BSE Seminar

Sandor Szücs

sandor.szuecs@fu-berlin.de

Berlin, 24. Oktober 2010

Department of Mathematics and Computer Science
Institute for Computer Science
Software Engineering Working Group

Responsible University Professor: Prof. Dr. Lutz Prechelt

Inhaltsverzeichnis

1 Einführung

DCI, Data Context Interaction, ist eine neue Software Architektur, die das Ziel hat Kommunikation zwischen Objekten verständlich zu implementieren. DCI unterstützt die Umsetzung von Anwendungsfällen, so dass Durchsicht und Implementierung an einer Stelle möglich ist. DCI ist der bekannten MVC Architektur nahe, verallgemeinert den Ansatz aber auf die Implementierung von Anwendungsfällen[1] [Ree09]

Die Ziele von DCI sind es für folgende Probleme ein Lösungsansatz zu bieten:

- Anwendungsfälle sind häufig verstreut im Code, daher ist der verwendete Algorithmus nicht leicht nachvollziehbar. Eine Durchsicht der Anwendungslogik von Domänenexperten ist in der Praxis nur schwer möglich.
- Dynamische Interaktion von Objekten ist nicht gut in Klassen abbildbar. Das Beispielvideo *animate-arrows*[2] zeigt verschiedene Stern und Kreis Objekte, die durch Pfeile verbunden werden. Die Interaktion verbinden von 2 Formobjekten mit Hilfe eines Pfeils. Diese wird, laut Trygve Reenskaug, bisher mit Anwendung von objektorientierter Programmierung nur unlesbar implementiert.
- Das mentale Modell über die Funktionsweise der Software von Benutzer und Programmierer unterscheiden sich erheblich voneinander.

DCI ist ein Ansatz die genannten Probleme zu lösen, indem es eine Unterscheidung zwischen Datenobjekten, Kontextaufbau und Interaktion zwischen Datenobjekten erlaubt.

Datenobjekte sind einfache Objekte mit privaten Attributen. Methoden der Datenobjekte arbeiten ausschließlich auf den Daten des Objekts.

Kontextobjekte stellen zwei oder mehrere Objekte in Beziehung zu einander. Dies entspricht einer Rollenzuweisung für die Datenobjekte in einem konkreten Anwendungsfall. Ein Kontext vergibt einem Objekt zur Laufzeit die entsprechende Rolle.

Interaktionen spezifizieren das Laufzeitverhalten zwischen den Rollen. Es wird eine Systemoperation, die mehr als ein Objekt umfaßt implementiert. Die Interaktion kennt keine Datenobjekte, sie verwendet ausschließlich Rollen.

Eine Rolle definiert den Zweck eines Objekts. Jedem Objekt wird bei Eintritt in einen Kontext eine Rolle zu gewiesen. Rollen leben daher innerhalb des Kontexts und maskieren die Objekte. Die Idee ist eine Interaktion unabhängig von den beteiligten Objekten zu machen, um Wiederverwendung eines Szenarios zu ermöglichen.

Ein Szenario entspricht genau einer trigger[3] Methode, die die Interaktion zwischen zwei oder mehr Rollen beschreibt. Man kann den Szenario implementierenden Algorithmus 'Top-Down' lesen, was eine Durchsicht mit Hilfe eines Domänenexperten erleichtert.

[1] Anwendungsfall und Use Case werden im Text synonym verwendet.
[2] Video: http://heim.ifi.uio.no/~trygver/assets/animate-arrows.mov oder http://heim.ifi.uio.no/~trygver/assets/animate-arrows.mpeg
[3] Eine Methode des Kontexts, die ein Szenario auslöst.

Ein Use Case gruppiert verschiedene zusammenhängende Szenarien. Ein Kontext kapselt die entsprechenden Implementierungen auf natürliche Art und Weise.

Die am häufigsten in Bezug auf DCI erwähnte Beispielanwendung implementiert das Szenario Geldtransfer von einem Quellkonto auf ein Zielkonto. Im weiteren Text wird dieses Beispiel zur Erklärung der Bestandteile verwendet. Abschnitt 2 erklärt die Begriffe detailierter, bevor in Abschnitt 3 eine Variante der Implementierung dieses Szenario gezeigt wird. In Abschnitt 4 wird ein Werkzeug für die anschauliche Darstellung von DCI vorgestellt.

2 Theorie

Im folgenden Abschnitt werden die anfangs erwähnten Begriffe genauer erklärt.

2.1 Data

Datenobjekte zeigen aus *was* das System besteht.
Ein Datenobjekt repräsentiert das Modell der Domänen-Information. Es kapselt private Attribute und stellt gegebenenfalls öffentliche Getter- und Setter-Methoden bereit, falls Rollen Zugriff auf Attribute benötigen. Ein Datenobjekt hat ausschließlich Methoden, die auf den eigenen Daten operieren. Das 'D' in DCI verhält sich ähnlich zum 'M' des bekannten MVC Muster.

2.2 Context

Ein Kontext repräsentiert das Modell einer Systemoperation. Eine Systemoperation wird in einem Objektnetz von mehreren Objekten ausgeführt. Ein Kontextobjekt definiert eine Topologie zwischen den beteiligten Objekten der Systemoperation.
Einem Objekt wird zur Laufzeit eine Rolle zugewiesen. Methoden, die so eine Zuweisung vornehmen gehören zum Kontextobjekt und lassen sich als `role binding`[4] Methoden klassifizieren. Rollen sind Attribute des Kontexts. Sie bleiben dem Kontext solange erhalten bis das entsprechende Szenario endet. Nach der Rollenzuweisung werden die entsprechenden Rollenmethoden[5] in die Datenobjekte injiziert. Dies wird typischerweise mit Hilfe eines Mixin [BC90] oder Traits [SND+03] vorgenommen. Mixins und Traits enthalten Implementierungsdetails und können verschiedenen Klassen oder Objekten hinzugefügt werden.

2.3 Interaction

Eine Interaktion beschreibt das Systemverhalten.
Das Systemverhalten wird durch Interaktion zwischen verschiedenen Rollen erreicht. Die Implementierung einer Systemoperation, bzw. Szenario, steht in einer Rollenmethode. Der Algorithmus der Systemoperation kann dabei innerhalb einer Methode von oben nach unten gelesen werden. Interaktion ist eine Sicht auf die Implmentierung, die die Kollaboration der Teilnehmer übersichtlich darstellt. Dies ermöglicht das Schreiben

[4] `role binding` wurde von Trygve Reenskaug als Name innerhalb der BabyIDE verwendet.
[5] Rollenmethoden werden in Abschnitt 2.5 erklärt.

von Quellcode, der auch für einen Domänenexperten auffindbar, lesbar und verständlich ist. Eine Durchsicht ist dadurch wesentlich erleichtert.

2.4 Rollen

Eine Rolle definiert den Sinn eines Objekts innerhalb eines Szenarios. Rollen sind Attribute des Kontexts, die in der Realität Referenzen auf die zugewiesenen Objekte sind. Rollen haben keine eigenen Attribute. Der Zugriff auf Attribute des Datenobjekts muss der Rolle über Getter- und Setter-Methoden explizit erlaubt werden.

In einer in Bezug auf DCI idealen Programmiersprache wäre eine Rolle ein Kernbestandteil der Sprache, ähnlich wie eine Klasse in klassenorientierten OO-Sprachen wie Java. Objekte haben dabei zu Rollen eine n zu m Beziehung.
Beispiel: Geldtransaktion von einem Konto-Objekt S zu einem Konto-Objekt T.

- S hat die Rolle *Quellkonto*
- T verwendet die Rolle *Zielkonto*

Zu einem späteren Zeitpunkt können S und T die Rollen tauschen, was die n zu m Beziehung verdeutlicht.

2.5 Rollenmethoden

Rollenmethoden sind Methoden die Systemverhalten implementieren. Datenobjekte dürfen Rollenmethoden nicht überschreiben, da diese Methoden sicherstellen sollen, dass es keine versteckten Details gibt. Die Autoren empfehlen daher Rollenmethoden zur Laufzeit in die entsprechenden Rollenobjekte zu injizieren. Die Injektion überschreibt eventuell vorhandene gleichnamige Methoden des Datenobjekts, so dass der Methodenaufruf eindeutig sichergestellt ist. Eine solche Injektion wird durch die Erweiterung mit Hilfe von Mixins [BC90] oder Traits [SND+03] vereinfacht und im Abschnitt 3 gezeigt.

2.6 Use Case

DCI ist eine Use Case zentrierte Architektur, die es dem Entwickler erleichtert Anwendungsfälle zu lesen und zu schreiben.[Ree09] Ein Use Case im Sinne von DCI ist eine Gruppe zusammenhängender Szenarien. Szenarien werden beschrieben durch Akteure, die miteinander kommunizieren. Die Kommunikation ist mit DCI verständlich implementierbar:

- Ein Use Case wird im Quellcode auf ein Kontextobjekt abgebildet.
- Ein Szenario wird durch eine Rollenmethode umgesetzt.
- Ein Kommando der Applikation triggert die Ausführung einer Interaktion, die in einer Rollenmethode implmentiert ist.

Falls jegliche Kommunikation des Systems nur in eine Richtung geht, dann ist die Verwendung von DCI nicht notwendig. Diese Einwegkommunikation wird auch als *atomic event architecture* beschrieben.[CB10, OCU]

3 Anwendung

Eine DCI Architektur wird durch Sprachkonstrukte wie Mixin [BC90] oder Trait [SND+03] erheblich erleichtert. Aus diesem Grund habe ich mich dafür entschieden eine Beispielimplementierung in der Sprache Ruby vorzustellen die Mixins unterstützt. Es gibt keine Mixins oder Traits in Java, daher wird an dieser Stelle nur auf das Qi4J[6] Framework hingewiesen, das Traits und Mixins implementiert. In Java kann DCI mit der Hilfe von Qi4J leicht umgesetzt werden.

Im nächsten Abschnitt wird eine DCI Implementierung des Transaktionsbeispiels gezeigt, das häufig in Diskussionen verwendet wird.

3.1 Beispiel - Transaktion

Das Beispiel umfaßt zwei Konto Datenobjekte und ein Transaktionskontext. Der Anwendungsfall ist die Transaktion eines Betrags von einem Quellkonto auf ein Zielkonto.

3.1.1 Data

In Listing 1 sind die zwei Datenklassen `SavingsAccount` und `CheckingsAccount` zu sehen. Die Implementierung bekommen sie aus der Basisklasse `Account` über Vererbung. Die Konto Datenobjekte haben ein Attribut `@balance`, eine Gettermethode `balance`, einen Konstruktor `initialize` und drei Verhaltensmethoden. Die Verhaltensmethoden `withdraw` und `deposit` ändern das zum Objekt gehörende Attribut `@balance`. Die dritte Methode `update_log` schreibt einen Transaktionslog auf die Standardausgabe. Die drei Methoden spezifizieren nur lokales Verhalten und sind daher einfach zu verstehen.

3.1.2 Context

Die Implementierung des Kontexts ist in zwei Teile zerlegt. Das Hilfsmodul `ContextAccessor` aus Auflistung 3 ermöglicht den Zugriff auf die Getter Methoden des `TransferMoneyContext` und implementiert den Kontextwechsel in der Methode `execute_in_context`. Mit Hilfe der Delegation in `method_missing` werden alle undefinierten Methoden auf das Kontextobjekt umgeleitet. Dieses Helfermixin erleichtert die Kontextimplementierung und kann für alle weiteren Kontextklassen oder Kontextobjekte wiederverwendet werden.

Der Kontext des Anwendungsfalls Geldtransaktion ist in Auflistung 2 zu sehen. Das Kontextobjekt bekommt bei der Erzeugung einen Betrag `amount`, eine Quellkonto `source` und ein Zielkonto `destination`. Die Parameter werden im Konstruktor des Kontexts als Attribute gespeichert. Die Attribute sind Referenzen auf die Datenobjekte. Sie sind Akteure und lokale Daten für die anstehende Interaktion. Das setzen der Attribute ist die Rollenzuweisung (*role binding*). Als letztes werden den Objekten die Rollenmethoden injiziert. Die Methode `extend` erweitert ein Objekt, um das übergebene Modul, in diesem Fall `MoneySource` und `MoneySink`.

[6] Qi4j: `http://www.qi4j.org/`

```
class Account
  # getter for attribute @balance
  attr_reader :balance

  def initialize
    @balance = 0
  end

  def withdraw(amount)
    raise "Insufficient funds" if amount < 0
    @balance -= amount
  end

  def deposit(amount)
    @balance += amount
  end

  def update_log(msg, date, amount)
    puts "#{self.class}: #{msg}, #{date.to_s}, $#{amount}"
  end
end

class SavingsAccount < Account; end
class CheckingAccount < Account; end
```

Abbildung 1: Datenklassen `SavingsAccount` und `CheckingsAccount` werden durch Vererbung implementiert.

Die Instanzmethode `transfer` führt die Transaktion im vorher festgelegten Kontext auf der `MoneySource` Rolle aus. Methoden der Datenobjekte sind per Delegation transparent über die Kontextreferenzen erreichbar.

3.1.3 Interaction

Der Algorithmus für den Anwendungsfall Geldtransfer ist in Auflistung 4 abgebildet.

Das Datenobjekt, das sich hinter der `MoneySource` Rolle verbirgt, ist über die `self` Referenz erreichbar. Methoden des verborgenen Datenobjekts können daher eigentlich nicht direkt aufgerufen werden.

Zugriffe auf das Zielkonto sind über den Getter `context` zu erreichen. Durch die Delegation mit Hilfe der Kontexthilfe werden die Kontext-Gettermethoden `source`, `destination` und `amount` direkt aus der Rollenmethode `transfer` erreichbar.

In der DCI basierten Implementierung des Anwendungsfalls Geldtransfer steht das Domänenverhalten in der Rollenmethode. Alle relevanten Daten sind dort enthalten, was die Lesbarkeit steigert. Ein Domänenexperte ist in der Lage den Vorgang zu prüfen.

3.1.4 Ausführung

Das Hauptprogemm aus Auflistung 5 erzeugt zwei verschiedene Kontoobjekte für Alice und Bob. Bob bekommt eine Anzahlung von $1000, bevor er die Überweisung von

```
require "context_helper"
class TransferMoneyContext
  # getter methods
  attr_reader :source, :destination, :amount
  include ContextAccessor # static mixin injection

  # syntactical sugar TransferMoneyContext::transfer
  def self.transfer(amount, src, dst)
    TransferMoneyContext.new(amount, src, dst).transfer
  end

  def initialize(amount, source, destination)
    # role bindings
    @amount = amount
    @source = source
    @destination = destination

    # rolemethod injection
    @source.extend MoneySource
    @destination.extend MoneySink
  end

  # instance method
  def transfer
    execute_in_context do
      source.transfer
    end
  end
end
```

Abbildung 2: Getter Methoden und Ausführung der Interaktion `transfer` innerhalb des Blocks der Methode `execute_in_context`, ermöglicht die Implementierung des Szenarios in einer wiederverwendbaren Methode.

```ruby
module ContextAccessor
  def context
    Thread.current[:context]
  end
  def context=(ctx)
    Thread.current[:context] = ctx
  end
  def execute_in_context(&blk)
    if block_given?
      old_context = self.context
      self.context = self
      blk.call
      self.context = old_context
    end
  end
  # unknown message m delegate to context.m
  def method_missing(m, *args, &blk)
    context.send(m, *args, &blk)
  end
end
```

Abbildung 3: Das wiederverwendbare Kontext-Hilfs-Modul speichert den alten Kontext vor der Ausführung von Rollenmethoden aus dem übergebenen Block in einer lokalen Variable des aktuellen Threads. In Kontextobjekten, die das gezeigte Mixin verwenden, wird mittels `execute_in_context` die Ausführung eines Szenarios begonnen. Die Methode `method_missing` deligiert Nachrichten an das Kontextobjekt.

```ruby
require "context_helper"
module MoneySource
  include ContextAccessor

  def transfer
    raise "Insufficient Funds" if source.balance < amount
    source.withdraw amount
    destination.deposit amount
    source.update_log "Transfer Out", Time.now, amount
    destination.update_log "Transfer In", Time.now, amount
  end
end
module MoneySink; end
```

Abbildung 4: Das Szenario Geldtransfer ist in der Methode `transfer` zu sehen.

```
#!/usr/bin/env ruby

require 'data'
require 'context'
require 'interaction'

bob = SavingsAccount.new
bob.deposit(1_000)
alice = CheckingAccount.new
puts "#{bob.class}: $#{bob.balance}, #{alice.class}: $#{alice.balance}"

# transfer money from one account to the other
# bob ---200---> alice
TransferMoneyContext.transfer(200, bob, alice)
puts "#{bob.class}: $#{bob.balance}, #{alice.class}: $#{alice.balance}"
```

Abbildung 5: Hauptprogramm

```
$ ruby money_transfer_application.rb
SavingsAccount: $1000, CheckingAccount: $0
SavingsAccount: Transfer Out, Sat Oct 23 17:01:25 +0200 2010, $200
CheckingAccount: Transfer In, Sat Oct 23 17:01:25 +0200 2010, $200
SavingsAccount: $800, CheckingAccount: $200
```

Abbildung 6: Programmausführung mit Ausgabe

$200 von an Alice ausführt. In der Mitte der Ausgabe in Abbildung 6 sind die Transaktionslogs zu sehen.

4 Werkzeugunterstützung

Trygve Reenskaug hat in seinem Test eine IDE[7] für DCI in Squeak[8] implementiert. Die BabyIDE [Ree09] enthält drei Grundansichten auf den Programmcode je eine für Data, Context und Interaction. Die Ansicht der IDE ist auch nach Bedarf erweiterbar. Im Papier [Ree09] sind mehrere Beispiele an Erweiterungen gezeigt. In der Beispielumgebung BabyIDE[9] können die Beispiele begutachtet werden.

4.1 Datensicht

Abbildung 7 zeigt die Ansicht auf die Datendomäne. Es können, wie in IDEs üblich, die definierten Methoden eines Objekts betrachtet werden. Methoden werden zu Kategorien gruppiert. Es gibt die Gruppe der *access* Methoden, die im Beispiel nicht nur Getter und Setter Methoden enthalten, sondern auch zustandsändernde Methoden wie `decrease` und `increase`.

[7] IDE: integrated development environment
[8] Squeak ist eine freie Implementierung der Programmiersprachenfamilie Smalltalk.
[9] http://heim.ifi.uio.no/~trygver/themes/babyide/baby-downloads.html

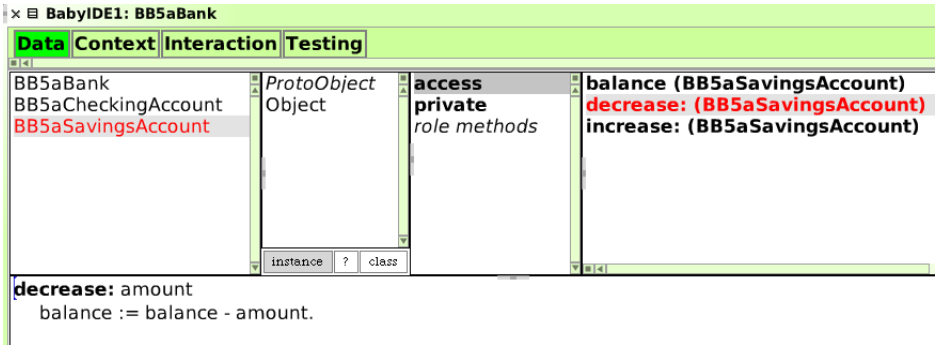

Abbildung 7: Daten-Sicht: Datenobjekte sind oben links zu sehen. Ausgewählt ist die `access` Methode `decrease`, des Objekts `BB5aSavingsAccount`. Im unteren Teil ist die Implementierung der ausgewählten Methode zu sehen

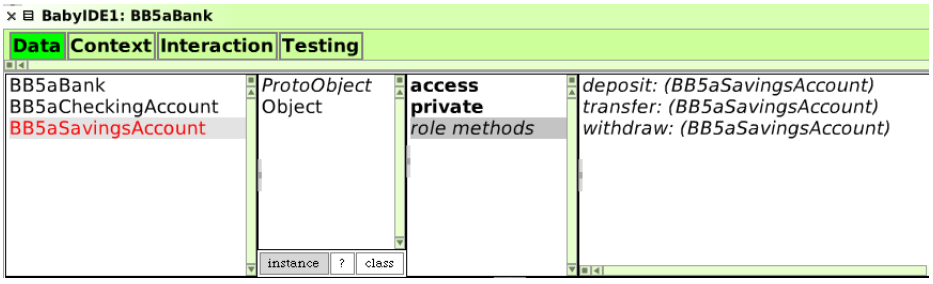

Abbildung 8: Daten-Sicht: Eine Liste von Rollenmethoden ist im Fenster rechts oben zu sehen.

Die kursiv gedruckte Kategorie `role methods` listet alle korrespondierenden Rollenmethoden des Datenobjekts in Abbildung 8 auf. Rollenmethoden implmentieren die Szenarien, eines ausgewählten Objekts. Szenarien von Datenobjekten kann man dadurch leicht nachvollziehen.

Die markierte Zeile aus Abbildung 9 zeigt die Rollen des gewählten Datenobjekts als Eigenschaft *uses* an. Man bekommt eine Übersicht welche Rollen ein Datenobjekt ausfüllt, ohne dass viel Quellcode gelesen werden muss.

4.2 Kontextsicht

In der Kontextansicht aus Abbildung 10 sieht man die zwei notwendigen Rollen für den Kontext einer Geldtransaktion. Die Rolle `TransferMoneySink` ist das Zielkonto der Transaktion. `TransferMoneySource` ist die Rolle des Quellkontos.

Abbildung 11 zeigt die dazu gehörige Rollenzuweisung der *trigger* Methode `transfer` an. Dazu ist die Implementierung der Rollenzuweisung und die Ausführung der Interaktion in diesem Kontext zu sehen. Der Transfer wird aus Sicht der `TransferMoneySource` Rolle ausgeführt, d.h. die `self` Referenz zeigt auf `TransferMoneySource`.

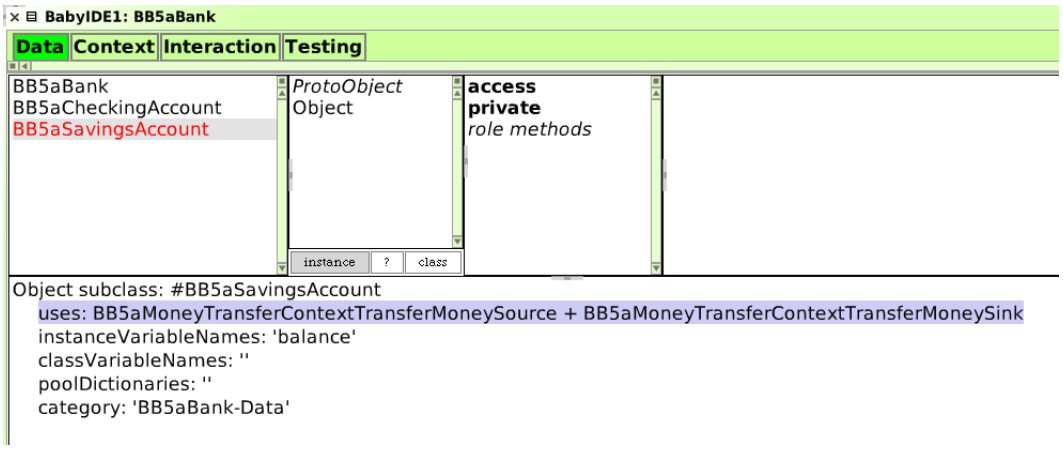

Abbildung 9: Daten-Sicht: Die markierte Zeile zeigt alle Rollen des Objekts.

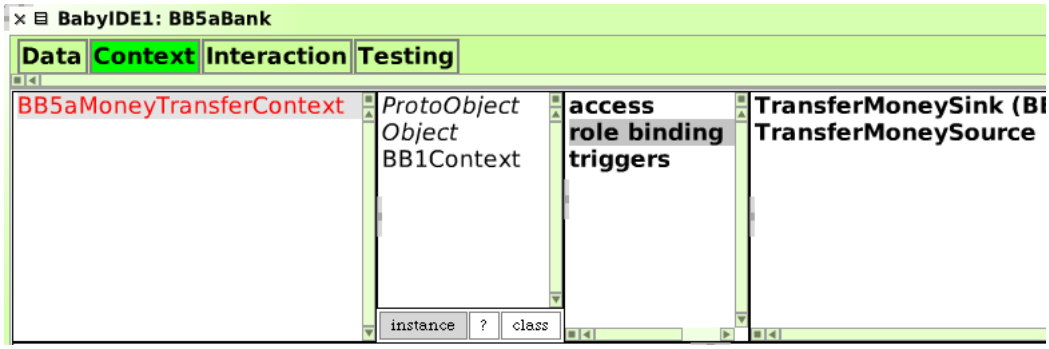

Abbildung 10: Die Kontextansicht zeigt zwei Rollen, `TransferMoneySink` und `TransferMoneySource`, die im Kontext `MoneyTransferContext` existieren.

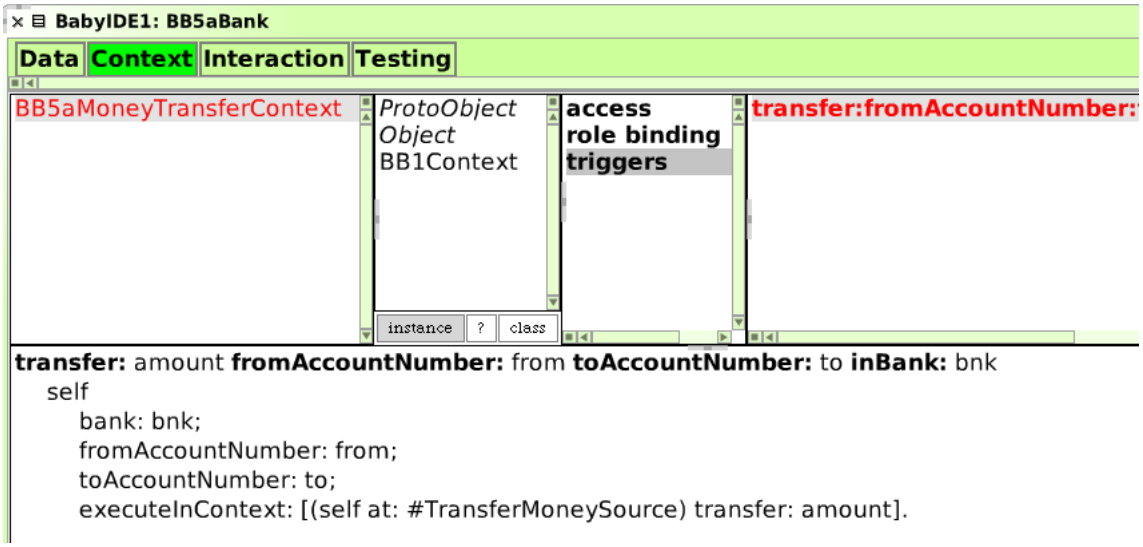

Abbildung 11: Die Kontextansicht zeigt die *trigger* Methode `transfer`, die vor der Ausführung des Szenario die beteiligten Objekte in den richtigen Kontext setzt. Zum Schluss wird der Transfer aus Sicht der `TransferMoneySource` Rolle ausgeführt.

4.3 Interaktionssicht

In Abbildung 12 ist die Interaktion `transfer` zwischen zwei Rollen `TransferMoneySource` und `TransferMoneySink` aus der Sicht der Rolle `TransferMoneySource` zu sehen. Der Anwendungsfall Geldtransfer ist in zwei Teilschritte, Szenarien, unterteilt. Wie in Abbildung 12 gezeigt übernimmt die Rollenmethode `transfer` die Ausführung. `Self` ist bezogen auf die aktuell ausgewählte Rolle, d.h. `withdraw: amount` ist eine Nachricht an `TransferMoneySource`. Danach wird die Rolle `TransferMoneySink` informiert den Betrag des Zielkontos um die gleiche Zahl zu erhöhen. Die Implementierung der Nachricht `deposit: amount` und die Sicht aus der Perspektive der `TransferMoneySink` Rolle wird in Abbildung 13 gezeigt.

Die Interaktionsansicht zeigt alle Anwendungsfälle des Kontexts `BB5aMoneyTransferContext` sehr übersichtlich. Die Korrektheit der Implementierung von Anwendungsfällen kann mit Hilfe dieser Sicht erheblich leichter nachvollzogen werden, als mit einer bisher üblichen IDE.

5 Fazit

DCI trennt Daten von den implementierten Anwendungsfällen. Verschiedene Datenobjekte erhalten Rollen vor der Ausführung eines Anwendungsfalls. Die Rollen sind in einem Kontext definiert in dessen Kontext die Interaktion zwischen den Rollen stattfindet. DCI kann in objektorientierten Programmiersprachen wie in diesem Papier gezeigt mit Ruby durchgeführt werden. Öffentlich zugängliche Beispielimplmentierungen[10] in Sprachen wie C++, Java, Scala oder PHP existieren. Das gezeigte Beispiel aus Abschnitt 3 ist ursprünglich ebenfalls aus der Quelle[11]. Das Beispiel wurde von mir zur

[10] https://groups.google.com/group/object-composition/files
[11] https://groups.google.com/group/object-composition/files

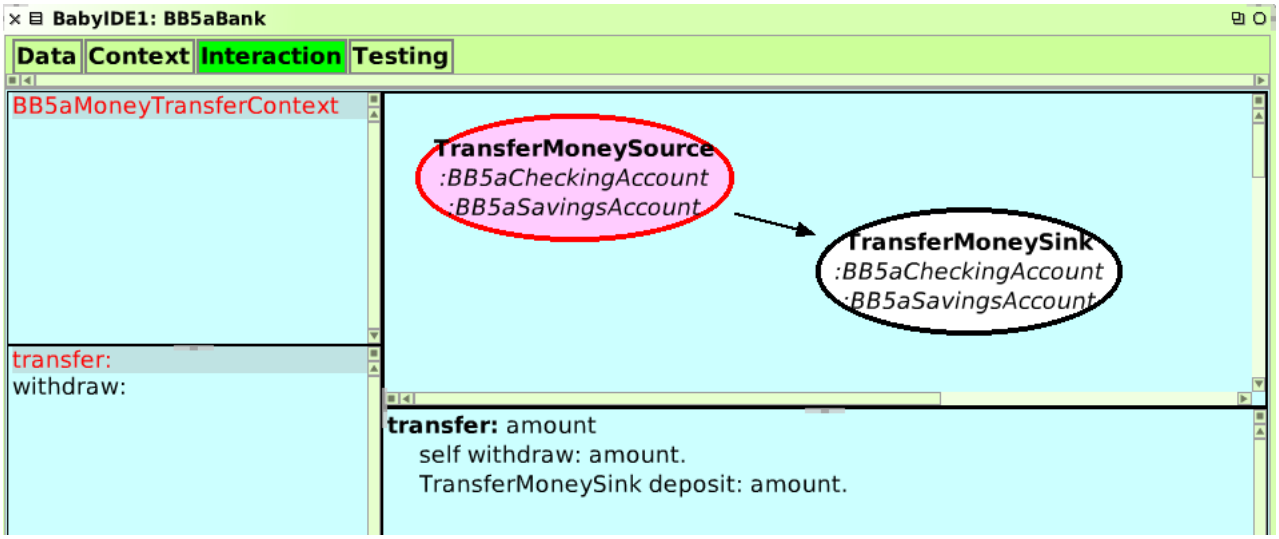

Abbildung 12: Interaktions-Sicht der BabyIDE von Trygve Reenskaug. Das Kontextobjekt hält zwei Rollen, die grafisch dargestellt sind. Die Methoden `transfer` und `withdraw` der Rolle `TransferMoneySource` sind links unten aufgelistet. Im Teilfenster unten rechts befindet sich die Implementierung der Rollenmethode `transfer`.

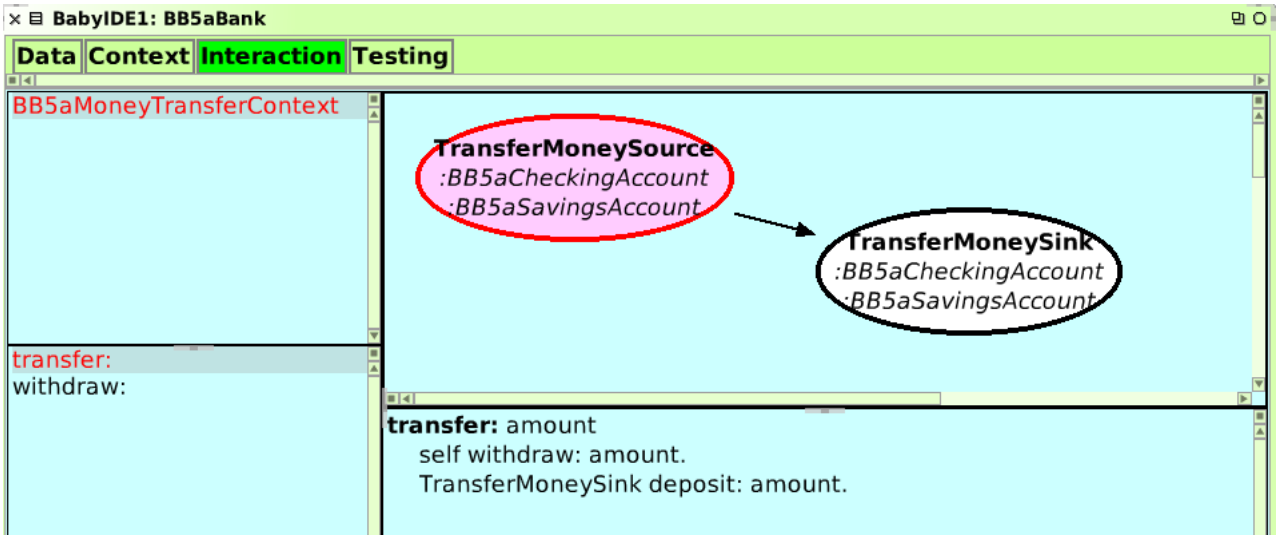

Abbildung 13: Interaktion: Szenario Geldtransfer aus Sicht des Empfängers `TransferMoneySink`.

besseren Lesbarkeit modifiziert.

Zur besseren Unterstützung von DCI hat Trygve Reenskaug eine Beispiel IDE für Squeak Smalltalk entwickelt, die die Verwendung der DCI Architektur unterstützt. Eine fachliche Durchsicht ist meiner Meinung nach durch die BabyIDE erheblich vereinfacht, da die Trennung der Sichten den Fokus auf die Sicht der Anwendungslogik ermöglicht. Ein Domänenexperte wird nicht mit unnötigem Programmcode abgelenkt und kann sich auf das Verhalten der Rollen inerhalb der Interaktionssicht konzentrieren.

DCI sehe ich als gute Architektur im Bereich von komplexen Anwendungsschichten. Ich denke es ist wahrscheinlich, dass DCI Architekturen schon einige Zeit aktiv in Verwendung sind, ohne eine theoretische Grundlage gehabt zu haben.

Literatur

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM.

[CB10] Jim Coplien and Gertrud Bjørnvig. *Lean Architecture: for Agile Software Development*. Wiley, 2010.

[OCU] Object composition group. http://groups.google.com/group/object-composition/browse_thread/thread/513deeef009cf28d. (Online abgerufen am 7.10.2010).

[Ree09] Trygve Reenskaug. The common sense of object oriented programming. Technical report, Department of Informatics, University of Oslo, Norway, 2009.

[SND+03] Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black. Traits: The formal model. Technical report, Universität Bern, 2003.