The Expression Problem

Prof. Dr. Ralf Lämmel
Universität Koblenz-Landau
Software Languages Team
Suppose you have some **data variants** (say “apples” and “oranges”) and you have some **operations** on such data (say “drink” and “squeeze”), how would you go about the design of data and operations so that you can hopefully add new data variants and new operations later on?
Expression problem: Why the name?

• Consider such data:
  – Expressions as in programming languages:
    – Literals
    – Addition
    – ...

• Consider such operations:
  – Pretty print expressions
  – Evaluate expressions
  – ...

• Consider such extension scenarios:
  – Add another expression form
    • Cases for all existing operations must be added.
  – Add another operation
    • Cases for all existing expression forms must be added.

The name goes back to Phil Wadler who defined the expression problem in an email sent to mailing lists and individuals in November 1998.
Expression problem: What problem?

- In basic OO programming:
  - It is *easy* to add new data variants.
  - It is *hard* to add new operations.
Data extensibility based on simple inheritance

• class `Expr`: The base class of all expression forms
  • Virtual methods:
    • method `prettyPrint`: operation for pretty-printing
    • method `evaluate`: operation for expression evaluation
  • Subclasses:
    • class `Lit`: The expression form of "literals"
    • class `Add`: The expression form of "addition"
```java
/**
 * The base class of all expression forms
 */
public abstract class Expr {

    /**
     * Operation for pretty printing
     */
    public abstract String prettyPrint();

    /**
     * Operation for expression evaluation
     */
    public abstract int evaluate();
}
```

Beware of code bloat!
/**
* The expression form of "literals" (i.e., constants)
*/

public class Lit extends Expr {

    private int info;
    public int getInfo() { return info; }
    public void setInfo(int info) { this.info = info; }

    public String prettyPrint() {
        return Integer.toString(getInfo());
    }

    public int evaluate() {
        return getInfo();
    }
}

Beware of code bloat!
/**
 * The expression form of "addition"
 */

public class Add extends Expr {

    private Expr left, right;
    public Expr getLeft() { return left; }
    public void setLeft(Expr left) { this.left = left; }
    public Expr getRight() { return right; }
    public void setRight(Expr right) { this.right = right; }

    public String prettyPrint() { ... }

    public int evaluate() {
        return getLeft().evaluate() + getRight().evaluate();
    }
}

That is, another data variant but with the same operations.

Beware of code bloat!
/**
 * The expression form of "negation"
 */
public class Neg extends Expr {

    private Expr expr;
    public Expr getExpr() { return expr; }
    public void setExpr(Expr expr) { this.expr = expr; }

    public String prettyPrint() {
        return "-(" + getExpr().prettyPrint() + ")";
    }

    public int evaluate() {
        return -getExpr().evaluate();
    }
}

Beware of code bloat!

That is, yet another data variant but, again, with the same operations.
Expression problem: What problem?

• In basic OO programming:
  – It is easy to add new data variants.
  – It is hard to add new operations.
• In OO programming with instance-of / cast:
  – It is easy to add new operations.
  – It is easy to add new data variants.

But beware of programming errors
Full (?) extensibility based on instance-of tests and type casts

- We use `instanceof` tests and `casts` to dispatch functionality on data.
- We use a specific exception and `try-catch` blocks to extend operations.
- We encapsulate operations in objects so that extensions are self-aware.
- This is approach is **weakly typed**.
- In particular, there is no guarantee that we have covered all cases.
public abstract class Expr {}

public class Lit extends Expr {
    private int info;
    public int getInfo() { return info; }
    public void setInfo(int info) { this.info = info; }
}

public class Add extends Expr {
    private Expr left, right;
    public Expr getLeft() { return left; }
    public void setLeft(Expr left) { this.left = left; }
    public Expr getRight() { return right; }
    public void setRight(Expr right) { this.right = right; }
}
public class EvaluatorBase {

    public int evaluate(Expr e) {
        if (e instanceof Lit) {
            Lit l = (Lit)e;
            return l.getInfo();
        }
        if (e instanceof Add) {
            Add a = (Add)e;
            return evaluate(a.getLeft()) + evaluate(a.getRight());
        }
        throw new FallThroughException();
    }

}
public class EvaluatorExtension extends EvaluatorBase {

    public int evaluate(Expr e) {
        try {
            return super.evaluate(e);
        }
        catch (FallThrouhException x) {
            if (e instanceof Neg) {
                Neg n = (Neg)e;
                return -evaluate(n.getExpr());
            }
            throw new FallThrouhException();
        }
    }
}
Expression problem: What problem?

- In basic OO programming:
  - It is easy to add new data variants.
  - It is hard to add new operations.
- In OO programming with visitors:
  - It is easy to add new operations.
  - It is hard to add new data variants.
- In OO programming with instance-of / cast:
  - It is easy to add new operations.
  - It is easy to add new data variants.
Operation extensibility based on visitors

PREVIEW

- class Expr: The base class of all expression forms
  - Subclasses *as before*.
- Virtual methods:
  - Accept a visitor ("apply an operation")
- Visitors:
  - class PrettyPrinter: operation for pretty-printing
  - class Evaluator: operation for expression evaluation
The Visitor Pattern
A book recommendation
Elevator speech

Q: How can we add functionality to a given (perhaps even closed) class hierarchy such that behavior can vary across concrete classes?

A: Represent an operation to be performed on the elements of an object structure in a class separate from the given classes.
Adding an operation to lists

interface List {}
class Nil implements List {}
class Cons implements List {
    int head;
    List tail;
}

Now suppose we need functionality to compute the sum of a list.
1st attempt:
instance-of and type cast

List l = ...;
int sum = 0; // contains the sum after the loop
boolean proceed = true;
while (proceed) {
    if (l instanceof Nil)
        proceed = false;
    else if (l instanceof Cons) {
        sum = sum + ((Cons) l).head; // Type cast!
        l = ((Cons) l).tail; // Type cast!
    }
}
1st attempt: 
instance-of and type cast

Casts are evil!

Requirement: 
static type checking

interface List {
    int sum();
}
class Nil implements List {
    public int sum() { return 0; }
}
class Cons implements List {
    int head;
    List tail;
    public int sum() {
        return head + tail.sum();
    }
}
2nd attempt: virtual methods

Patching is evil!

Requirement: operation extensibility

3rd attempt: the visitor pattern

Part I: An interface for operations

```java
interface Visitor {
    void visitNil(Nil x);
    void visitCons(Cons x);
}
```
3rd attempt:
the visitor pattern

interface List {
    void accept(Visitor v);
}
class Nil implements List {
    public void accept(Visitor v) {
        v.visitNil(this);
    }
}
class Cons implements List {
    int head;
    List tail;
    public void accept(Visitor v) {
        v.visitCons(this);
    }
}
Part III:
A concrete visitor

class SumVisitor implements Visitor {
  int sum = 0;
  public void visitNil(Nil x) {}
  public void visitCons(Cons x){
    sum += x.head;
    x.tail.accept(this);
  }
}
3rd attempt: 
the visitor pattern

Demo

```java
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```
### Summary so far

<table>
<thead>
<tr>
<th></th>
<th>Frequent type casts?</th>
<th>Frequent recompilation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instanceof and type casts</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Dedicated methods</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>The Visitor pattern</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

What if we need a new class?
Summary so far

• Class hierarchy implements virtual `accept`.
• A `Visitor` object has a visit method for each class.
• Double dispatch:
  • `accept` method takes a specific `Visitor` object.
  • `accept` redirects to class-specific `visit` method.

Two levels of polymorphism
The Visitor pattern: generic solution

Applicability

- An object structure contains different classes of objects with different interfaces and operations on these objects depend on their concrete classes.

- Many unrelated operations need to be performed on the objects and you want to keep related operations together.

- The classes defining the object structure rarely change, but you want to define new operations over the structure. (If the object structure classes change often, it is better to define the operations in those classes.)
Operation extensibility based on visitors

- class *Expr*: The base class of all expression forms
  - Subclasses as before.
  - Virtual methods:
    - Accept a visitor (“apply an operation”)
  - Visitors:
    - class *PrettyPrinter*: operation for pretty-printing
    - class *Evaluator*: operation for expression evaluation

*Operation* extensibility
public abstract class Expr {

    /*
     * Accept a visitor (i.e., apply an operation)
     */
    public abstract <R> R accept(Visitor<R> v);
}
Intermezzo: visitor types

• **Returning visitors**
  - Visit methods have a return type.

• **Void visitors**
  - Visit methods are void.

In principle, void visitors are sufficient because one can always (and often has to) aggregate the result through state which is ultimately to be returned through a designated getter.
/**
 * A concrete visitor describes a concrete operation on expressions.
 * There is one visit method per type in the class hierarchy.
 */

public abstract class Visitor<R> {

    public abstract R visit(Lit x);
    public abstract R visit(Add x);
    public abstract R visit(Neg x);

}
/**
 * The expression form of "literals" (i.e., constants)
 */
public class Lit extends Expr {

    private int info;
    public int getInfo() { return info; }
    public void setInfo(int info) { this.info = info; }

    public <R> R accept(Visitor<R> v) {
        return v.visit(this);
    }
}
/**
   * The expression form of "addition"
   */
public class Add extends Expr {

    private Expr left, right;
    public Expr getLeft() { return left; }
    public void setLeft(Expr left) { this.left = left; }
    public Expr getRight() { return right; }
    public void setRight(Expr right) { this.right = right; }

    public <R> R accept(Visitor<R> v) {
        return v.visit(this);
    }
}

Another visitor-enabled data variant.

Beware of code bloat!
/**
 * The expression form of "negation"
 */
public class Neg extends Expr {

    private Expr expr;
    public Expr getExpr() { return expr; }
    public void setExpr(Expr expr) { this.expr = expr; }

    public <R> R accept(Visitor<R> v) {
        return v.visit(this);
    }
}

---

Beware of code bloat!

Yet another visitor-enabled data variant.
/**
 * Operation for pretty printing
 */

public class PrettyPrinter extends Visitor<String> {
    public String visit(Lit x) {
        return Integer.toString(x.getInfo());
    }
    public String visit(Add x) {
        return x.getLeft().accept(this) + " + " + x.getRight().accept(this);
    }
    public String visit(Neg x) {
        return "- (" + x.getExpr().accept(this) + ")";
    }
}

Beware of code bloat!
Another operation represented as a concrete visitor.

```java
/**
 * Operation for expression evaluation
 */
public class Evaluator extends Visitor<Integer> {
    public Integer visit(Lit x) {
        return x.getInfo();
    }
    public Integer visit(Add x) {
        return x.getLeft().accept(this) + x.getRight().accept(this);
    }
    public Integer visit(Neg x) {
        return - x.getExpr().accept(this);
    }
}
```

Beware of code bloat!
101 implementations

- javaComposition
- javaStatic
- javaInheritance
- javaVisitor
Summary

- Try to modularize (say, separate concerns).
- Try to understand extensibility requirements.
- Avoid modularity exorcism.
- Take refactoring and testing very seriously.

Please don’t print the slides, or print resource-friendly.
Everything beyond this point is “for your information” only.
The Expression Problem
Cont’d
The notion of extensibility

• The initial system $I$:
  • Provides operations $o_1, \ldots, o_m$.
  • Handles data variants $d_1, \ldots, d_n$.
• Consider any number of extensions $e_1, \ldots, e_k$:
  – $e_i$ could be a data extension:
    • Introduce a new data variant.
    • Provide cases for all existing operations.
  – $e_i$ could be an operation extension:
    • Introduce a new operation.
    • Provide cases for all existing data variants.
  – Any extension can be factored into such primitive ones.
• The fully extended system $e_k (\ldots (e_1 (I)))$
More on extensibility

• **Code-level extensibility:**
  – Extensions can be modeled as code units.
  – Existing code units are never edited or cloned.

• **Separate compilation:**
  – The extensions are compiled separately.

• **Statically type-checked extensibility:**
  – The extension are statically type-checked separately.
Beyond Java

• Functions and pattern matching à la SML, Haskell
• Multi-file predicates in Prolog
• Partial classes in C#
• ...

A C# approach to extensibility

• Use the partial class mechanism of C#
  • A class can be “declared” many times.
  • The list of members is accumulated.
public abstract partial class Expr {
}

public partial class Lit : Expr {
    public int info;
}

public partial class Add : Expr {
    public Expr left, right;
}
The evaluation slice

```csharp
public abstract partial class Expr {
    public abstract int Evaluate();
}

public partial class Lit {
    public override int Evaluate() {
        return info;
    }
}

public partial class Add {
    public override int Evaluate() {
        return left.Evaluate() + right.Evaluate();
    }
}
```
The pretty-print slice

```csharp
public abstract partial class Expr {
    public abstract string PrettyPrint();
}
public partial class Lit {
    public override string PrettyPrint() {
        return info.ToString();
    }
}
public partial class Add {
    public override string PrettyPrint() {
        return left.PrettyPrint()
        + " + "
        + right.PrettyPrint();
    }
}
```
A data extension

```csharp
public partial class Neg : Expr {
    public Expr expr;
}

public partial class Neg {
    public override string PrettyPrint() {
        return "- (" + expr.PrettyPrint() + ")";
    }
}

public partial class Neg {
    public override int Evaluate() {
        return -expr.Evaluate();
    }
}
```
Discussion of the C# approach

• Code extensibility only
  • All extension need to be compiled together.
  • Slices are not even checked independently.
A Haskell approach to operation extensibility

- data type \( \text{Expr} \): The union of all expression forms
  - Public constructors:
    - \( \text{Lit} \): The expression form of "literals"
    - \( \text{Add} \): The expression form of "addition"
  - Functions defined by pattern matching on \( \text{Expr} \):
    - function \( \text{prettyPrint} \): operation for pretty-printing
    - function \( \text{evaluate} \): operation for expression evaluation

\[ \text{Operation} \text{ extensibility} \]
Algebraic data types of Haskell are closed!

module Data where

-- A data type of expression forms

data Exp = Lit Int | Add Exp Exp

2 constructor components
module PrettyPrinter where

import Data

-- Operation for pretty printing

prettyPrint :: Exp -> IO ()
prettyPrint (Lit i) = putStrLn (show i)
prettyPrint (Add l r) = do prettyPrint l; putStrLn " + "; prettyPrint r

A new module can be designated to each new operation.

Operation defined by case discrimination
module Evaluator where

import Data

-- Operation for expression evaluation

evaluate :: Exp -> Int
evaluate (Lit i)   = i
evaluate (Add l r) = evaluate l + evaluate r
Discussion of the Haskell approach

- Lack of encapsulation?
- Only operation extensibility
A Prolog approach to extensibility

• Define operations as predicates.

• Model data variants through functors.

• Achieve extensibility through “multi-file” predicates.
The initial program

```prolog
:- multifile prettyPrint/2.

prettyPrint(lit(I),S) :-
    string_to_atom(S,I).

prettyPrint(add(L,R),S) :-
    prettyPrint(L,S1),
    prettyPrint(R,S2),
    format(string(S),"~s + ~s", [S1,S2]).
```
A data extension

:- multifile prettyPrint/2.
:- multifile evaluate/2.

prettyPrint(neg(E),S)
  :- prettyPrint(E,S1),
     format(string(S),"- (~s)",[S1]).

evaluate(neg(E),I)
  :- evaluate(E,I1),
     I is - I1.
An operation extension

```
:- multifile evaluate/2.

evaluate(lit(I),I).

evaluate(add(L,R),I) :-
    evaluate(L,I1),
    evaluate(R,I2),
    I is I1 + I2.
```
Program composition

:- ['InitialProgram.pro'
    , 'OperationExtension.pro'
    , 'DataExtension.pro'
    ].
Discussion of the Prolog approach

• Full extensibility

• No descriptive overhead

• Lack of static typing

• Performance penalty
Solutions of the expression problem

- Open classes in more dynamic languages (Smalltalk, etc.)
- Extensible predicates in Prolog (code-level extensibility only)
- Java, C# & Co.
  - Clever encodings (Torgersen, ECOOP 2004)
  - AOP-like Open classes (introductions)
  - .NET-like partial classes (code-level extensibility only)
  - Expanders (Warth et al., OOPSLA 2006)
  - JavaGI (Wehr et al., ECOOP 2007)
  - ...

Summary on extensibility

- **Wanted**
  - Data extensibility **and**
  - Operation extensibility **and**
  - Soft or strong static type checking **and**
  - Separate compilation/deployment **and**
  - Unanticipated extension **and**
  - Configurability and Composability **and**
  - Efficiency ("No distributed fat")