

The Expression Problem

Gina Häussge

January 12, 2006

Abstract

In this essay a short introduction to the expression problem will be given: Definition, varieties and also a couple of implementations solving it. It is based on two papers by Mads Torgersen [Torgersen] and Matthias Zenger and Martin Odersky [Zenger/Odersky], outlining and summarizing three of their solutions.

1 The Problem

A quite common situation in programming is the need to represent a recursive data structure consisting of a number of related classes and operations defined upon them, e.g. a set of arithmetic expressions. This is achieved by using the *Composite design pattern*[GoF]. There are mainly two approaches in structuring such data internally, the data-centered and the operation-centered approach.

In the data-centered approach, each operation is defined as a virtual method in the base class of the data, and then overwritten or rather defined in each of the specific data classes. The advantage of this approach is the ease of adding new data classes, as only the base class has to be extended and the necessary operations need to be implemented. On the down side it is rather expensive to add new operations, as they have to be defined in each already existing data class as well, therefore making changes to existing code a necessity.

On the other hand we have the operation-centered approach which utilizes the *Visitor design pattern*[GoF]: For each defined operation a visitor class is created, implementing a handler method for each data class available. Contrary to the data-centered approach here it is easy to add new operations by simply defining a new visitor and implementing the handlers, but quite complicated to add new data classes, as this makes it necessary to extend all already existing visitor classes in order to add a handler for the new data class.

We see: Upon adding new data types or operations to the structure, one is faced with the *expression problem*: Depending on design and organisation of the code, it is usually either simple to add new data types or new operations, but always rather hard to add the other kind. There are already many solutions to this problem, but most of them break one of the following criteria which - according to [Torgersen] and [Zenger/Odersky] - define a valid solution to the expression problem:

- Make it possible both to extend the data model and the operations defined on it in an unlimited way, even combining independent extensions,
- without the need to modify or repeat existing code and while conserving a strong type safety and the usability of old data structures even after the modification

We now want to present three of the solutions introduced by the three authors, two proposed by Torgersen, one by Zenger/Odersky.

2 Selected Solutions

2.1 Torgersens data-centered approach

As we already know, adding new data-classes to an existing data-centered approach is not difficult, problems only arise when adding new operations to the existing implementation. Given an implementation which utilizes a basic interface which specifies defined operations and which is implemented by all defined data-classes, in order to add a new operation we would have to extend said interface and reimplement all data-classes. As we are not allowed to modify existing sourcecode, a problem arises as soon as we need to access inherited variables, which still are of the unextended type and therefore do not provide the newly defined methods. We cannot adjust the existing basic interface. A solution is an implementation based on generics, achieving a parameterisation of the expressions. Extensions then restrict the allowed types as necessary and therefore ensure the usage of the newly defined base when accessing old code.

In an example given by Torgersen we start with an implementation of basic arithmetic expression consisting only of an add operation and a literal, which then gets extended to allow a new “eval” operation on it, which simply returns the integer value of an expression. Torgersen creates a new extended Interface from the original “Exp” called “EvalExp” with the defined method “eval”, new subclasses of “Add” and “Lit” called “EvalAdd” and “EvalLit” and then restricts the allowed types in the parameterisation to subclasses of the new basic Interface “EvalExp” (as opposed to the former “Exp”).

A type-safe data-centered code-level-extensible solution is the result.

2.2 Torgersens operation-centered approach

As already hinted at, when adding new data-types to an operation-centered implementation we have a dual problem to the addition of a new operation to a data-centered approach. Again we are faced with a type checking problem, this time in the accept method of the implemented *visitor*.

But this time parameterisation of the expressions is not working alone. As we see in the code snippet provided, a problem is encountered in the `visitAdd` method of the `AlePrint` visitor, as the code doesn’t know that `this` is of the `V` type. A type error is raised. To circumvent this, the visitor to use is simply added as a parameter to the `visitAdd` method, therefore making it clear that it is of the requested type `V`. After this,

Algorithm 1 Torgersens operation-centered approach: Unsolved issues

```
class Add(V extends AleVisitor) implements Exp<V> {
    public Exp<V> left, right;
    public void accept(AleVisitor v) {
        v.visitAdd(this);
    }
}
class AlePrint implements AleVisitor {
    ...;
    public <V extends AleVisitor>
    void visitAdd(Add<V> add) {
        add.left.accept(this);
        System.out.print('+');
        add.right.accept(this);
    }
}
```

an extension to allow a new data type is a simple matter of creating the new data class and subclassing the defined visitors to include a new data type handler.

2.3 Zenger/Oderskys data-centered approach

Last but not least we also want to summarise a solution proposed by Zenger and Odersky. We take a look at their data-centered approach. It is suggested to also take a look at the programming language Scala in order to understand the mechanics of the described solution, especially at the nature of *traits*[Scala].

Algorithm 2 Zenger/Oderskys data-centered approach: Addition of a new operation

```
trait Show extends Base {
    type exp <: Exp;
    trait Exp extends super.Exp {
        def show: String;
    }
    class Num(v: int) extends super.Num(v) with Exp {
        def show = value.toString();
    }
}
```

In order to add a new operation (in this case a “show” operation) to an existing design, we first have to create a new trait, extending the base trait. The newly introduced method also needs to be implemented in all defined data-types, therefore we need to subclass all of them and define the “show” operation - compared to the Java based solutions this is rather uncomplicated thanks to the nature of Scala.

To combine existing independent operation extensions, we need to do a *deep mixin composition* (as opposed to the simpler *mixin composition* needed for combining independent data extensions): One of the traits is extended and the information of the other traits is mixed in. Again we see that one type of addition - in this case the addition of an operation to a data-centered approach - is more expensive than another, due to the dual nature of the expression problem. Thanks to the design of Scala the costs aren't as high as with other languages though.

3 Conclusion

Torgersen and Zenger/Odersky both propose a number of solutions to the described problem. Torgersen depends on new features of the Java programming language, generics and wildcards, which are available since JDK1.5. Zenger and Odersky on the other hand use the programming language Scala, which - while being a very well defined object oriented language with great possibilities - slightly lacks the amount of real world usage of Java, and makes it hard to implement the proposed solutions in productive environments.

References

- [Torgersen] M. Torgersen. *The expression problem revisited - Four new solutions using generics.*
- [Zenger/Odersky] M. Zenger, M. Odersky. *Independently Extensible Solutions to the Expression Problem.* Technical Report IC/2004/33
- [GoF] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley 1994
- [Scala] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger. *An Introduction to Scala.* August 2005.