The Scrap Your Boilerplate Pattern in Java (Draft)

Sven Eric Panitz
`www.panitz.name`

29th March 2005

Abstract

It is shown how the scrap your boilerplate design pattern, which has been presented in the functional programming community, can easily be implemented in Java. It addresses generic programming of tree traversal. Generic methods for traversing an arbitrary tree structure are presented. The actual application logic will be clearly seperated from the boilerplate traversal code. A crucial rôle for the basic tree traversion plays the interface `Iterable`, which has been added to the latest version of the Java API .

1 Introduction

Genericity is the key to software reuse. Generic code solves general problems and can be specialized to specific problems. A general problem is the handling of hierarchical data. Almost all interesting data is hierarchical data. XML[T. 04], the universal data exchange format pays tribute to this fact. XML documents represent a hierarchical structure. In object oriented languages tree nodes are represented by different classes. Child nodes are represented as fields of these classes. Number, type and name of fields representing child nodes are arbitrary.

Quite often the classes representing tree nodes are generated by some tool, as e.g. by the parser generator tool javacc or for some XML DTD.

A common programming task for tree like structures is to traverse through the tree. Generally two different task may be archieved during traversal:

- certain tree nodes may be modified,
- or certain information may be collected from the tree.

In object oriented languages there are mainly two ways how tree traversal may be put into realization:

- in a pure object oriented manner: classes representing tree nodes contain specialized methods for different tasks.
- in a functional way by means of the visitor pattern[GHJV95]: tree classes implement an interface `Visitable`. Algorithms can be expressed by terms of visitor classes. A visitor class overloads a function for every kind of tree node.

Both ways are not generic. Both require that the tree traversal code is implemented for every node type. Even worse, in both cases the tree traversal code mixes with the actual application logic. Thus the tree traversal code gets duplicated for every algorithm working on the tree. The methods in the tree node classes or the overloaded methods `visits` in the visitor need to code, how to access the child nodes.

Furthermore, both techniques are not very flexible as in respect to program evolution. If in the object oriented scenario a new class is added, then this class needs to implement all methods, which for some reason traverse over the tree. A visitor class, on the other hand is only applicable to a certain type of tree node. It is specialized for some type. If a new node class is added to the tree type, then every visitor needs to implement a new overloaded method variant for the new class.

A generic way for tree manipulation will clearly seperate the tree traversal code and the actual application logic. Furthermore it will be very flexible in terms of programm evolution: new classes arriving on the scene will not force any changes in the existing code.

A simple pattern for tree traversal has been proposed for the programming language Haskell[PJ03]. The *scrap your boilerplate code* pattern[LP03][LP04] factors out the tree traversal code and makes it reusable for different algorithms. We will apply this pattern to the programming language Java.

The paper is structured as follows: in section 2 the problem to be solved is presented by way of a concrete example taken from [LP03]. Section 3 gives some naïv non generic solutions to the problem. Section 4 implements a generic tree transformer and section 5 implements a generic query method for trees. Both implementations are refined in section 6. The Java code within this paper is complete and can be downloaded from the web (`http://panitz.name/paper/boilerplate/index.html`) .

2 The Problem

We use the same example as presented in [LP03]. It is a simple company structure, which can be expressed by the following XML document type definition.

company.dtd
```
<!DOCTYPE Company SYSTEM "company.dtd" [
  <!ELEMENT Company (Dept)*>
  <!ELEMENT Dept (Name,Employee,(PU|DU)*)>
  <!ELEMENT PU (Employee)>
  <!ELEMENT DU (Dept)>
  <!ELEMENT Employee (Person,Salary)>
  <!ELEMENT Person (Name,Address)>
  <!ELEMENT Salary (#PCDATA)>
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Address (#PCDATA)>
]>
```

This definition gives directely rise to a Java implementation. For every element type defined in the DTD we can provide a Java class. The child elements are represented by way of fields in these classes. For the alternative operator `|` of several child elements as e.g. expressed in `(PU|DU)` a common superinterface for the classes `PU` and `DU` can be defined. For the repetitive occurrence of child elements as e.g. `(Dept)*` the standard Java interface `List` is used.

Java classes for the DTD above are given in appendix A. Usually some generator tool can be used to generate classes from a DTD or from a XML schema definition.

Throughout this paper we will work with an example instance of `Company`. It is the same example instance as used in [LP03].[1]

MyComp.java
```
package name.panitz.boilerplate;
import java.util.*;
class MyComp{
  static Company getCompany(){
   Company company= new Company();
   Employee blair= new Employee(new Person("Blair","London"),new Salary(100000.0));
   Employee nn=new Employee(new Person("NN","Mars"),new Salary(90000.0));
   Employee marlow=new Employee(new Person("Marlow","Cambridge"),new Salary(2000.0));
   Employee joost = new Employee(new Person("Joost","Amsterdam"),new Salary(1000.0));
   Employee ralf = new Employee(new Person("Ralf","Amsterdam"),new Salary(8000.0));
   List<SubUnit> ralfSu = new ArrayList<SubUnit>();
   ralfSu.add(new PU(joost));ralfSu.add(new PU(marlow));
   company.add(new Dept("Research",ralf,ralfSu));
   company.add(new Dept("Strategy",blair,new ArrayList<SubUnit>()));
   ralfSu.add(new DU(new Dept("Research2",nn,new ArrayList<SubUnit>())));
   return company;
  }
}
```

Tasks to be solved for a structure like the company above are:

- increase the salary of each employee.
- liquidate a certain sub department.
- sum up all salary numbers for an overall salary bill of the company.
- represent the company by a `String`.
- compare two companies for equality.
- represent the company in an XML document.

All these task need to travers the company. This will make it necessary to rewrite the traversal code for each of these tasks. We want traverse tree structures in a generic way, i.e. write the traversal code once for all.

The domain specific language XSLT[Wor99], the for transforming XML documents gives us some hints on how this can be achieved in Java. In XSLT rules for nodes with a certain condition can be expressed. Elements nodes, which do not match any of these conditions simply traverse to the child nodes. XSLT allows to express default templates of how to traverse for elements, which do not match any specific rule. The first of the above tasks can easily be expressed in XSLT by:

IncreaseSalary.xsl
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="Salary">
  <xsl:variable name="arg1"><xsl:apply-templates select="text()"/></xsl:variable>
  <Salary><xsl:value-of select="$arg1 * 1.17" /></Salary>
</xsl:template>
<xsl:template match="*">
  <xsl:copy select="*" ><xsl:apply-templates/></xsl:copy >
```

[1]Not quite it is extended with a subdepartment, otherwise one of the algorithms in this (as well as in the original paper) does not have any effect.

```
</xsl:template>
</xsl:stylesheet>
```

Note that this XSLT program will work on any XML document. It is not necessary that the document is structured according to some schema or DTD. It will automatically traverse the complete document and increase the numbers found within every `Salary` node. XSLT relies on the fact that XML documents are trees. The `<xsl:apply-templates/>` apllies to the direct children of the current node. We would like to get an comparatively simple solution for this task in Java.

3 Naïve hand coded instances

In the previous section it turned out that the notion of child nodes is crucial for any tree traversal. In order to result in some generic tree traversal code, we will need some uniform method of how to retrieve the direct child nodes of some node. For this we can instrumentalize Java's standard interface `Iterable`. We will use `Iterable` such that it will not iterate over all descendents of a tree node, but only its direct children. In fact this is what we did for the classes representing the `Company` DTD as can be found in appendix A. Since the type of the child nodes is arbitrary we implemented the most general instantiation of the interface: `Iterable<Object>`.

3.1 Show

With iterator over the direct child nodes, the boilerplate code allready implodes to a minimal amount. As an example we give an implementation of a generic `show` method. Thanks to Java's new *for-each* loop the tree traversal boilerplate code is just one line.

Show.java

```
package name.panitz.boilerplate;
class Show{
  static String show(Object o){
    StringBuffer result = new StringBuffer();
    if (o instanceof Iterable) {
      result.append("new "+o.getClass().getSimpleName()+"(");
      boolean first = true;
      for (Object x:((Iterable)o)){//the tree traversal code
        if (first) first=false;else result.append(",");
        result.append(show(x));
      }
      result.append(")");
    }else result.append(o);
    return result.toString();
  }
}
```

This method calculates a `String` representation for any object. If the object is iterable, we will traverse its child nodes.

TestShow.java

```
package name.panitz.boilerplate;
class TestShow{
  public static void main(String [] args){
    System.out.println(Show.show(MyComp.getCompany()));}}
```

4 Generic Transformers

In this section, we will generalize the traversal code. The actual application logic will be encapsulated in a method `transform`. We can define an interface for this method:

Transformer.java
```
package name.panitz.boilerplate;
interface Transformer{void transform(Object o);}
```

The generic tree traversal code is divided onto two methods. The first method is a generic `map` function. It iterates over the direct child nodes of the argument node and applies the transform method to these:

Transform.java
```
package name.panitz.boilerplate;
abstract class Transform  implements Transformer{
  public  void gmap(Object x){
    if (x instanceof Iterable) for (Object a:((Iterable)x)) transform(a);
  }
```

For the complete traversal of the tree, the method `everywhere` is provided. it needs an inner `Transform` object, which will be cached:

Transform.java
```
  Transform everywhereT = null;
  public void everywhere(Object x){
```

The inner `everywhere` transformer will recursively call the method `everywhere` of this transformer:

Transform.java
```
    final Transform dies = this;
    if (everywhereT==null)
      everywhereT
       = new Transform(){public void transform(Object o){dies.everywhere(o);}};
```

Now we can use the generic `map` function to recurse everywhere into the child nodes. Afterwards we can apply the actual transform method to the root node:

Transform.java
```
    everywhereT.gmap(x);
    transform(x);
  }
}
```

The technique of providing a general `map` function for the direct children of an node is not very novel, e.g. [CF92] a tree like data definition induces the compiler to generate a `map` function.

4.1 Example: increase of all salary

Now we can implement a very simple solution to the first task of increasing the salary in every salary node of the tree. We can achieve this by extending the abstract class `Transform` where we implement the method `transform` according to our needs.

Increase.java

```java
package name.panitz.boilerplate;
import java.util.*;
class Increase extends Transform{
 Double k; Increase(Double k){this.k=k;}
 public void transform(Object o){
  if (o instanceof Salary){Salary s = (Salary)o;s.amount=s.amount*(1+k);}
 }
}
```

This looks even simpler than the XSLT solution. We do not need to bother about the tree traversal code. Applying this transformer to out company is equally as easy and straightforward.[2]

TestIncrease.java

```java
package name.panitz.boilerplate;
class TestIncrease{
  public static void main(String [] args){
    Company company = MyComp.getCompany();
    for (int i=0;i<100000;i++)
      new Increase(.000017).everywhere(company);
    System.out.println(Show.show(company));}}
```

4.2 Example: flatten company structure

The second of the task stated in the beginning of this paper, was to liquidate certain subdepartments and integrate its employees into the next upper department. As we see, this is allready a rather complex application taks. Fortunatly we no longer are concerned with the tree traversal code and can concentrate on the actual task at hand:

Flatten.java

```java
package name.panitz.boilerplate;
import java.util.*;
class Flatten extends Transform{
  String deptName;
  Flatten(String deptName){this.deptName=deptName;}
  public void transform(Object o){
   if (o instanceof Dept){
     List<SubUnit> newSus=new ArrayList<SubUnit>();
     Dept d = (Dept)o;
     for (SubUnit su:d.subUnits){
       if (su instanceof DU){
         Dept du= ((DU)su).dept;
         if (du.name.equals(deptName)){
           newSus.add(new PU(du.manager));newSus.addAll(du.subUnits);
         }else newSus.add(su);
       }else newSus.add(su);
     }
     d.subUnits=newSus;
   }
  }
}
```

This new transformer can be applied to our company in the same way as the `Increase` transformer.

[2]We apply this transformer 100000 times. This is done for some rough performance measure later in the paper.

TestFlatten.java
```java
package name.panitz.boilerplate;
class TestFlatten{
  public static void main(String [] args){
    Company company = MyComp.getCompany();
    new Flatten("Research2").everywhere(company);
    System.out.println(Show.show(company));}}
```

Note, that our transformers work on any object. We do not need to know the exact tree structure or all classes involved in a company structure. Our algorithms still work, even if new classes representing new types of subdepartments are added to the company structure.

4.3 Using Reflection

In the solution above, we did not get completely rid of boilerplate code. We needed to specify the concrete class, for which the method transform is specified. This amounts in the `instanceof` construct. It is well known, how to get rid of such conditions by way of Java's reflection mechanism, e.g. in [PJ98] generic tree traversal is achieved by way of using reflection. Following [PJ98] we can implement a generic default instance of the method `transform`.

TransformReflect.java
```java
package name.panitz.boilerplate;
class TransformReflect  extends Transform{
  public void transform(Object o){
    try{getClass().getMethod("transform",o.getClass()).invoke(this,o);
    }catch(IllegalAccessException  _){}
     catch(IllegalArgumentException _){}
     catch(java.lang.reflect.InvocationTargetException _){}
     catch(NoSuchMethodException _){}
  }
}
```

This method checks the type of the argument and looks up via reflection, if a specialized version of `transform` for this class is available. Thus the transformer to increase the salary can simply be written as:

IncreaseReflect.java
```java
package name.panitz.boilerplate;
import java.util.*;
class IncreaseReflect extends TransformReflect{
 Double k; IncreaseReflect(Double k){this.k=k;}
 public void transform(Salary s){s.amount=s.amount*(1+k);}
}
```

This implementation is completely free of any boilerplate code. However, reflection is slow, very slow. As a rough measure compare execution times of the non-reflective with the reflective solution:

TestIncreaseReflect.java
```java
package name.panitz.boilerplate;
class TestIncreaseReflect{
  public static void main(String [] args){
    Company company = MyComp.getCompany();
    for (int i=0;i<100000;i++)
      new IncreaseReflect(.000017).everywhere(company);
    System.out.println(Show.show(company));}}
```

Execution of this version compared to the original version gives the following result:

```
sep@pc216-5:~/boilerplate> time java -cp classes/ name.panitz.boilerplate.TestIncrease
new Company(new Dept(Research,new Employee(new Person(Ralf,Amsterdam),new Salary
(43790.9463569525)),new ArrayList(new PU(new Employee(new Person(Joost,Amsterdam)
,new Salary(5473.868294619063))),new PU(new Employee(new Person(Marlow,Cambridge)
,new Salary(10947.736589238126))),new DU(new Dept(Research2,new Employee(new Pers
on(NN,Mars),new Salary(492648.1465157075)),new ArrayList()))))),new Dept(Strategy,
new Employee(new Person(Blair,London),new Salary(547386.8294619016)),new ArrayLis
t()))

real    0m0.912s
user    0m0.849s
sys     0m0.013s
sep@pc216-5:~/boilerplate> time java -cp classes/ name.panitz.boilerplate.TestIncreaseReflect
new Company(new Dept(Research,new Employee(new Person(Ralf,Amsterdam),new Salary
(43790.9463569525)),new ArrayList(new PU(new Employee(new Person(Joost,Amsterdam)
,new Salary(5473.868294619063))),new PU(new Employee(new Person(Marlow,Cambridge)
,new Salary(10947.736589238126))),new DU(new Dept(Research2,new Employee(new Pers
on(NN,Mars),new Salary(492648.1465157075)),new ArrayList()))))),new Dept(Strategy,
new Employee(new Person(Blair,London),new Salary(547386.8294619016)),new ArrayLis
t()))

real    0m53.016s
user    0m52.010s
sys     0m0.238s
```

5 Generic Queries

The transformers in the last section destructively updated the tree. In this section we will apply the same technique to retrieve some data from the tree. We want to state queries over the tree data and leave the tree object unchanged. What we have in mind is a simple function taking a tree and giving some query result. Since functions are not first class citizens in Java, we express a funtion by way of an interface:

Function.java
```java
package name.panitz.boilerplate;
interface Function<argType,resultType>{resultType eval(argType x);}
```

We provide a second interface for binary functions.We resisted the temptation of expressing these via currying as `Function<a,Function<b,c>>`.

Function2.java
```java
package name.panitz.boilerplate;
interface Function2<a,b,c>{c eval(a x,b y);}
```

We can now apply the same technique for queries as for transformers. We define the generic query class `Query`. It is gerneric over the result of the query. With Java's generic types this can be elegantly expressed:

Query.java
```java
package name.panitz.boilerplate;
import java.util.*;
abstract class Query<b> implements Function<Object,b>,Function2<b,b,b>{
```

Traversal is again done in two steps. First we define a generic `map` method. This method applies the query to the child nodes and results in the list of results:

Query.java
```java
  public List<b> gmap(Object x){
    List<b> result= new ArrayList<b>();
    if (x instanceof Iterable) for (Object a:((Iterable)x)) result.add(eval(a));
    return result;
  }
```

Analogously to the method `everywhere` in transformer case we define a method `everything`. It applies the same trick. It defines an inner query which traverses over the child nodes. Since we get a list of results for the child nodes, we need a function to add these up to an overall result. That is the reason why `Query` implements `Function2<b,b,b>`. This function is used to sum up the partial result to an overall result.

Query.java
```java
  public b everything(Object x){
    b result = eval(x);
    final Query<b> dies = this;
    Query<b> everythingQ = null;
    if (everythingQ==null)
      everythingQ = new Query<b>(){
          public b eval(Object o){return dies.everything(o);}
          public b eval(b x,b y) {return dies.eval(x,y);}
      };
    for (b y:everythingQ.gmap(x)) result=eval(result,y);
    return result;
  }
}
```

A concrete query can now be implemented by extension of class `Query`. Two methods need to be implemented: the method `eval`, which represents the actual query method and the binary method `eval`, which defines how to combine partial results. In the example of a salary bill this might look like:

SalaryBill.java
```java
package name.panitz.boilerplate;
class SalaryBill extends Query<Double>{
  public Double eval(Object x){
    if (x instanceof Salary) return ((Salary)x).amount;
    return 0.0;
  }
```

Note that the method needs an neutral default result.

The second method to be implemented is used to combine partial results. In our example the simple addition function will do.

SalaryBill.java
```java
   public Double eval(Double x,Double y){return x+y;}
}
```

Thus we implemented the task of summing up all salaries, without reimplementation of the tree traversal code.

TestSalaryBill.java
```java
package name.panitz.boilerplate;
class TestSalaryBill{
  public static void main(String [] args){
    System.out.println(new SalaryBill().everything(MyComp.getCompany()));
  }
}
```

5.1 Back to XML

We started this paper by giving an example as XML document type definition. Eventually we turn back to XML. The class `Query` enables us to generically write a query object, which builds an XML tree for some tree structure. We need some document builder, which we instantiate once via a factory method:

ToXML.java
```
package name.panitz.boilerplate;
import javax.xml.parsers.*;
import org.w3c.dom.*;
class ToXML extends Query<Node>{
  static Document doc;
  static {
   try{doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
   }catch (Exception e) {throw new RuntimeException(e);}
  }
```

The two eval methods are very simple. We create elements with the class name of the object. Non iterable types, especially `String` nodes result in XML text nodes:

ToXML.java
```
  public Node eval(Object o){
    if (o instanceof Iterable)
      return doc.createElement(o.getClass().getSimpleName());
    return doc.createTextNode(o.toString());
  }
```

The second method `eval` for combining partial results, inserts the second argument as child to the first argument.

ToXML.java
```
  public Node eval(Node x,Node y){x.appendChild(y);return x;}
}
```

This simple query object generically transforms any hierarchical structure into an XML document. For a simple test we generate an XML document for the company.

TestToXML.java
```
package name.panitz.boilerplate;
import org.w3c.dom.Node;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;

class TestToXML{
  public static void main(String [] args){
    System.out.println(serialize(new ToXML().everything(MyComp.getCompany())));}
```

We use the following simple method for transforming an XML node to a `String`:

TestToXML.java
```
  static public String serialize(Node doc){
    try{
      StringWriter writer = new StringWriter();
      javax.xml.transform.Transformer t
        = TransformerFactory.newInstance().newTransformer();
      t.setOutputProperty(OutputKeys.INDENT,"yes");
      t.transform(new DOMSource(doc),new StreamResult(writer));
      return writer.getBuffer().toString();
    }catch (TransformerException _){return "";}
  }
}
```

6 Controling the traversal

A close look at the result of the last example reveals that it does not build an XML document according to the DTD given in section 2. This is not a defect in our implementation of `ToXML` but due to the fact that the Java classes for the company do not exactly match this DTD. No classes for person or department names are given. These are represented directly as a `String`.

We need specialized cases for certain nodes. Furthermore we need some way to express a *cut*, i.e. some means to express, that for this node the automatic tree traversal shall stop. Therefore we write a refinement of the class `Query`. A third abstract method is added, the method `condition`. It returns true for some arbitrary node, if the tree traversal is supposed to descent into the child nodes:

QuerySome.java

```
package name.panitz.boilerplate;
import java.util.*;
abstract class QuerySome<a>
        implements Function<Object,a>,Function2<a,a,a>{
  public abstract boolean condition(Object o);
```

Now the method `gmap` only descents into the child nodes, if the condition for the current node is true:

QuerySome.java

```
  public List<a> gmap(Object x){
    List<a> result= new ArrayList<a>();
    if (condition(x) && x instanceof Iterable)
      for (Object a:((Iterable)x)) result.add(eval(a));
    return result;
  }
```

The rest of this class is implemented exactly like in the class `Query`:

QuerySome.java

```
  public a everything(Object x){
    a result = eval(x);
    final QuerySome<a> dies = this;
    QuerySome<a> everythingQ = null;
    if (everythingQ==null)
      everythingQ = new QuerySome<a>(){
       public a eval(Object o){return dies.everything(o);}
       public a eval(a x,a y) {return dies.eval(x,y);}
       public boolean condition(Object o){return dies.condition(o);}
      };
    for (a y:everythingQ.gmap(x)) result=eval(result,y);
    return result;
  }
}
```

The class `Query` could thus be implemented as the specialisation of `QuerySome` with a constant `true` condition.

Query2.java
```
package name.panitz.boilerplate;
import java.util.*;
abstract class Query2<a> extends QuerySome<a>{
  public boolean condition(Object o){return true;}
}
```

No we can write a specialized version for creation of an XML documents, which meets the special needs of the company structure. Special cases for two node classes are given. Further traversal is stopped for these nodes:

ToXML2.java
```
package name.panitz.boilerplate;
import org.w3c.dom.*;
import java.util.*;
import static name.panitz.boilerplate.ToXML.*;
class ToXML2 extends QuerySome<Node>{
  public Node eval(Object o){
    if (o instanceof Person){
      Element p = doc.createElement("Person");
      Element n = doc.createElement("Name");
      Element a = doc.createElement("Address");
      n.appendChild(doc.createTextNode(((Person)o).name));
      a.appendChild(doc.createTextNode(((Person)o).address));
      p.appendChild(n);
      p.appendChild(a);
      return p;
    }
    if (o instanceof Dept){
      Dept d =(Dept)o;
      Element result = doc.createElement("Dept");
      Element n = doc.createElement("Name");
      n.appendChild(doc.createTextNode(d.name));
      result.appendChild(n);
      result.appendChild(everything(d.manager));
      for (SubUnit su:d.subUnits) result.appendChild(everything(su));
      return result;
    }
    if (o instanceof Iterable)
      return doc.createElement(o.getClass().getSimpleName());
    return doc.createTextNode(o.toString());
  }
  public boolean condition(Object o){
    return ! (o instanceof Person||o instanceof Dept);}
  public Node eval(Node x,Node y){x.appendChild(y);return x;}
}
```

This will generate an XML document according to `company.dtd`.

TestToXML2.java
```
package name.panitz.boilerplate;
import static  name.panitz.boilerplate.TestToXML.*;
```

```java
class TestToXML2 extends ToXML2{
  public static void main(String [] args){
    System.out.println(serialize(new ToXML2().everything(MyComp.getCompany())));}}
```

7 Conclusion

The pattern presented in this paper is very simple, general and powerfull. The techniques presented in the original paper for the programming language Haskell[PJ03] could directly be adapted to an object oriented framework. Instead of higher order functions subtyping and late-binding are used. The functions `gmap`, `everywhere` and `everhting` are parameterized by overridden methods of the transform and query objects.

Things are even simpler than in the Haskell framework, since most necessary means are available: a common object type and a type instance operator.

The pattern can be implemented without using the Java's reflection mechanism. Thus it avoids performance problems concerned with reflection. Reflection can simplify the usage of the pattern, but is not recommendable for reasons of performance.

7.1 Future Work

We did not yet explore if with some minimal and optimized use of reflection, construction of trees can be implemented with adquate performance. This would enable to write generic code for the inverse functions to the transformers `ToXML` or `Show`.

In [Vis01] a system based on visitor combination is presented. It might be fruitful to apply these techniques to the transformer presented in this paper.

A Company classes

Company.java

```java
package name.panitz.boilerplate;
import java.util.*;
class Company extends ArrayList<Dept>{}
```

Dept.java

```java
package name.panitz.boilerplate;
import java.util.*;
class Dept implements Iterable<Object>{
  String name;
  Employee manager;
  List<SubUnit> subUnits;
  Dept(String name,Employee manager,List<SubUnit> subUnits){
    this.name=name;this.manager=manager;this.subUnits=subUnits;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();
    result.add(name);result.add(manager);result.add(subUnits);
    return result.iterator();
  }
}
```

SubUnit.java

```java
package name.panitz.boilerplate;
import java.util.*;
class SubUnit implements Iterable<Object>{
  public Iterator<Object> iterator(){return new ArrayList<Object>().iterator();}
}
```

PU.java
```java
package name.panitz.boilerplate;
import java.util.*;
class PU extends SubUnit{
  Employee employee;
  PU(Employee employee){this.employee=employee;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();result.add(employee);
    return result.iterator();
  }
}
```

DU.java
```java
package name.panitz.boilerplate;
import java.util.*;
class DU extends SubUnit{
  Dept dept;
  DU(Dept dept){this.dept=dept;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();result.add(dept);
    return result.iterator();
  }
}
```

Employee.java
```java
package name.panitz.boilerplate;
import java.util.*;
class Employee implements Iterable<Object>{
  Person person;Salary salary;
  Employee(Person person,Salary salary){this.person=person;this.salary=salary;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();
    result.add(person);result.add(salary);
    return result.iterator();
  }
}
```

Person.java
```java
package name.panitz.boilerplate;
import java.util.*;
class Person implements Iterable<Object>{
  String name;String address;
  Person(String name,String address){this.name=name;this.address=address;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();
    result.add(name);result.add(address);
    return result.iterator();
  }
}
```

Salary.java
```java
package name.panitz.boilerplate;
import java.util.*;
class Salary implements Iterable<Object>{
  Double amount;
  Salary(Double amount){this.amount=amount;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();result.add(amount);
    return result.iterator();
  }
}
```

References

[CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[LP04] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[PJ98] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, pages 9–15. IEEE Computer Society, 1998.

[PJ03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[T. 04] T. Bray, and al. XML 1.1. W3C Recommendation, February 2004. `http://www.w3.org/TR/xml11`.

[Vis01] Joost Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 270–282. ACM Press, 2001.

[Wor99] World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.