Designing Applications Using Data-Context-Interactions Architecture In Morpheus

Abstract: *Data-Context-Interactions architecture (DCI) is a software paradigm whose main goal is to bring the end user's mental models and computer program models closer together. It has been shown, however, that developing applications according to the DCI principles is rather difficult in the current languages. This paper presents a novel way of developing applications according to DCI in the framework of Morpheus, which is a proof-of-concept implementation of Object Morphology (OM) in Scala. OM is a new object-oriented paradigm developed to model highly mutable phenomena that may mutate not only with regard to state, but also with regard to type. Instead of classes, OM uses the concept of morph models describing possible forms of objects. Using a simple example it is demonstrated how natural and straightforward it could be to implement applications following the DCI architectural tenets.*

Key words: Data-Context-Interactions, software architecture, object-oriented programming, object morphology, Scala, mutable objects, Morpheus.

1. Introduction

Data, context, interactions (DCI) is a software paradigm introduced by Reenskaug (2008) and further elaborated by Coplien, et al. (2010). Its main goal is to bring the end user's mental models and computer program models closer together. In other words, the user must feel that he or she directly manipulates the objects in computer memory that correspond to the images in his or her head. In this respect, DCI can be seen as a further development of the Model-View-Controller design pattern, whose goal is to provide the illusion of a direct connection from the end user’s brain to the computer’s “brain”, i.e. its memory and processor (Reenskaug and Coplien, 2009).

According to this paradigm, data, context and interactions are the three fundamental facets of the end user's interpretation of computer data. Each of these facets is considered a component of the application under development.

The data are represented by domain objects, which “know” everything about their state and how to maintain it, but “know” nothing about other objects in the system (Hasso, et al., 2014). Coplien, et al. (2010) use the term *dumb objects* to refer to those domain objects. The technique for the identification of domain objects responsibilities is often called “Do it Myself” and is described in (Wirfs-Brock, et al., 2003) and (Coad, et al., 1997).

Next to the domain objects there is another kind of objects called interaction objects. These objects capture the interactions between domain objects as system functionality. Importantly, the interaction objects are treated by DCI as first class citizens. In contrast to domain objects, the concept of interaction objects is a novelty introduced by DCI.

Only after the data are put into a context, in which they are subject to interactions amongst themselves, can the data yield some information (Reenskaug, 2008). The context can be seen as the locus of the use case enactment in the architecture encapsulating the interactions (or the roles) that define the system dynamics (Coplien and Reenskaug, 2012).

In summary, in DCI, every use case is represented by means of context. The context defines roles performing interactions among themselves. Each role in the context is played by one corresponding domain object (data, entity). The role contains the code that would otherwise reside in the object's class. Thus, roles effectively separate the stable part of the code from the unstable.

The context itself only defines roles and triggers the use-case. The context and roles should reside in a single dedicated file so that one can easily investigate the interactions.

DCI is anchored in object-oriented programming (OOP), however, it must cope with some inherent flaws in OOP. It is generally accepted that OOP is very good at capturing the system's state by means of classes and their properties. OOP is also good at expressing operations with the state captured by a class, unless these operations involve some kind of collaboration with instances of other classes.

Nevertheless, implementing applications within the DCI paradigm is quite difficult in the traditional OOP, by large on account of the fact that OOP fails to properly express collaborations between objects (i.e. use cases), as shown in (Savkin, 2012). Let us consider an object that appears in several use cases and behaves very differently in each case. Because of the conceptual insufficiency of OOP, a developer is forced to express such a use-case-specific behavior of the object as an operation in the object's class. Unfortunately, this has several undesirable consequences:

First, there is no single file or other artifact dedicated solely to one use case where a developer could see all interactions among objects. This makes orientation in and maintenance of the code quite difficult.

Second, the whole behavior of the use case is scattered across the classes of the collaborating objects. This forces the developer to add a number of unrelated methods to classes with every new use case (causing higher coupling and lower cohesion of classes).

Third, it is virtually impossible to separate the stable part of the code, i.e. that which captures the data, from the variable part, i.e. the use cases (behavioral part).

As mentioned in the "Related works" section, there are a number of DCI implementations in various programming languages, which more or less completely implement the DCI elements. In contrast to those implementations, which are built on top of the traditional object-oriented paradigm, the DCI demonstration shown in this paper is developed using Morpheus (Šlajchrt, 2015a), which is a proof-of-concept Scala implementation of the *Object Morphology* paradigm (OM), a novel object-oriented approach to modeling mutable phenomena (Šlajchrt, 2015b). The central concept of OM is that of the *morph model*, which describes all possible forms of an object. In OM, morph models are used as a conceptual framework replacing the notion of class.

The main goal of this tract is to demonstrate that DCI naturally arises from the framework of Object Morphology as a design pattern or an architectonic style. This is what distinguishes the "DCI over Morpheus" approach from the other DCI implementations, which must often go far beyond the traditional OOP and stretch the underlying languages to their limits. Provided that Morpheus is an implementation of OM, it should be easy and straightforward to develop an application following the DCI principles on top of Morpheus, as illustrated in this paper on the paradigmatic DCI example – the money transfer simulation.

2. Related Works

There are a number of concepts, which share some conceptual components with DCI. The following paragraphs present the most influential and interesting ones.

Mixins or *dynamic traits* provide a means for encapsulating a specific object behavior, such as interactions with other objects, into a special language construct reminding a class with no state (Groovy, 2016). An object may be turned into a DCI role by injecting a role mixin into it at runtime. However, it may be quite difficult to consistently combine multiple mixins at the level of a use case (Wikipedia, 2016a).

Dependency injection allows segregating some object's behavior into an external object. This segregation is accompanied by the specification of the interface between the original and external object. The behavior of the original object may be adapted to a given context (use case) by injecting an external object (role) at runtime. However, this approach suffers from the problem of *self-schizophrenia*, which is a condition in which the execution of some logic is carried out in the context of two or more objects (more selves), instead of only one. Such a condition may lead to subtle problems in the design (Sekharaiah, 2002). The problem of self-schizophrenia is treated well by DCI.

Aspect oriented programming (AOP) aims at increasing modularity of programs by the separation of crosscutting concerns, which is the goal shared with DCI. AOP provides a developer language tools for specifying the so-called *pointcuts* referring to places in the existing code, where additional behavior, such as logging or profiling, will be added. Nevertheless, Steinmann (2006) polemicizes with the claim that AOP improves the code quality. Instead, he argues that AOP has a negative impact on both modularity and readability of the code.

Role-oriented programming (ROP) is in many respects similar to DCI, mainly with regards to the dependency of roles on context and the emphasis on the interaction among a group of roles within the context. There are, however, a couple of differences between these two paradigms. While ROP encourages using inheritance as an important concept in designs, DCI avoids using inheritance completely when modeling roles. Furthermore, according to (Coplien and Reenskaug, 2012), object-schizophrenia remains a problem in ROP.

Subject-oriented programming (SOP) aims at facilitating the development and evolution of suites of cooperating applications. In SOP, these applications can be developed separately and composed afterwards. A new application can be introduced to an existing composition of applications without requiring modification of the other applications. The term subject refers to a collection of state and behavior of an object as seen by a particular application (Harrisson and Ossher, 1993). In other words, subjects capture subjective perceptions of objects. A subject may be seen as a close relative to a role in DCI. Also, there is close relationship to AOP. In contrast to AOP, SOP restricts placing join-points to field access and method call (Wikipedia, 2015).

There are also a number of DCI implementations developed on top of various languages differing in how much they conform to the DCI principles. The full list is available online (FullOO, 2016).

3. Research Method

The use of Morpheus to develop an application following the DCI rules is illustrated on the paradigmatic DCI example – the money transfer. This section begins with a conceptual analysis of the example followed by the implementation in Morpheus.

3.1. Money Transfer Example

The use case scenario is this: the end user uses the bank terminal to transfer money from one account to another. He or she selects the source and the destination accounts from the list of available accounts (Fig. 1). Then he or she specifies the amount of money to be transferred and starts the transaction. Some exceptions can be raised, of course, for instance as long as there is not enough balance in the source account to perform the transfer.

For the sake of simplicity, let us assume that the data model of the bank application is just the list of the end user's accounts. Every account is represented by an object encapsulating some basic properties like the balance along with some basic operations like `increaseBalance` and `decreaseBalance`. We can expect that such a data model will be pretty aligned with the end user's mental model.

Fig. 1: Symbols of the user's bank accounts

When transferring money the user will intuitively be familiar with the basic steps of the procedure. The user will know that it is a simple interaction between two accounts, one playing the role of the source account and the other playing the destination account role and that the balance of the source account will be decreased by the amount of the transferred money while the destination's balance will be increased by the same amount. The schema of this mental model is depicted in Fig. 2. The source and destination roles are depicted as a square and rectangle. Both shapes have the same "hole" representing the slots that can be filled by any account circle symbol from Fig. 1.

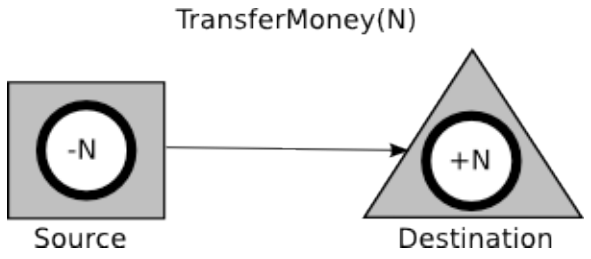

Fig. 2: The schema of the money transfer with the roles and the slots for the accounts taking part in the use case

It is important to note that the roles, i.e. the "rectangle" and "triangle" roles have no identity. They can be regarded as costumes of the account objects. It is the object playing the role that carries the identity.

In many object oriented programming languages it is usually very easy to express both the data and the abstract context (i.e. using roles that are not bound to the objects yet). What is not that easy, however, is the binding of objects to their respective roles in the context. In this use case the goal is to fill the holes in the rectangle and the triangle by the chosen account objects (Fig. 3).

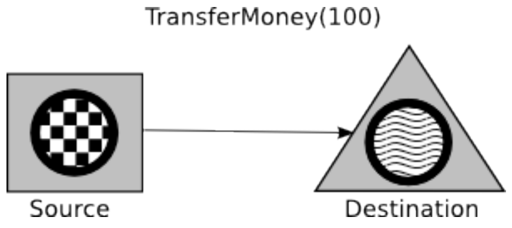

Fig. 3: After the accounts are assigned to the roles, the money transfer can be executed

This is the moment at which the things are becoming complex. Without modern programming concepts like mixins, traits, aspects or meta-programming it is may be very difficult to overcome this point. Not to mention that any of the above-mentioned concepts has its own issues and does not perfectly fit to DCI. Let us suppose that this difficult step may be overcome and the objects are assigned to their roles making the context ready to transfer the money.

3.2. Morpheus Overview

Object Morphology (OM), of which Morpheus is a proof-of-concept implementation, is a general approach to modeling primarily the so-called *protean objects*. A *protean object* is a term referring to a phenomenon occurring in a multitude of forms and defying the traditional Aristotelian class-based categorization (Madsen, 2006, p.8). The concepts (abstractions) of such objects may often be only loosely defined, e.g. by means of family resemblance rather than by specifying strict rules for class membership.

Examples are fetal development, insect metamorphosis, phase transitions, autopoietic (self-maintaining and self-reproducing) systems such as cells, roles in society, crisis and other biological, social or economic phenomena.

Instead of building type or class hierarchies, protean objects are modeled through the construction of *morph models* describing the forms that the protean objects may assume. The individual forms are called *morph alternatives*.

In its essence, a morph model is an abstraction (or concept) of related protean objects. The individual alternatives in the model are in fact the abstractions of the prototypical or exemplary instances among the abstracted protean objects (i.e. the concept's extension). Each alternative consists of the so-called fragments, which represent properties or features of the protean objects (i.e. the concept's intension).

A key idea of OM is that the state and type of an object are, in general, two intertwined aspects of the object determining each other. For example, the *square-rectangle problem* is an illustration of such a mutual connection (Šlajchrt, 2016a). This problem is used as an illustration of the issues arising when the traditional OO languages are used to model objects, whose type is determined by their mutable state. The crux of the problem lies in the fact that mutable phenomena are modeled using type or class hierarchies. Objects, i.e. instances of the classes in the hierarchies, are prevented from skipping from one class to another in the hierarchy, although it would reflect the reality.

In OM, the problem virtually disappears, mostly on account of the fact that protean phenomena are modeled using morph models, instead of type hierarchies. (Note: hierarchies are actually a special kind of morph models (Šlajchrt, 2015b, pp.140-170).) The model describing the square/rectangle objects contains two alternatives: one representing pure (non-square) rectangles and another for squares (Fig. 4). This model expresses the type/state correspondence: a pure rectangle (Alt1), whose width and height are set to the same values, becomes a square, or more accurately, a rectangle with the square feature (Alt2). The square feature is preserved until the two dimensions violate the square constraint.

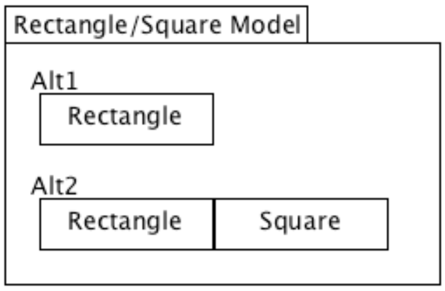

Fig. 4: The schema of the rectangle/square morph model

On a statically typed OO platform, the compiler may build the morph model by parsing the morph model's type expression at compile-time. The compiler may analyze the morph type expression, build the model instance and perform various checks to guarantee that all alternatives in the model are complete and consistent.

A morph alternative describes one of the forms of a protean object and consists of one or more *fragments*. A fragment is a building block representing a typological, behavioral and structural element of protean objects. It represents a property or feature of a protean object and semantically corresponds to the concept of trait as defined in Scala or Groovy. Fig. 5 depicts the `Rectangle` and `Square` fragments with their attributes and methods.

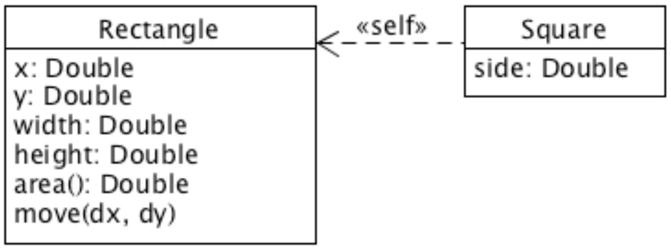

Fig. 5: Rectangle and Square fragments

There is one significant difference in the diagram Fig. 5: the relationship between the `Square` and `Rectangle` is not inheritance as usual, but a special kind of dependency denoted by the `self` stereotype. This relationship requires that any instance of the `Square` fragment be accompanied by an instance of the `Rectangle` fragment in the same object; in other words, if an object contains a `Square` instance, the same object must also contain an instance of `Rectangle`.

Note: The stereotype name is inspired by Scala, which has a special language construct called the *self-type*. This construct is used in traits to declare a type constraint on the class of which the trait can become part.

Instances of alternatives are called *morphs*, which are created by a *recognizer* according to the alternative selected by the recognizer's *morphing strategy*. On every morph instantiation the morph strategy evaluates all possible alternatives and selects the one that matches best the current state of the object or outer conditions such as input parameters. In the case of the square/rectangle model, the morphing strategy would be selecting the second alternative if the width and height are the same, otherwise it would select the first alternative.

3.3. Morph Model Entanglement

In order to elucidate the idea of using OM in a way following the DCI principles it is necessary to explain another important subject in OM, which is *morph model entanglement*. The name suggests that the subject deals with combining morph models. The idea is that there is a morph assembled according to an alternative from its model; this morph is called the *source morph* and its model is called the *source model*. Further, there is another model, whose alternatives contain fragments depending on some fragments from the source model. The goal is to create a morph, called the *target* *morph*, from the *target morph model* in such a way that the selected target alternative matches as much as possible the source alternative used to construct the source morph. The resulting target morph will consist of the new instances of the fragments from the target model and of the existing fragment instances taken from the source morph. The shared source fragment instances will thus be part of two morphs at the same time.

Putting the process of entanglement into the context of DCI, the source morph corresponds to a domain object, which encapsulates the state and contains non-interacting logic manipulating the object's state. On the other hand, the target morph corresponds to a role played by the domain object in a given use case. The following paragraphs illustrate this idea on the money transfer example.

In order to make the connection between OM and DCI more explicit, the demonstration uses a modified version of the example, in which the accounts are protean objects, i.e. objects whose type and state are interconnected and subject to change. In particular, an account may become a *blocked account* when the account balance is negative. Thus the "blocked account" condition manifests not only as a special account state, but also as an additional account type, which may provide some specific additional and/or overriding behavior.

The morph model of the savings account is in Fig. 6. The model contains two alternatives: the first corresponds to the non-blocked mode and the second corresponds to the blocked mode.

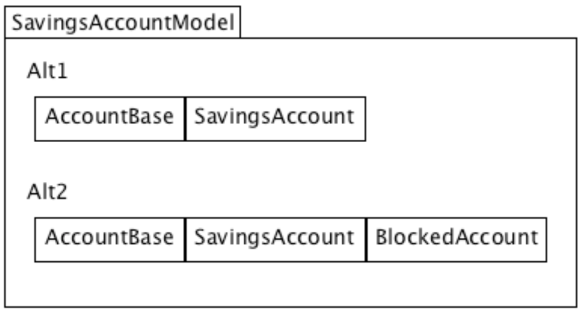

Fig. 6: Savings account morph model

The analogous morph model for the checking account would contain the `CheckingAccount` fragment in place of `SavingAccount`. The relationships between the fragments are depicted in Fig. 7.

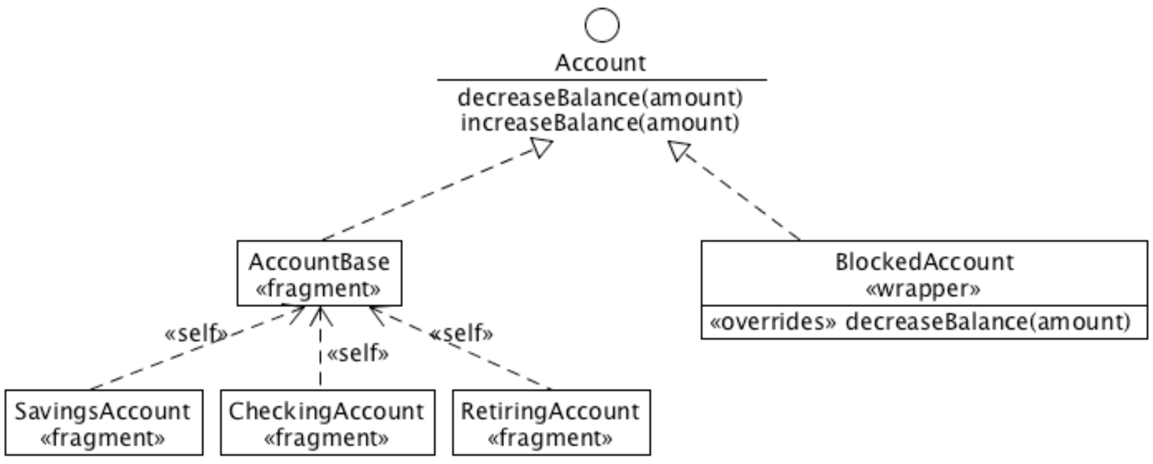

Fig. 7: Relationships between bank account fragments

The Account interface is implemented only by `AccountBase`, which is the common base for all account types, and by `BlockedAccount`. In contrast to `AccountBase`, which implements all methods from `Account`, `BlockedAccount` only overrides the `decreaseBalance` method. Although to override an abstract method from the `Account` interface may sound apparently illogical, it makes a perfect sense here since `BlockedAccount` is a special fragment called *wrapper*, which may exist only in a morph containing another "wrappable" fragment instance, i.e. an instance of the `Account` interface. Note: a wrapping fragment is only allowed to override the inherited methods and it cannot extend in any way the type from which it inherits.

Interestingly, the three particular account types do not extend (or inherit from) the `AccountBase` (just as the `Square` does not inherit from `Rectangle`). Instead, they use the self relationship to declare that the instances of those account types must be accompanied by an instance of `AccountBase` in a morph.

So far, only the domain objects have been dealt with. Now, let us turn our attention to roles and use cases. According to what has been mentioned above, the roles can be modeled as target morphs entangled with source morphs. As far as the money transfer destination role is concerned, the source morph is the destination account. The target morph model has only one alternative consisting of the Destination fragment depending on the Account fragment from the source morph model. The schema of the target morph model is in Fig. 8. The dashed box corresponds to the `Account` abstract fragment, on which the `Destination` fragment depends.

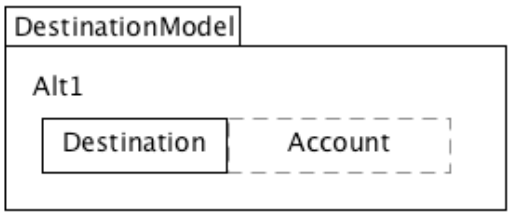

Fig. 8: The `Destination` *role morph model*

The target morph model for the source role is a bit more complicated, as it consists of two alternatives: the first alternative implements the behavior corresponding to the non-blocked source account, while the second one contains an additional fragment – `BlockedSource` – implementing a behavior specific to blocked source accounts (Fig. 9).

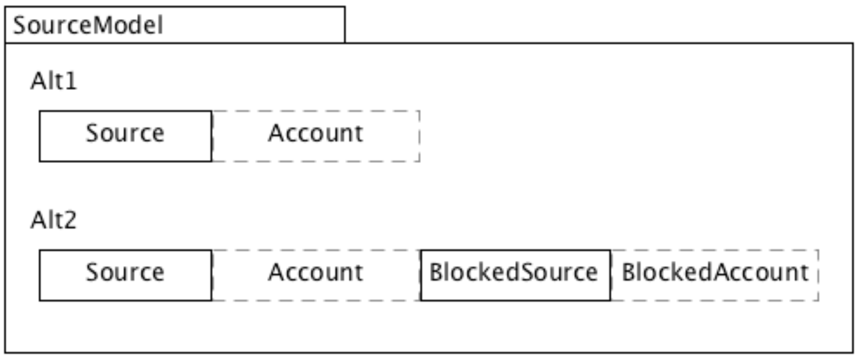

Fig. 9: The `Source` *role morph model*

The relationships between the role (target) and source fragments are depicted in Fig. 10.

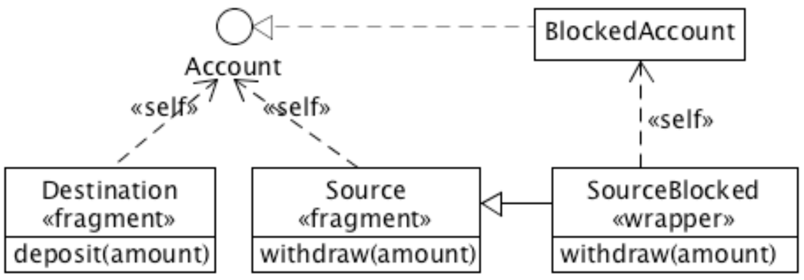

Fig. 10: The relationships between the target and source fragments

Both the Destination and Source role fragments depend on the `Account` interface. The self relationship is used here to express the requirement that a instance of a fragment implementing the `Account` interface be present in the target morph (i.e. in both the target and destination roles). The `SourceBlocked` wrapper fragment implements the specific behavior by overriding the `Source` fragment's `withdraw` method. This behavior takes place when the source account is blocked only.

Fig. 11 shows the destination morph instance after having been entangled with the destination account morph. In contrast to Fig. 8 the destination morph contains the fragment instances of `AccountBase` and `CheckingAccount` originating in the source morph.

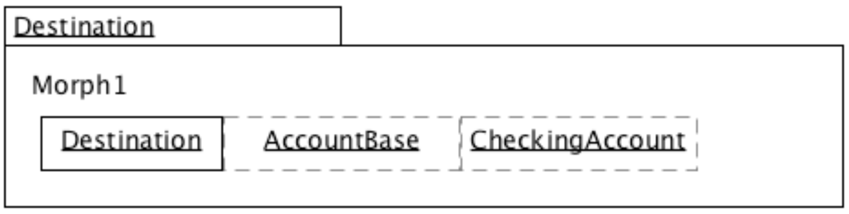

Fig. 11: The relationships between the target and source fragments

Contrarily, the schema of the money transfer source morph consists of two possible morphs (Fig. 12). The first morph is created when the source account is not blocked, while the second one is created when the account is blocked.

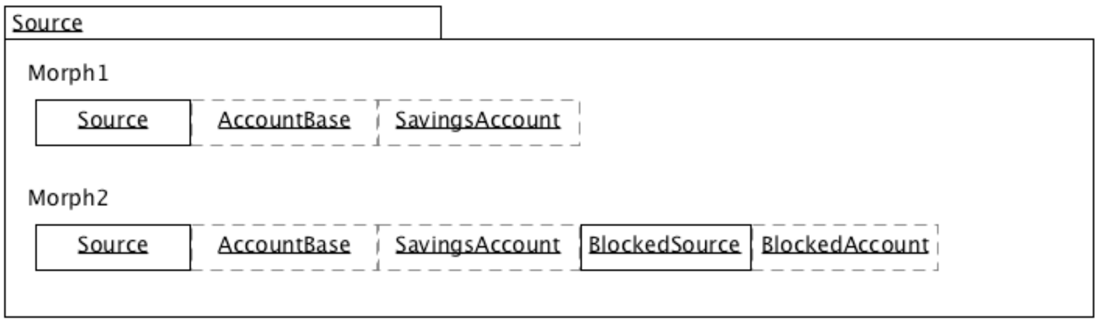

Fig. 12: The relationships between the target and source fragments

The following section deals with a detailed implementation of the above-mentioned ideas in the framework of Morpheus.

3.3. Using Morpheus

Modeling Domain Objects

Using Morpheus to model the domain objects, i.e. the bank accounts, is very straightforward. Listing 1 contains the `Account` interface and its default implementation `AccountBase`, both designed as Scala traits.

```
trait Account {
  def balance: BigDecimal

  def decreaseBalance(amount: BigDecimal): Unit

  def increaseBalance(amount: BigDecimal): Unit
}

@fragment
trait AccountBase extends Account {

  private var bal: BigDecimal = 0

  def balance = bal

  def decreaseBalance(amount: BigDecimal) = bal -= amount

  def increaseBalance(amount: BigDecimal) = bal += amount
}
```

Listing 1: The `Account` interface and its implementation `AccountBase`

The `AccountBase` trait is annotated with the `fragment` annotation, which is used to mark fragments in Morpheus.

Next, there are three different account types, which are also designed as fragments. Accordingly to the design notes in the previous section, the specific account type do not inherit from `AccountBase`; instead, they establish the self relationship with it using the self-type declaration (`this: <self-type> =>`), as shown in Listing 2.

```
@fragment
trait SavingsAccount {
  this: AccountBase =>
}

@fragment
trait CheckingAccount {
  this: AccountBase =>
}

@fragment
trait RetiringAccount {
  this: AccountBase =>
}
```

Listing 2: Three account types implemented as fragments

The `BlockedAccount` wrapper represents, as its name suggests, a blocked account. An account becomes blocked when there is a negative balance on it. In this state, the account owner is not allowed to decrease the balance any more. This behavior is implemented by overriding the `decreaseBalance` method of `Account`, from which `BlockedAccount` inherits (see Listing 3).

```
@wrapper
trait BlockedAccount extends Account {
  override def decreaseBalance(amount: BigDecimal): Unit = {
    sys.error("Blocked")
  }

}
```

Listing 3: The `BlockedAccount` wrapper

Listing 4 demonstrates how a savings account morph can be created.

```
val savingsAcc = {
 val model = parse[AccountBase with /?[BlockedAccount] with
                    SavingsAccount](true)
 val strat = unmaskFull[BlockedAccount](model)(rootStrategy(model),{
    case Some(m) if m.balance > 0 => Some(0)
    case _ => None
  })
  singleton(model, strat).~
}
savingsAcc.increaseBalance(10)
savingsAcc.remorph
```

Listing 4: The initialization of the source account

The account morph is created in the block, whose result is assigned to the `savingsAcc` variable.

The first statement in this block parses the morph model of the savings account. The type argument of the `parse` macro is used to specify the morph model type. The model type contains the `/?[S]` type, which is used to specify an optional part (sub-model) `S` of a morph model. Here, the optional part is `$[BlockedAccount]`. Due to this optional part the resulting morph model has two alternatives: `AccountBase` **`with`** `SavingsAccount` and `AccountBase` **`with`** `BlockedAccount` **`with`** `SavingsAccount`.

The `true` value in the parse macro just instructs the parser to validate the parsed model. It is important to remark that although it looks like a normal method invocation happening at run-time, the parse macro is invoked at compile-time. The compilation would fail if there were any issue with the morph model type.

The second statement in the initialization block creates the morphing strategy, which will be selecting the morph alternative from the morph model according to the morph's state. Briefly, the `unmaskFull` macro creates a strategy, which selects the alternative not containing `BlockedAccount` if the account balance is greater than 0. A detailed explanation of how the selection mechanics works is out of the scope of this paper. The subject is explained in depth in (Šlajchrt, 2015b, pp.123-127).

The last statement is the invocation of the `singleton` macro, which creates a recognizer, which can be seen as a kind of factory for creating morphs. Every recognizer provides the so-called default morph, which is a lazily initialized morph associated with the recognizer. It is obtained by invoking the `~` method on the recognizer. This default morph is also the object assigned to the `savingsAcc` variable.

Although Morpheus supports passing initial values during the morph construction, here, for the sake of simplicity, the initial value is set by means of the `increaseBalance` method. Since this method modifies the state, which may influence the type of the morph, the `remorph` method is invoked. This method is a platform method present at all morphs. It re-morphs the morph by invoking the underlying recognizer, which created the morph. The recognizer determines the new form (type) of the morph in cooperation with the morphing strategy, which selects the morph alternative according to the current balance. The resulting type of the morph can be checked as follows:

```
println(savingsAcc.myAlternative)
```

It should print the following texts:

```
List(AccountBase, SavingsAccount)
```

The initialization of other accounts would be done analogously.

Modeling Roles

Modeling roles is not much different. The `Source` role is designed as a fragment declaring the self relationship with an `Account` fragment (Listing 5). In other words, this relationship declares that any morph containing a `Source` fragment instance must also contain a fragment instance implementing the `Account` interface.

```
@fragment
trait Source {
  this: Account =>

  private def withdraw(amount: BigDecimal) {
    decreaseBalance(amount)
  }

  def transfer(destination: Destination, amount: BigDecimal): Unit =
{
    destination.deposit(amount)
    withdraw(amount)
  }

}
```

Listing 5: The `Source` role

Besides the `Source` role there is also its wrapper `BlockedSource` activated only when the source account is blocked (Listing 6). This condition is expressed by the self-type that includes `BlockedAccount`. The `BlockedSource` wrapper emulates the behavior as if the transferred amount were 0.

```
@wrapper
trait BlockedSource extends Source {
  this: Account with BlockedAccount =>

  override def transfer(destination: Destination, amount:
BigDecimal): Unit = {
    println("Warning: No transfer. Account blocked.")
    destination.deposit(0)
  }

}
```

Listing 6: The `BlockedSource` wrapper activated when the source account is blocked. This condition is expressed by self-type `Account with BlockedAccount`.

The `Destination` role is designed like the `Source` role (Listing 7).

```
@fragment
trait Destination {
  this: Account =>

  def deposit(amount: BigDecimal): Unit = {
    increaseBalance(amount)
  }
}
```

Listing 7: The `Destination` role

Modeling Use-case Context

There are several ways to implement the use-case context. Here, the context is designed as a class whose constructor has three arguments (Listing 8). The first two arguments are references to the source and destination account morphs. The third argument specifies the amount transferred between the source and destination accounts.

```
class Context(srcAcc: &[$[Source] with Account with
               /?[$[BlockedSource] with BlockedAccount]],
              dstAcc: &[$[Destination] with Account],
              amount: BigDecimal) {

  val source = *(srcAcc).~
  val destination = *(dstAcc).~

  def run(): Unit = {
    source.transfer(destination, amount)
  }

}
```

Listing 8: The context. The `srcAcc` *and* `dstAcc` *morph references are the locus of the entanglement of the domain objects and the role fragments.*

The `&[M]` type is used in Morpheus to declare a reference to a morph, where `M` is an entangled morph model type consisting of the target and source fragments. To distinguish the target fragments from the source ones the `$[T]` type is used, where `T` is a target fragment. Thus, the type of the destination account argument `&[$[Destination] with Account]` declares that the reference can be assigned with a source morph of a model containing a fragment implementing the `Account` interface. This source fragment will be entangled with an instance of the target `Destination` fragment.

Similarly, the source argument type `&[$[Source] with Account with /?[$[BlockedSource] with BlockedAccount]]` contains the `/?` type to specify the optional sub-model `$[BlockedSource] with BlockedAccount`. It makes the entangled model a morph model with two alternatives: `$[Source] with Account` and `$[Source] with Account with $[BlockedSource] with BlockedAccount`. The source account reference can thus be assigned with a source morph created from a model having two alternatives: one containing `Account` and the other containing `Account` and `BlockedAccount`. The first alternative will be entangled with an instance of the target `Source` fragment. The second target alternative will have an additional instance of the `BlockedSource` target wrapper.

To use the roles, i.e. the entangled account morphs, the references must be dereferences first by means of the dereference operator `*(r)`, where `r` is a morph reference. The result of the dereference operator is not a target morph, but an instance of the recognizer responsible for creating the morph. To obtain the morph from the recognizer, the ~ method is invoked, which returns the default morph of the recognizer.

The `run` method simply mediates the interaction between the source and destination roles.

Running the Use-case

To run the use case a new instance of the `Context` class is created and the `run` method is invoked. The `savingsAcc` and `checkingAcc` account morphs are passed as the source and destination accounts to the constructor.

```
println(s"Source balance is: ${savingsAcc.balance}")
println(s"Destination balance is: ${checkingAcc.balance}")

val ctx = new Context(savingsAcc, checkingAcc, 5)
ctx.trans()

println(s"Source balance is now: ${savingsAcc.balance}")
println(s"Destination balance is now: ${checkingAcc.balance}")
```

Listing 9: The execution of the use-case

The complete source code can be downloaded from (Šlajchrt, 2016b).

4. Discussion

It is shown in the previous section how to use Morpheus to implement the paradigmatic money transfer example illustrating the ideas behind DCI.

The domain objects are implemented very straightforwardly as stateful traits annotated with the `fragment` annotation. In contrast to the original version of the example, here, the accounts may change its type and behavior dynamically. There are two alternative forms of an account: one representing a valid account and the other representing a blocked account. This mutability is described by the account morph model using a special type expression.

To instantiate the domain objects a couple of Morpheus macros are needed to parse the account morph model type expression, to create the morphing strategy and the recognizer, which serves as a factory instantiating the account morphs.

The roles are implemented similarly to the domain objects; i.e. as fragments. The Source role is a protean object having two alternative forms. Which form is used is determined by the source account's morph alternative; if the source account is blocked the source role will contain the `BlockedSource` wrapper. In contrast to the source role the destination role is monomorphic.

The use-case logic is written in pure Scala and does not require special knowledge of Morpheus. Morpheus is only needed to specify the role references and to dereference them to get the role instances. The use case logic is localized in one artifact and is not scattered across the domain classes.

It should be also noted that the roles are not instantiated by means of delegation or other forms of wrapping causing object schizophrenia due to the two identities – the wrapped account and the wrapping role – composing the role. Instead, the composition of the account and its role in Morpheus could be described as inserting the role fragment instances into the array of the account fragment instances at the right positions making so the role morph. This composition might be also described as a dynamic version of the Cake Pattern, which is used to compose classes from traits utilizing the self-type in Scala (Rollins, 2014).

A significant advantage of using Morpheus is that it combines static code analysis with dynamic composition of objects. This might be called *controlled dynamism* since, in contrast to the languages with dynamic traits, such as Groovy, Morpheus uses statically validated morph models to compose and re-compose objects at runtime. Thus, it rules out a creation of an invalid composition of fragments since the validated model guarantees that all fragment combinations are valid.

The downside of the described approach is primarily its performance, which might be attributed to the fact that Morpheus is still in the proof-of-concept stage. Other problems concern the readability of the code and the model of morphing strategies, which is still rather immature.

If the concept of Object Morphology proves vital it could possibly give rise to a new language based on the experience acquired by using Morpheus to develop complex "protean" applications.

Conclusion

The main goal of this paper was to demonstrate how the concept of Object Morphology and, in particular, its proof-of-concept implementation Morpheus, can be used to develop applications following the DCI principles. Although, there have been several implementations of DCI based on various programming languages, most of them struggle in the attempt to fully implement DCI. It may be concluded that the main reason is that the DCI ideas do not fit much to the traditional concept of object-oriented programming.

On the other hand, Object Morphology, which is an alternate object-oriented paradigm specialized on modeling protean phenomena, may offer new perspectives on problems solved in the framework of the traditional OOP only with difficulties, the DCI being one of such problems.

To show that OM and Morpheus are really capable of providing a DCI framework in a natural way, they are used to analyze and implement the paradigmatic DCI example: the money transfer. Despite being a very simple use case, it is often cited and used to elucidate the basic DCI principles and ideas.

The conclusion is that the implementation of the example was very straightforward and the resulting program followed the principles of DCI. The domain objects and the roles were modeled by means of morph models assembled from special artifacts called fragments. The context of the use case was implemented as a simple class encapsulating the interaction between the roles.

Among the downsides of the presented approach, the performance is the most significant. It might be attributed to the immaturity of Morpheus that is still in the proof-of-concept stage.

Future work should therefore focus on optimizing Morpheus' performance and also on improving the usability of the whole framework. There is also the potential for the development of a flexible and type-safe new programing language based on the OM paradigm utilizing the experience gained from applications of Morpheus in complex use cases.

References

[1] Coad, P., North, D. and Mayfield, M., 1997. *Object Models: Strategies, Patterns, and Applications*, Yourdon Press, Upper Saddle River.

[2] Coplien, J. and Bjørnvig, G., 2010. *Lean Architecture: For Agile Software Development*, 1st Edition, Wiley, West Sussex.

[3] Coplien, J. and Reenskaug, T., 2012. The data, context and interaction paradigm. In: *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12*. Tucson, AZ, USA, October 21-25, 2012, ISBN 978-1-4503-1563-0, pp.227 – 228.

[4] FullOO.info. 2016. *Existing DCI Implementations*. Available at: <http://fulloo.info/doku.php?id=existing_dci_implementations> [Accessed 3 April 2016].

[5] Groovy Documentation. 2016. *Traits*. Available at: <http://docs.groovy-lang.org/latest/html/documentation/core-traits.html> [Accessed 2 April 2016].

[6] Harrison, W. and Ossher, H., 1993. Subject-oriented programming (a critique of pure objects). In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)*. Washington, D.C.

[7] Madsen, O., 2006. *Open Issues in Object-Oriented Programming, A Scandinavian Perspective*. Computer Science Dept., Aarhus University. Available at: <https://users-cs.au.dk/olm/index.html/PUB/OpenIssuesInOO.pdf> [Accessed 25 March 2016].

[8] Hasso, S. and Carlson, C., 2014. Software composition using behavioral models of design patterns. In: *Journal of Software Engineering and Applications, 7(2)*, pp. 104-117.

[9] Reenskaug, T. (2008). *The Common Sense of Object Orientated Programming*. Available at: <http://folk.uio.no/trygver/2008/commonsense.pdf> [Accessed 3 April 2016].

[10] Reenskaug, T. and Coplien, J., 2009. *The DCI Architecture: A New Vision of Object-Oriented Programming*. Available at: <http://www.artima.com/articles/dci_vision.html> [Accessed 3 April 2016].

[11] Rollins, A., 2014. *The Cake Pattern in Scala - Self Type Annotations vs. Inheritance.* Available at: <http://www.andrewrollins.com/2014/08/07/scala-cake-pattern-self-type-annotations-vs-inheritance/> [Accessed 3 April 2016].

[12] Savkin, V., 2012. *Data Context Interaction: The Evolution of the Object Oriented Paradigm.* Available at: <http://www.sitepoint.com/dci-the-evolution-of-the-object-oriented-paradigm/> [Accessed 3 April 2016].

[13] Sekharaiah, K. and Ram, D., 2002. *Object Schizophrenia Problem in Modeling Is-role-of Inheritance.* Available at: <http://users.jyu.fi/~sakkinen/inhws/papers/Sekharaiah.pdf> [Accessed 3 April 2016].

[14] Steimann, F., 2006. The paradoxical success of aspect-oriented programming. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. ACM, New York, NY, USA, pp. 481-497.

[15] Šlajchrt, Z., 2015. *Morpheus, A Proof-of-Concept Implementation of Object Morphology in Scala*. Available at: <https://github.com/zslajchrt/morpheus> [Accessed 3 April 2016].

[16] Šlajchrt, Z., 2015. *Object Morphology*. A draft of the dissertation thesis. Available at: <https://github.com/zslajchrt/morpheus/raw/master/src/main/doc/thesis/thesis.pdf> [Accessed 3 April 2016].

[17] Šlajchrt, Z., 2016. *A Solution to the Square-Rectangle Problem Within the Framework of Object Morphology*. Available at: <https://github.com/zslajchrt/morpheus/raw/master/src/main/doc/papers/acta/square-rectangle-problem.pdf> [Accessed 3 April 2016].

[18] Šlajchrt, Z., 2016. *DCI over Morpheus Source Code*. Available at: <https://github.com/zslajchrt/morpheus-tutor/tree/master/src/main/scala/org/cloudio/morpheus/dci/moneyTransfer>.

[19] Wikipedia, 2015. *Subject-oriented Programming*. Available at: <https://en.wikipedia.org/wiki/Subject-oriented_programming> [Accessed 3 April 2016].

[20] Wikipedia, 2016. *Data, context and interactions*. Available at: <https://en.wikipedia.org/wiki/Data,_context_and_interaction#cite_ref-20> [Accessed 3 April 2016].

[21] Wikipedia, 2016. *Circle-Ellipse Problem*. Available at: <https://en.wikipedia.org/wiki/Circle-ellipse_problem> [Accessed 3 April 2016].

[22] Wirfs-Brock, R. and McKean, A., 2003. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley, Boston.