Transactions as a Cross-cutting Concern

Egon Wuchner
Siemens AG
Otto-Hahn-Ring 6
81739 Munich
Germany
egon.wuchner@siemens.com

Introduction

This focus group submissions takes a look at transactions as a cross-cutting concern. It takes the approach of starting from a possible solution of AspectJ and trying to find a corresponding pattern solution. This pattern solution tries to address the forces mentioned by the focus group submissions: As a result it shows some encountered difficulties in finding a pattern solution which copes with all forces mentioned above.

However, having a closer look at component/container models reveals some new facets. Component/container models (like application servers and EJBs) encapsulate cross-cutting concerns like transactions, security and persistence. Nevertheless, as the section … shows component/container models sufficiently focus on a subset of the forces above, mainly transparency and ex-changeability of policies. On the other hand modularization (and reusability of cross-cutting concerns) has less significance within their context, since deploying a component allows to generate auxiliary classes specific to the original component. Thus the cross-cutting concern does not have to be intrinsically modularized (and re-usable).

Transactions and Patterns

Let us consider a simple account class allowing to withdraw and deposit money for a customer.

```
class Account {
private: double balance; // package private
 public:
  void debit( double withdraw )
    throws InsufficientBalanceEx
  {
    double newBal = this.balance-withdraw;
    if(newBal<0.0)
      throw InsufficientBalanceEx;
    else
      this.balance = newBal;
  };

  void credit( double supply ) {…};
  …
 }
```

We would like to execute each account operation (debit, credit) within a transaction. Starting with aspects the AspectJ solution (see [Laddad]) might look as follows:

```
aspect Transaction {
  abstract pointcut transactedOp();

  Object around() : transactedOp() {
      try {
        proceed(); // continues transactedOp
        this.commit();
      catch(…) {
        this.rollback();
   };

private:
  void commit(){ dbConn.commit();}
  void rollback(){ dbConn.rollback();}

  Connection dbConn = … }
```

```
aspect AccountTransaction extends Transaction
{
  pointcut transactedOp():
    execution(void Account.debit(..)) ||
    execution(void Account.credit(..))
}
```

The solution provides an abstract aspect defining an advice containing all the transaction code. This advice is connected to the execution of an abstract pointcut. The sub-aspect binds this pointcut to concrete method executions like debit and credit. The basic idea behind is comprehensible without being very familiar with AspectJ: the original method execution (called within an aspect by the "proceed" statement) is controlled by the surrounding transactional code of the advice.

Now, let us find a solution by using patterns. We would like to have the functionality of transaction management bundled in one place to be able to easily change its implementation. We use the Method Object (see Command of [GoF95]) and the Template Method pattern to modularize the transactional code in a reusable way. The transaction logic handles exceptions of the original code as an indication of failure of the original operation and initiates a rollback. The transactional steps surrounding the original method are fixed and indicate a possible usage of the Template Method pattern [GoF95]. The Method Object pattern is needed in order to apply the transactional logic to each transacted method (debit and credit, respectively). Consequently, we need at least a combination of several patterns.

```
/* Transaction */
abstract class Transaction {
  abstract void transactedOp();

  void execute()
  {
    try {
      transactedOp(); // continues transactedOp
      conn.commit();
    catch(…) {
      conn.rollback();
  }

private: Connection conn = …
}


/* Account */
class Account {
 private: double balance;

  /* inner class AccountDebitTransaction */
  class AccountDebitTransaction
  extends Transaction
  {
    AccountDebitTransaction( double  money )
    { this.money = money; }

    void transactedOp()
    { double newBal = this.acc.balance-withdraw;
      if(newBal<0.0)
        throw InsufficientBalanceEx;
      else
        this.acc.balance = newBal;
    }

   private: double money;
  }

  /* inner class AccountCreditTransaction */
  class AccountCreditTransaction
  extends Transaction {…}

public:
  void debit( double withdraw )
  { Transaction tr =
      new AccountDebitTransaction( withdraw );
    tr.execute();
  }

  void credit(…) … }
```

In contrast the AspectJ solution comes up with another advantage. For instance, changing the transactional policy from auto-commit of each Account operation to top-level transactions can be done easily by using the AspectJ solution. As an example a Transfer class of the accounting system does start a high-level transaction in order to guarantee the proper operation of transferring money. Therefore, `Account.debit` and `Account.credit` have to run in the same transactional context when transferring money from one account to another.

```
class TransferSystem {
public:

  void transfer( Account from, Account to,
                 double sum)
    throws InsufficientBalanceEx
  { from.debit(); to.credit(); }
}
```

By using aspects (see [Laddad]) it is possible to identify each potential top-level transaction start in a straightforward way (briefly mentioned without getting into the details by using a cflowbelow pointcut declaration). Thus, the AspectJ solution supports the exchangeability of policies specifying how to apply a cross-cutting concern. With respect to a large software system this proves to be of major benefit. Consequently, the question comes which pattern combination might achieve the same goal. Furthermore, a pattern solution should also aim to introduce transactions and the application of different policies in a highly transparent and non-invasive way (as done by AspectJ)

Transactions and Component models

The above pattern solution fosters independent ex-changeability of the transaction concern, but it is not at all transparent to the developer of the Account class. However, relying on the fact that the focus of these forces can change depending on the context, even a pattern solution covering partial forces can be useful. For example component models of enterprise systems (like application servers hosting EJBs) emphasize the exchangeability of transaction policies and transparency of cross-cutting concerns to components. Therefore another solution comes up leveraging a combination of the Decorator and Factory (or Lookup) patterns [GoF95].

```
/* interface AccountIF */
interface AccountIF {
  void debit( double withdraw )
    throws InsufficientBalanceEx;

  void credit( double supply );
}

/* class Account */
class Account implements AccountIF{
public:
  void debit( double withdraw )
    throws InsufficientBalanceEx
  {
    double newBal = this.balance-withdraw;
    if(newBal<0.0)
      throw InsufficientBalanceEx;
    else
      this.balance = newBal;
  };

  void credit( double supply ) {…};
  … }
```

```
/* class AccountDecorator  */
class AccountDecorator implements AccountIF {
public:
  void debit( double withdraw )
    throws InsufficientBalanceEx
  {
    try {
      this.account.debit( withdraw);
      conn.commit();
    catch(…) {
      conn.rollback();
  }

  void credit( double deposit ) {…};

private:
  AccountIF  account = …
  Connection conn = …
  …
}
```

```
/* factory class or lookup class*/
class Factory {
public:
  static AccountIF create(…)
  {
    // creation dependent on the
    // transaction policy, e.g. specified
    // in a descriptor
    //
    if( /* auto-commit */)
      return new Account(…);
    else /* top-level transaction */
      return new AccountDecorator(…);
  }

  static AccountIF lookup(…) {…} }
```

In order to make the transaction policy exchangeable we have to facilitate the decorator pattern for the TransferSystem class as well (it also implies making the static methods non-static). As for transparency, the solution keeps the original component code (like Account and TransferSystem) clean of any transactional code. As a side effect exchanging transaction policies is modularized within the factory.

But note that this pattern solution is in no way reusable across classes different to Account and TransferSystem. Furthermore it also takes some extra effort to modularize the transaction code. Nevertheless it suits a component/container model. The container is a kind of tool allowing to generate the corresponding decorator and factory classes at deployment time of the component.

REFERENCES

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns-Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[Laddad] R. Laddad: AspectJ in Action, Manning, 2003