Comments on the essential mechanisms for DCI implementations

Trygve Reenskaug,
Dept. of Informatics, University of Oslo, Norway
Home: http://folk.uio.no/trygver
E-mail: trygver@ifi.uio.no

Full utilization of the DCI paradigm requires two essential mechanisms.

1. *Programming with roles*. This applies equally to methodless and methodful roles. The mechanism includes the naming of roles, the ability to reference roles in the source code, and the binding of roles to objects when needed at runtime.
2. *Methodful roles*. We associate methods with roles and inject these methods into the objects playing the roles or their classes.

We also need an IDE that supports using DCI as a programmer's mental model, but do not discuss it here.

The GOF Design Patterns book[1] p. 22:

> *An object-oriented program's runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. The* ***runtime structure consists of rapidly changing networks of communicating objects****.* (GOF p. 22, my highlighting.)

and

> *..it's clear that the code won't reveal everything about how a system will work.*

The problem is the misuse of polymorphism. Polymorphism is a great invention. But too much and too little of a good thing spoils it.

Too much polymorphism - our programs get out of control because the code will not reveal the identity of the communicating objects, their behavior, and their dynamic structure.

Too little polymorphism - and we lose the encapsulation of state and behavior. We lose the ability to "call by intent"; to separate a service request (message) from a corresponding service implementation (method).

DCI offers two mechanisms between massive polymorphism and no polymorphism: *Programming with roles* and *Methodful roles*.

1.Gamma, E; Helm, R; Johonson, R; Vlissides, J: *Design Patterns*; ISBN 0-201-63361-; Addison-Wesley, Reading, MA. 1995.

1 Programming with roles

In DCI, runtime *networks of communicating objects* are replaced by structurally similar compile-time *networks of interconnected roles*. We need to name the roles, to enable the use of the role names in code, and to replace the names with actual objects at runtime. This applies to all roles, be they methodless or methodful. The solution advocated by the DCI paradigm is to create a Context class that has these responsibilities.

1. 1 Use role names in code

We would like to write code such as (underlined symbols are role names):

```
1.  transfer (amount)
2.      sourceAccount.withdraw amount
3.      destinationAccount.deposit amount
```

or with Smalltalk syntax;

```
4.  transfer: amount
5.      sourceAccount withdraw: amount.
6.      destinationAccount deposit: amount.
```

The simplest way to achieve this is to follow the GOF Mediator pattern or the UML Collaboration metaclass. In both, the roles are attributes of a Mediator/Collaboration class and the system interaction is controlled by methods in this class.

Example due to Steen Lehmann:

```
7.  class TransferMoneyContext
8.      attr_reader :source_account, :destination_account, :amount
```

Figure 1(A) shows a simple interaction. The roles are instance variables in the *TransferMoneyContext* (Mediator) object. The vertical rectangle on the Mediator life line denotes the execution of the Mediator interaction method.

```
9.  @source_account.transferTo amount, @destination_account
```

Other methods in the TransferMoneyContext class assign values to the role variables.

This structure is extreme centralization; all logic is concentrated in one class.

Figure 1(B), is a variant of the above. Knowledge about the roles is still centralized in the Mediator object, but it delegates to the *sourceAccount* to do the actual transfer:

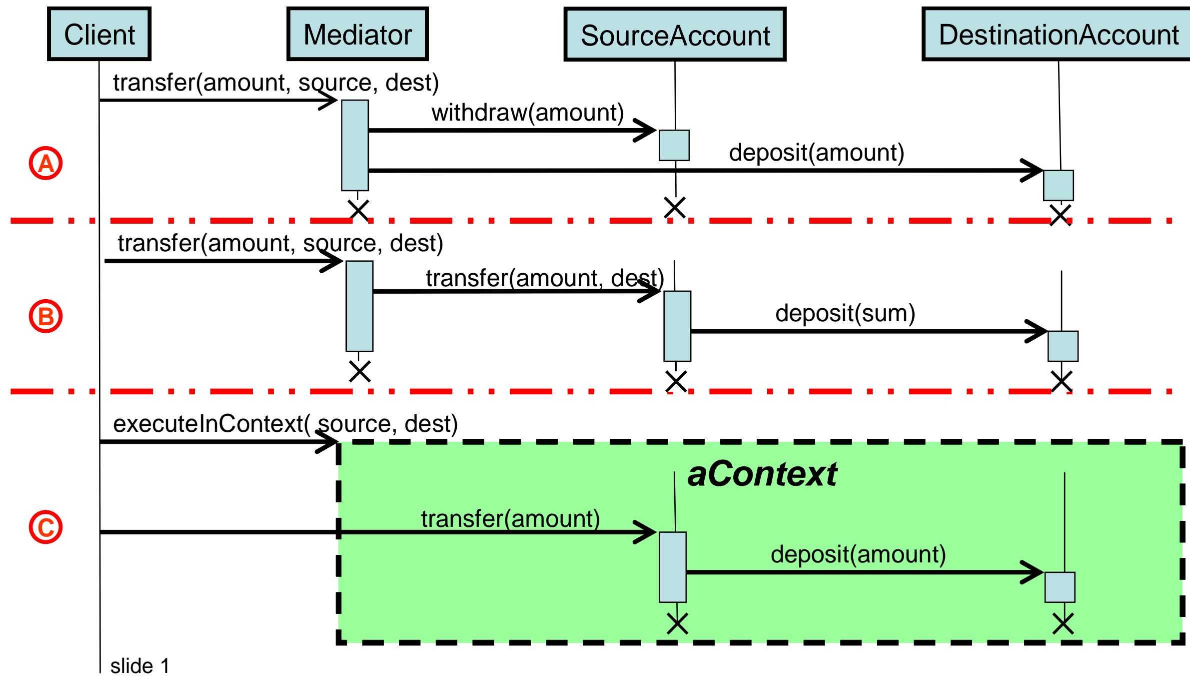
```
10. @source_account.transferTo amount, @destination_account
```

A more general solution could be to pass the *Mediator* object itself, the *source_account* object could then get the destination_account object by asking the *Mediator* object for it.

```
11. @source_account.transferTo amount, @self (or something like that)
```

In both figure 1(A) and (B), there is a single object that knows the roles. (The object is called *Mediator* in the figure because I have prefer to reserve the term *Context* for dynamic variables as discussed below.)

Figure 1: UML Sequence Diagrams showing role binding alternatives.



In figure 1(C), knowledge about roles is in a *Context* object. This object is known to all methods that directly or indirectly participate in the system operation (use case). The advantage of distributed logic is that participating objects can do “the right thing” according to their nature.

> But this means polymorphism! Haven’t I outlawed polymorphism at the interaction level? The answer is yes, but I am coming to believe that this answer is unnecessarily restrictive. I believe we can allow some polymorphism. The code will still be reasonably readable because we know the sender and receiver of every message through their role names and the binding algorithms.

1. 2 Finding the context

It is seems unacceptably clumsy to pass the *context* object as a parameter all over the place. It would be better if we could get at it through a general mechanism distinct from the messages. Some candidates:

Context in a global variable (e.g., #Context):
: We can only have one active at the time. This is clearly unacceptable.

Context in a named global variable (e.g., TransferMoneyContext)
: Blocks reentrant contexts, unacceptable.

Context in a global stack
: A context is put on a global stack whenever the context class is instantiated. Permits reentrant code. Pushes the context on this stack at the start of a system operation; pops it when the operation is completed. Take care to catch every completion including exception returns. (May be easier said than done?)

Context on the regular execution stack
: Context is put on the execution stack when starting a system operation. It will be available to all methods that are activated during the execution of the operation irrespective of actual objects or classes. The context object will automatically be popped from the stack at the completion of the operation. (This applies to sequential execution, multi-threading is for further study)

The last solution is chosen in BabyIDE[1]. The BabyIDE code for the SourceAccount in figure 1(C) is shown in code lines 4 through 6 above.

The BabyIDE implementation defines an abstract *BB1Context* class; there is a subclass for each system operation (use case). A *playerForRole:* static method in the *BB1Context* class knows how to find the objects currently bound to a role name.

So the full text of the *transfer:* -method in the class implementing *SourceAccount* is:

```
12. transfer: amount
13.     self withdraw: amount.
14.     (BB1Context playerForRole: #DestinationAccount) deposit amount.
```

The Squeak compiler and its friends have been extended to inline this lookup so that the code can use the role names directly. The result is shown in code lines 4 through 6 above.

1. 3 Role binding

The next problem to be solved is that we need to bind roles to objects at runtime. The solution is to have a Mediator/Context selection method for each role. In Steen's example, the Mediator factory method included the mapping in its initialization parameters:

```
15. class TransferMoneyContext
16.     attr_reader :source_account, :destination_account, :amount
17.     def initialize(amount, source_account_id, destination_account_id)
18.         @source_account = Account.find(source_account_id)
19.         @source_account.extend TransferMoneySource
20.         @destination_account = Account.find(destination_account_id)
21.         @amount = amount
22.     end
```

Another solution is to provide the context class with a mapping method for each role. This is the solution chosen in BabyIDE, but the main thing is that the context knows how to match a role name to the object that happens to play that role at runtime. The exact mechanism can be adapted to the actual requirements since each context is specified in a separate class.

2 Methodful Roles

The second DCI mechanism is methodful roles. It includes the capability to define role methods, i.e. methods that are associated with a role and are injected into all objects that play this role. (More commonly: the classes that specify these objects).

Schärli et.al. introduced the notion of traits in their 2003 ECOOP paper:[1]

> *"We then present traits, a simple compositional model for structuring object-oriented programs. A trait is essentially a group of pure methods that serves as a building block for classes and is a primitive unit of code reuse. In this model, classes are composed from a set of traits by specifying glue code that connects the traits together and accesses the necessary state."*

1.This solution is almost maximally inefficient as it is currently implemented, but there appears to be ample opportunities for optimization. This is left for further study.

1.Schärli, N; Ducasse, S; Nierstrasz, O; Black, A.; "*Traits: Composable Units of Behavior,*" Proc. ECOOP'03, LNCS, vol. 2743; Springer Verlag, July 2003, pp. 248—274. [DOI] 10.1007/b11832 [web page] http://www.iam.unibe.ch/~scg/Archive/Papers/Scha03aTraits.pdf

A methodful role can be created by associating a role with a trait. The trait methods are injected into all classes that implement the role. All objects playing the role will behave in the same way when they play this role.

Vanilla traits are stateless methods that are always executed in the context of an object. A trait method can access the state of this object through messages to *self*.

We have seen in section 1. 2 that the DCI Context shall be known to all methods that directly or indirectly participate in the transfer. This includes the role methods. Role methods can, therefore, send messages to roles as well as to *self*.

Many Traits features are not yet utilized in BabyIDE1. These features will be utilized if and when the need arises.

Early versions of BabyIDE did not include methodful roles. All system behavior was specified in terms of roles. The first published version assumes that all system behavior is specified by role methods. This solution seems to be equivalent to the centralized GOF Mediator. A possible future version of BabyIDE may be richer by allowing classes to override role methods. This is for further study.

3 Further work

The Trait based Squeak implementation of methodful roles is one of the possible solutions. Several good people have been working on interesting solutions in other languages: Jim Coplien (C++), Bill Venners (Scala), Serge Beaumont (Python), David Byers (Python), Steen Lehmann (Ruby), and Agata Przybyszewska (Java). (My sincere apologies if I have got it wrong.)

Work on dynamic variables for binding role names to objects at runtime does not appear to be as far advanced, but it is still early days.

There is an urgent need for more realistic examples; examples where a DCI solution is clearly superior to a traditional one. It will also be interesting to see counter-examples where the traditional solution is preferable. This can lead to guidelines for when to apply DCI and when to avoid it.

Realistic examples also require realistic tools. DCI/IDEs should be created or extended as the need arises.