Teaching object-oriented programming on top of functional programming

Kristensen, Jens Thyge; Hansen, Michael Reichhardt; Richel, Hans

Published in:
Proceedings on 31st Annual Frontiers in Education Conference

Link to article, DOI:
10.1109/FIE.2001.963848

Publication date:
2001

Document Version
Publisher's PDF, also known as Version of record

Teaching Object-Oriented Programming on Top of Functional Programming

Jens Thyge Kristensen, Michael R. Hansen and Hans Rischel[1]

Abstract - *In the Informatics Programme at the Technical University of Denmark, we base the first course in object-oriented programming (using the Java language) on a preceding course in functional programming (using the SML language). The students may hence exploit concepts from functional programming in the construction of OO programs. This is done following a method where the program design is expressed in SML and afterwards implemented in Java. The use of different languages in design and implementation is an advantage as it makes the distinction between these two stages very clear. We give examples showing that SML designs allow us to develop and compare OO implementations with different class structures for the same programming problem. A discussion of this kind is not supported in traditional OO methodology. The program design in SML has also shown to be useful for the students when documenting the program.*

Keywords - Functional Design, Object-Oriented Programs, Program Design, Implementation

Introduction

Teaching Java in an introductory level is a great challenge if you expect the students to do more than just play with GUI's and other parts of the technology. Choosing a good program structure (classes, objects and methods) for a given problem is quite difficult and there is not much help to get in a standard Java textbook. In a problem to be solved by the students you will hence have to add hints on the program structure – unless the problem is so simple that the structure to be used is obvious. As a result, the students will learn something about Java, but they do not learn how to write a program. This paper discusses a different approach, which has been used for some years now, where we teach useful guidelines on how to structure a Java program by solving a given programming problem.

In the Informatics programme at the Technical University of Denmark, the first programming course is in functional programming using the SML programming language [3]. In this course, emphasis is put on structured data such as records, lists, disjoint unions, trees, sets and tables and their use in different kinds of applications. Furthermore, the rich type system of SML is used to express a program design in a succinct manner, where type expressions are used for data modelling and function types are used to specify the parameters and the results of the main tasks of the program.

After learning functional programming (in SML) the students are introduced to Object-Oriented (OO) programming (in Java). A major challenge of this course plan is to teach OO programming in a way that allows the students to take advantage of skills obtained in the preceding functional programming course. This is done by using program designs expressed in SML as the basis for Java implementations.

Our experience is that this gives a significant improvement in the quality of the produced Java code as well as the program documentation. The reason is, we believe, that making an SML design forces the students to clarify important concepts in the problem instead of rushing into implementation. This clarification of important concepts is of great value for implementation as well as for documentation.

An "SML design" in the sense of our course is an SML *signature*. From a methodology point of view, it covers the following decisions about the program:

- *Concepts and terminology:* names for important data and functions.
- *Data:* the structure of important data, expressed in terms of SML types of the data.
- *Functions:* the parameters and the results for important functions, expressed in terms of SML types of the functions.

The SML design helps the student in designing the overall *structure* of the program. It is our experience that this is the difficult part in developing a program – to complete the code is much easier. In doing so one must, however, consult the original problem statement to find out what each function "should do" as this information is not included into the SML design.

The SML design can be extended to cover also the dialogue with the user. This extension of the design describes the structure of the input and output data and the sequencing of inputs and outputs. We will, however, not pursue this part of the design in the present paper.

The transition from an SML design to the class structure of a Java program is comprised of the following decisions:

- identifying and naming the classes of the program,
- allocating each data item of the SML design to a specific class, and
- allocating each function of the SML design to a specific class.

These decisions fix the class structure of the program, so the code can now be filled in. In addition, the approach provides the basis for comparing and assessing possible OO structures corresponding to the SML design.

In the next section we give an example-based introduction to SML designs and possible ways of deriving Java programs. Section III contains an example of symbolic differentiation

1 Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, {jtk,mrh,hsr}@imm.dtu.dk

involving expression trees. Section IV discusses the experience we have had with student's projects using SML designs as a basis for implementing as well as documenting the Java program. Finally, section V discusses a tool supporting the transition from SML designs to Java (skeleton) programs.

Example

In this section we present and discuss two different SML designs and their Java implementations. The presentation will be based on the following example:

A bank keeps track of the customer's bank accounts by a register which associates the balances of accounts with the names of customers. Furthermore, there are operations to create an empty register containing no accounts, insert a customer with a given balance into a register, to deduct an amount of money from a customer's account in a register, and to delete a customer from a register.

Version 1: Focus on the structure of data

In the following design of bank accounts with operations, we will focus on the structure of data. Important entities in the problem formulation, i.e., *register*, *name*, *balance* and *amount* are named and typed. Furthermore, types are provided for values and for functions, i.e., for *empty register* and for *insert*, *deduct* and *delete*:

```
type register
type name = string
type balance = int
type amount = int

empty :  register
insert:  register * name * balance -> register
deduct:  register * name * amount -> register
delete:  register * name -> register
```

This is an example of an SML *signature* (in a slightly relaxed syntax) that expresses the SML *design* of the program.

Note that the use of SML signatures supports a succinct documentation of the program design. It gives a clear overview of important concepts, and we just need to add a short, textual explanation to the SML types of the functions. For example, the explanation for the deduction operation is: "The value of the expression deduct(reg, n, a) is the register obtained from *reg* by deducting a dollars from person n's account".

We will assume that the old register needs not be retained when a new register has been computed. This is actually a property of the user interface which is not considered here. It is important to note that there are many Java implementations of this design.

An implementation focussing on the register
One observation of the design is that the functions take a register as part of their argument, and they give a new register as their result. Because we have assumed that the old register is not needed any more, it is natural to implement the design with a Java class `Register` containing a private data field to hold the current value of the register and methods for each of the specified functions.

This leads to the following Java skeleton:

```
class Register
{ private ... register;

  Register(){...}

  void empty(){...}
  void insert(String name, int balance){...}
  void deduct(String name, int amount){...}
  void delete(String name){...}
}
```

Note:

1. It is not decided how the register is represented, just that it is a private, mutable object, accessible by the methods of the class.
2. The program skeleton of each method is systematically derived from the corresponding function type.
3. The constructor `Register` and the method `empty` have the same functionality. Thus, the method `empty` is not needed in the final implementation.

The above sketch of a Java implementation has a drawback. It is not easy to extend. For example, if a new type of account is introduced for special customers, then one has to modify the above code and recompile the whole system. Similarly, a recompilation is needed if the representation of the register is changed for some reason. As a result, Java programs are usually written in a different way, as shown in Section B below.

An implementation focussing on the customer
In analysing the function types, one observes that the methods `insert`, `deduct` and `delete` could be hosted either in a register class or in a customer class. Customers are given by their names in the function types.

The methods in the `Customer` class then have a register as a parameter and `name` as a private data field. Details are left to the reader. Hence, this simple example shows that many implementation decisions have to be made in designing an objected-oriented implementation, e.g., which data should be arguments to methods and which should be private data fields of the objects.

Version 2: Focus on a Register Interface

In the design below, we declare an SML datatype `command` that describes the different operations and their parameters. We then only need a single function `eval` to execute any operation. The types for `register`, `name`, `balance` and `amount` are as before:

```
type register
type name = string
type balance = int
type amount = int

datatype command = Empty
                 | Insert of name*balance
                 | Deduct of name*amount
                 | Delete of name

datatype result = Ok of string
                | Error of string

eval :  register * command -> register * result
```

The different operations on the register are enumerated in the declaration of command, e.g., an insert operation is given by a name and a balance. Similarly, in the declaration of `result`, there is an enumeration of the different kinds of results of evaluating commands. The datatypes `command` and `result` are examples of disjoint unions. The register interface is comprised of commands, results and the evaluation function.

When we aim to implement a Java program that is easy to extend, e.g., with new kinds of accounts, there are two possibilities:

1. Operations on accounts are implemented by *inheritance* from a given class, e.g., a register class. New operations are then implemented and compiled separately, and this does not affect the existing implementation of the old operations on accounts.
2. Operations on accounts are described by a Java *interface*. Each command is obtained as an implementation of that interface. Hence, each command gets its own class and recompilation of old commands is not necessary when implementing new commands.

An Implementation using Inheritance

Looking at the above SML design, with an eye to achieving an easily extensible Java implementation, we observe that:

1. The introduction of a new command, for a new kind of account, must be accompanied with an evaluation function for that new command.
2. When updating a register, we assume as before, that the old register is not needed any more. As a result, we can make a class `Register` as before, with a private data field containing the current register.
3. The Java method `eval` can be hosted in the `Register` class, as the evaluation function takes the current register as part of its argument and gives a new register as part of its value.

A corresponding Java skeleton is:

```
class Register
{ private static ... register;

  Register(){...}

  Result eval(){}
}
```

The implementation of the various commands is now obtained by extension of the `Register` class. In each case, one must give a constructor and an evaluation function for the command.

```
class Empty extends Register
{ Empty(){...}
  Result eval(){...}
}
```

Note that `register` must be declared `static` in the `Register` class, as one would otherwise create a new register each time a command object is generated, e.g., by executing `new Empty()`.

The class for `Insert` is declared by:

```
class Insert extends Register
{ private String name;
  private int balance;

  Insert(String name, int balance)
  {this.name = name; this.balance = balance;}

  Result eval(){...}
}
```

The above declaration of the Java constructor `Insert` corresponds to the part: `Insert of name*balance` of the SML datatype declaration of command, which expresses that an insert command is given by: `Insert`(n, b), where n is a string (the name) and b is an integer (the balance).

The remaining commands are implemented similarly:

```
class Deduct extends Register
{ private String name;
  private int amount;

  Deduct(String name, int amount)
  {this.name = name; this.amount = amount;}

  Result eval(){...}
}

class Delete extends Register
{ private String name;

  Delete(String name){this.name = name;}

  Result eval(){...}
}
```

It remains to give a Java implementation of the `result` datatype in the SML design expressing the possible results as the disjoint union of `Ok`-messages and `Error`-messages. We shall implement such disjoint unions by implementing an abstract class in the following way:

```
interface Result
{ String toString(); }

class Ok implements Result
{ private String s;

  Ok(String s){this.s = s; }

  public String toString(){...}
}

class Error implements Result
{ private String s;

  Error(String s){this.s = s; }

  public String toString(){...}
}
```

We have added skeletons for the `toString` methods in the implementations.

Observations about the Java implementation:

1. It is easy to extend this implementation with new kinds of commands, and such an extension does not require a recompilation of the existing classes for the commands or the register.
2. The Java skeleton is much bigger than the SML design, so the advantages of using the succinct SML design for documentation purposes are obvious.
3. Inheritance is not really used in the Java implementation. The `Register` class just provides the declaration of a data field for the register and the type of the evaluation method.
4. Starting from the SML design, one could expect that the Java implementation would have a class `Command`. This is, however, not the case in the above skeleton.

An Implementation using Java Interfaces

The classes for results are given in Section B.1. The classes for commands are implemented in a similar way. The evaluation function, however, takes a command as part of its argument and, therefore, it is included in the command interface:

```
interface Command
{ Result eval(Register register);

  String toString();
}
```

The type of the method `eval` is interpreted as follows: It takes the host command and a reference to a register as its arguments, and it returns a result. As a side-effect, it may change the register. Note how this fits the corresponding SML design.

Classes for the different commands are obtained as implementations of the command interface:

```
class Empty implements Command
{ Empty(){...}

  public Result eval(Register register){...}

  public String toString(){...}
}

class Insert implements Command
{ private String name;
  private int balance;

  Insert(String name, int balance)
  {this.name = name; this.balance = balance;}

  public Result eval(Register register){...}

  public String toString(){...}
}

class Deduct implements Command
{ private String name;
  private int amount;

  Deduct(String name, int amount)
  {this.name = name; this.amount = amount;}

  public Result eval(Register register){...}

  public String toString(){...}
}

class Delete implements Command
{ private String name;

  Delete(String name){this.name = name;}

  public Result eval(Register register){...}

  public String toString(){...}
}
```

The SML design expresses the existence of a register type, but nothing about its structure. Hence, the concrete form is not visible at the register interface, and we arrive at the following trivial Java skeleton:

```
class Register{...}
```

The register class must, of course, contain methods which can be used in the implementations of the various `eval` methods above.

This Java implementation is extensible using the approaches as in the previous implementation. Furthermore, it has the nice property that the two datatype declarations (command and result) from the SML design are represented the same way in the Java skeleton. Furthermore, it is much more natural that the evaluation function is associated with command classes, rather than with the `Register` class as it was the case in the previous implementation.

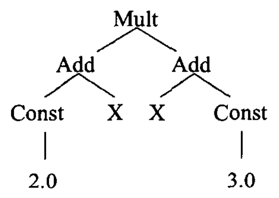
Symbolic Differentiation

This section shows how to obtain a Java program from an SML design containing a recursive datatype, using the derivation methods described in Section B.2. The example is that of symbolic differentiation, where expression trees are defined recursively, and where we use D to denote the differentiation function:

```
datatype Fexpr = Const of real
               | X
               | Add of Fexpr • Fexpr
               | Mult of Fexpr * Fexpr

D : Fexpr -> Fexpr
```

SML-values of the type `Fexpr` are expression trees like:



A tree representing $f(x) = (2 + x) \cdot (x + 3)$.

The abstract class for `Fexpr` is declared by:

```
interface Fexpr
{
  Fexpr D();

  String toString();
}
```

where the interface contains the method D, as `Fexpr` is the argument type of the function D.

The Java skeletons for the various expression trees are:

```
class Const implements Fexpr
{ private double c;

  Const( double c ){ this.c = c; }

  public Fexpr D(){...}

  public String toString(){...}
}

class X implements Fexpr
{ X(){}

  public Fexpr D(){...}

  public String toString(){...}
}

class Add implements Fexpr
{ private Fexpr fe1, fe2;

  Add(Fexpr fe1, Fexpr fe2)
     { this.fe1 = fe1; this.fe2 = fe2;}

  public Fexpr D(){...}

  public String toString(){...}
}

class Mult implements Fexpr
{ private Fexpr fe1, fe2;

  Mult(Fexpr fe1, Fexpr fe2)
      { this.fe1 = fe1; this.fe2 = fe2;}

  public Fexpr D(){...}

  public String toString(){...}
}
```

Having this skeleton, it is easy to fill out the remaining parts to get a complete program. We only give the remaining code for the `Mult` class:

```
public Fexpr D()
{
  return new Add( new Mult( fe1.D(), fe2 ),
                  new Mult( fe1, fe2.D() ) );
}

public String toString()
{
  return "(" + fe1 + " * " + fe2 + ")";
}
```

Note that only a few lines of code need to be added and that the declaration of the method D corresponds to the product rule for differentiation.

Symbolic differentiation is a complicated programming problem for second semester students in a Java course. Our experience is that many students are not able to solve it at all, while others come up with Java programs with a bad structure, i.e., programs that are hard to understand, even for the programmer.

Our approach gives an awareness on how to choose a class structure to get a proper Java implementation. This example shows that a method relating SML designs to Java skeletons helps students solve problems that they were not be able to solve properly otherwise.

Experience

The students in our second semester Java course are evaluated based on reports documenting the solution to a software engineering problem. Previous experience with these reports has shown that the students had severe difficulties making a good object-oriented program as well as a good documentation of the program. Some years ago, the form of these projects has changed to require that students document the design using an SML design as described above. This has led to a significant improvement of both the implementation and the documentation.

The reason for these improvements are, we believe, that the SML design supports an abstract view of the system, where decisions about implementation details are postponed. In the example section, we saw how much freedom the SML design leaves in making an object-oriented representation of the account system. It also provides a useful frame of reference when comparing different object-oriented implementations. Furthermore, the abstract view of the SML design is a much better starting point for the documentation of the program, as it is not cluttered by implementation details. This positive experience with using SML designs, even when an object-oriented implementation was the goal, has led to focus on teaching systematic derivations of Java skeleton programs from SML designs.

Automating the Derivations

A prototype of a tool that can automatically generate Java skeleton programs from SML designs has now been constructed. Decisions about the OO structure appear as annotations to the SML design. A big advantage of the tool is that it generates a skeleton program which fits the chosen design, so that the programmer can begin filling in the implementation details of the specified entities. We saw in the section on symbolic differentiation, that only a few well-understood lines of code had to be added to the skeleton. Having a tool allows students to experiment with their design and the generated skeletons of the implementations. In the next semester, students will experiment with the tool.

Acknowledgements

We are grateful to the anonymous referees and Ann Quiroz Gates for very useful comments.

References

[1] Deimel, L.E. (Ed.), *Software Engineering Education*, LNCS, vol. 423, Springer-Verlag, 1990.

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley, 1995.

[3] Hansen, M.R., Kristensen, J.T., Rischel, H., A Theory-Based Introductory Programming Course, *Frontiers in Education (FIE'99)*, Volume 1, pages 11b4-25 to 11b4-30, IEEE 1999.

[4] Hansen, M.R., Rischel, H., *Introduction to Programming using SML*, Addison-Wesley, 1999.

[5] Sestoft, P., Kristensen, J.T., Ravn, A.P., Rischel, H., From Functional to Imperative Programming, *Les languages applicatifs dans l'enseignement de l'informatique*, pp. 26-33, Actes des 2èmes journées de travail, IFSIC-IRISA, Rennes 1993.

[6] Tucker, A.B. (Ed.), Computing Curricula 1991, *Communications of the ACM*, 34, June, pp. 69-84, 1991.