    public int value;  
    public void print() { System.out.print(value); }  
}  
  
class Add implements Exp {  
    public Exp left, right;  
    public void print() {  
        left.print(); System.out.print("+"); right.print();  
    }  
}
```

Data-centered approach

```
interface Exp {  
    void print();  
}
```

```
class Lit implements Exp {  
    public int value;  
    public void print()  
}
```

```
class Add implements Exp {  
    public Exp left, right;  
    public void print()  
        left.print(); System.out.print("+");  
        right.print();  
}
```

```
class Neg implements Exp {  
    public Exp exp;  
    public void print() {  
        System.out.print("-(");  
        exp.print();  
        System.out.print(")");  
    }  
}
```

Data-centered approach

```
interface Exp {  
    void print();  
}
```

```
class Lit implements Exp {  
    public int value;  
    public void print()  
}
```

```
class Add implements Exp {  
    public Exp left,  
    public void print()  
        left.print();  
}
```

```
class Neg implements Exp {  
    public Exp exp;  
    public void print() {  
        System.out.print("-(");
```

```
eval() ?
```

Operation-centered approach

```
interface Visitor {  
    void visitLit(Lit lit);  
    void visitAdd(Add add);  
}  
  
class Print implements Visitor {  
    public void visitLit(Lit lit) {  
        System.out.print(lit.value);  
    }  
    public void visitAdd(Add add) {  
        add.left.accept(this);  
        System.out.print("+");  
        add.right.accept(this);  
    }  
}
```

Operation-centered approach

```
interface Visitor {  
    void visitLit(Lit lit);  
    void visitAdd(Add add);  
}
```

```
class Print implements Visitor {  
    public void visitLit(Lit lit) {  
        System.out.println(lit.value);  
    }  
    public void visitAdd(Add add) {  
        add.left.accept(this);  
        System.out.println(" + ");  
        add.right.accept(this);  
    }  
}
```

```
class Eval implements Visitor {  
    public int result;  
    public void visitLit(Lit lit) {  
        result = lit.value;  
    }  
    public void visitAdd(Add add) {  
        add.left.accept(this);  
        int tmp = result;  
        add.right.accept(this);  
        result += tmp;  
    }  
}
```

Operation-centered approach

```
interface Visitor {  
    void visitLit(Lit lit);  
    void visitAdd(Add add);  
}
```

```
class Print implements Visitor {  
    public void visitLit(Lit lit) {  
        System.out.println(lit.value);  
    }  
    public void visitAdd(Add add) {  
        add.left.visit(this);  
        System.out.println(" + ");  
        add.right.visit(this);  
    }  
}
```

```
class Eval implements Visitor {  
    public int result;  
    public void visitLit(Lit lit) {  
        result = lit.value;  
    }  
    public void visitAdd(Add add) {  
        add.left.visit(this);  
        result = add.left.result + add.right.result;  
    }  
}
```

Neg ?

Solutions

- A combination of
 - programming language
 - an implementation in that language
 - a discipline for extension
- So that new types and operations can be added
 - anytime
 - without modifying existing code
 - without replicating business logic
 - handling all combinations of data and operations

Extensibility

- What is it that is not changed?
 - Source code
 - Compiled code
 - Running objects
- Surrounding code
 - creation code
 - client code

State of the art

- Source-level extensibility
 - Generative approaches
- Code-level extensibility
 - Statically type safe
 - Both data-centered and operation-centered
- Object-level extensibility
 - Either statically unsafe or
 - constrain mutations

Genericity-based solutions

- Source-level extensibility

- Generative approaches

- Code-level extensibility

- Statically type safe
- Both data-centered and operation-centered

- Object-level extensibility

- Either statically unsafe or
– constrain mutations

Standard genericity

Runtime genericity

Wildcards

Solution 1

- Data-centered approach
- Adding datatypes easy
- Adding operations hard

```
interface Exp {  
    void print();  
}  
  
class Add implements Exp {  
    public Exp left, right;  
    public void print() {  
        left.print();  
        ...;  
        right.print();  
    }  
}
```

Solution 1

- Idea: Use subclassing

```
interface Exp {  
    void print();  
}
```

```
class Add implements Exp {  
    public Exp left, right;  
    public void print() {
```

```
interface EvalExp extends Exp {  
    int eval();  
}
```

```
class EvalAdd extends Add implements EvalExp {  
    public int eval() {  
        return left.eval()+right.eval();  
    }  
}
```

Solution 1

- Type error!

```
interface Exp {  
    void print();  
}  
  
class Add implements Exp {  
    public Exp left, right;  
    public void print() {
```

```
interface EvalExp extends Exp {  
    int eval();  
}  
  
class EvalAdd extends Add implements EvalExp {  
    public int eval() {  
        return left.eval()+right.eval();  
    }  
}
```

Using F-bounded genericity

```
interface Exp<C extends Exp<C>> {  
    void print();  
}  
  
class Add<C extends Exp<C>> implements Exp<C> {  
    public C left, right;  
    public void print() {  
        left.print();  
        ...;  
        right.print();  
    }  
}
```

Using F-bounded genericity

```
interface Exp<C extends Exp<C>> {  
    void print();  
}
```

```
class Add<C extends Exp<C>> implements Exp<C> {  
    public C left, right;  
    public void print() {
```

```
interface EvalExp<C extends EvalExp<C>> extends Exp<C> {  
    int eval();  
}
```

```
class EvalAdd<C extends EvalExp<C>>  
    extends Add<C> implements EvalExp<C> {  
    public int eval() {  
        return left.eval()+right.eval();  
    }  
}
```

Fixing the classes

```
interface Exp<C extends Exp<C>> {  
    void print();  
}
```

```
class Add<C extends Exp<C>> implements Exp<C> {  
    public C left, right;  
    public void print() {  
        left.print();  
        ...;  
        right.print();  
    }  
}
```

```
interface ExpFix extends Exp<ExpFix>  
{  
}
```

```
class AddFix extends Add<ExpFix> implements Exp<ExpFix>  
{  
}
```

Surrounding code

- Creation code
 - Use abstract factories to parameterize
- Client code
 - Use parameterized methods

Solution 2

- Operation centered
- Adding operations easy
- Adding datatypes hard

```
interface Visitor {  
    void visitLit(Lit lit);  
    void visitAdd(Add add);  
}  
  
class Print  
    implements Visitor ...
```

Solution 2

- Use subclassing of Visitors

```
interface Visitor {  
    void visitLit(Lit lit);  
    void visitAdd(Add add);  
}
```

```
class Print  
    implements Visitor ...
```

```
interface NegVisitor extends Visitor {  
    void visitNeg(Neg neg);  
}
```

```
class NegPrint extends Print {  
    public void visitNeg(Neg neg) { ... }
```

Solution 2

- Type error!

```
class Neg implements Exp {
    public Exp exp;
    void accept(Visitor v) {
        v.visitNeg(this);
    }
}
```

```
interface Visitor {
    void visitLit(Lit lit);
    void visitAdd(Add add);
}
```

```
class Print
    implements Visitor ...
```

```
class NegPrint extends Visitor {
    visitNeg(Neg neg);
}
```

```
class NegPrint extends Print {
    public void visitNeg(Neg neg) { ... }
```

Using genericity

```
interface Exp<V extends Visitor> {  
    void accept(V v);  
}
```

```
class Add<V extends Visitor> implements Exp<V> {  
    public Exp<V> left, right;  
    public void accept(V v) {  
        v.visitAdd(this);  
    }  
}
```

```
class Neg<V extends NegVisitor> implements Exp {  
    public Exp<V> exp;  
    void accept(V v) {  
        v.visitNeg(this);  
    }  
}
```

Using genericity

```
interface Exp<V extends Visitor> {  
    void accept(V v);  
}
```

```
class Add<V extends Visitor> implements Exp<V> {  
    public Exp<V> left, right;  
    public void accept(V v) {  
        v.visitAdd(this);  
    }  
}
```

```
class Neg<V extends NegVisitor> implements Exp {  
    public Exp<V> exp;  
    void accept(V v) {  
        v.visitNeg(this);  
    }  
}
```

More greasy type
issues glossed over!

Solution 3

- Use Java Wildcards
- Super-bounds allow *new* datastructures to contain *old* ones => object-level extensibility
- Structure-mutating operations impaired

Solution 3

```
interface Exp<V extends Visitor> {  
    void accept(V v);  
}
```

```
class Add<V extends Visitor> implements Exp<V> {  
    public Exp<? super V> left, right;  
    public void accept(V v) {  
        v.visitAdd(this);  
    }  
}
```

```
class Neg<V extends NegVisitor> implements Exp {  
    public Exp<? super V> exp;  
    void accept(V v) {  
        v.visitNeg(this);  
    }  
}
```

Solution 4

- Combine benefits of data-centered and operation-centered
- Operations are both methods and visitors

Hybrid approach

- New datatypes
 - implement old operations as methods
 - provide extended visitor interface for future operations to implement
- New operations
 - are implemented for old datatypes with visitors
 - provide extended datatype interface with new methods for future datatypes to implement

Runtime dispatch

- Must implement runtime dispatch itself
- Not statically type safe!
- Becomes more elegant with runtime genericity

Conclusions

- Standard genericity gets you a long way
- Runtime genericity helps
- Wildcards *really* help