A DCI Execution Model

Trygve Reenskaug

Dept. of Informatics
University of Oslo

Abstract

Computers provide three essential services: Data transformation, data storage, and data communication. Transformation and storage have long been well entrenched in algorithmic and data languages. Unaccountably, communication is only seen as an I/O operation without explicit language support. At long last, this deficiency has now been remedied with the new Data - Context - Interaction (DCI) paradigm.

Alan Kay introduced the notion of *object orientation* in the early seventies. He regarded an object as a virtual computer and an object system as many computers hooked together by a very fast network. The locus of DCI is in the *DCI Context* where we find the specification of object system behavior. The Context declares a network of communicating objects where the nodes are called *Roles* and the edges *Connectors*. The system behavior algorithm is composed from *RoleMethods*; methods that are associated with the Roles.

This article is about an execution model for DCI. An important aspect is the handling of DCI Contexts at runtime; where they are stored, how they are accessed, and how RoleMethods are executed. DCI in general and the DCI Context in particular make data communication a first class citizen of computer programming.

Document history
2010.08.22 - version 0: Draft for review
2010.09.16 - version 0.1 Second draft for review
2010.10.20 - version 1.0 First release.
2011.12.20 - Version 2.0. Updated with injection-less DCI.
2012.05.05: Version 2.1 released

1 Introduction

Note: Readers familiar with the DCI paradigm may jump directly to
Section 2 (p. 4): *The Plain Old Java Object (POJO) Execution Model*

Computers can essentially provide three services: *Data transformation*, *data storage*, and *data communication*. Algorithmic languages such as FORTRAN and Algol are data transformation languages. Data languages such as SQL and NIAM support data storage and retrieval.

Communication has long been a first class citizen in the hardware world; the bus centered IBM PC came in 1981. The software world has been slow in following suit. Communication has remained an I/O operation outside the scope of most programming languages.

Alan Kay, who coined the term *object orientation*, regarded an object system as a network of communicating computers:

> *In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole--like data structures, procedures, and functions which are the usual paraphernalia of programming languages--each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network.* [Kay-93]

The Smalltalk programming language is a class-based language where a class specifies one of Kay's virtual computers. A class specifies what an instance will do with an incoming message; it says nothing about the message sender or the enclosing network of communicating objects. The roots of this deficiency can be found in a faulty assumption: If a class is programmed to make its instances "do the right thing"; these instances will behave appropriately when they are combined into systems of communicating objects. This assumption fails when "the right thing" depends on context so that a class cannot be properly understood when seen in isolation. Another problem with class-based programming is that it ignores the well known axiom that "the value of a system is greater than the sum of its parts". The added value is caused by a particular organization of the parts into a coherent structure, the Context. The most important feature of an object system is communication, i.e., what happens in the space between its objects.

True object oriented systems can be described in some modeling languages such as OOram[OORAM-92] and UML[UML-11]. Since Kay's notion of an object is recursive; an object can be a member of an enclosing network as well as being the container of an inner network.

A class based programming languages cannot be used to specify a system of communicating objects since it can only describe one object (class) at the time. This fundamental deficiency is remedied in the *Data-Context-Interaction (DCI)* paradigm. Its two fundamental concepts complements each other. The *class* is a description of the insides of an object, its environment is invisible. The Context is a description of a network of communicating objects, the insides of these objects are invisible. DCI advances communication to be a first class citizen of programming.

Figure 1 illustrates how a simple example of a network of communicating objects accomplish a simple task. (Note that this is not an example of sensible programming, but an illustration of the DCI paradigm). Consider a person, Joe Smith, who wants to transfer some funds from one of his bank accounts to another. He goes to his bank and requests that it shall effectuate the transfer. Joe gives the transfer order to the bank, the bank instructs `Account 1234` to do the transfer, this account deducts the amount from its balance before it instructs `Account 5432` to deposit the amount. This way of looking at a computer application has two advantages: It is easy to understand for the bank customer and it can be a high level picture of the bank's computer program.

Figure 1: A concrete user's mental model of a bank transfer transaction.

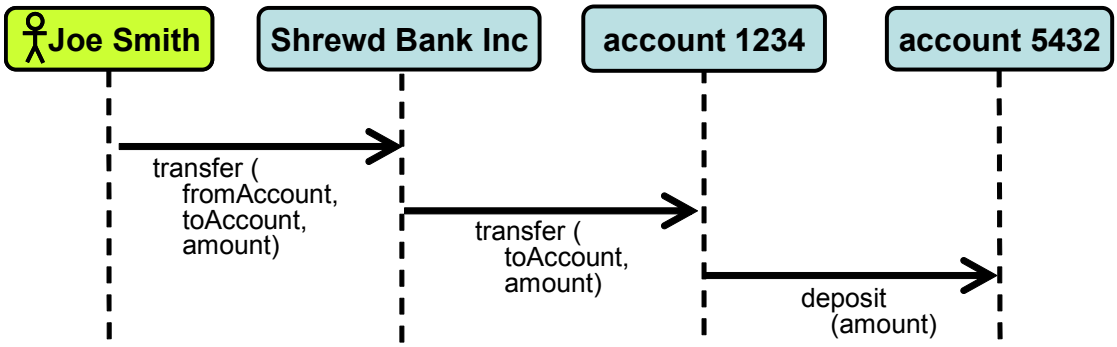
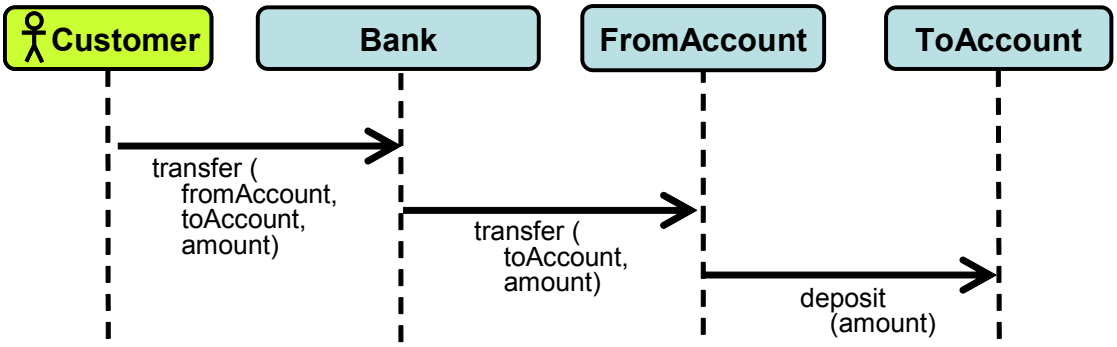


Figure 2 shows the corresponding messages sequence chart with its Roles and timelines.

Figure 2: A conceptual user's model of a bank transfer operation.



DCI can be studied from different viewpoints. When DCI is seen from the end user, we find use cases and user mental models. When DCI is seen from the application programmer's point of view, we see the DCI paradigm with its three perspectives of Data, Context, and Interaction. See for example the Wikipedia article *Data, Context, and Interaction*[DCI-Wiki] for a definition, *The DCI Architecture: A New Vision of Object-Oriented Programming* [ReeCop-09] for an overview, and *The Common Sense of Object Oriented Programming*[Ree-09] for details and commented examples.

We use the following definition

in this article:

SYSTEM

> A *system* is a part of the world *which we choose to regard as a whole*, separated from the rest of the world during some period of consideration, *a whole which we choose to consider as containing a collection of components*, each characterized by a selected set of associated data items and patterns, and by actions which may involve itself and other components.[Holbæk-Hanssen-75]

This article is written for the systems programmer who wants to create a DCI infrastructure. It is also written for the teacher of DCI to give him a deeper understanding of the subject.The purpose of the article is to gain an understanding of the realization of DCI in a computer. There are many possible ways of doing this, most of them require compromises caused by the nature of the available programming language and its runtime system. We here present a clean execution model that implementers can use as a point of reference.

2 The Plain Old Java Object (POJO) Execution Model

In POJO systems, memory is divided into two parts: A *heap* and a *stack*. The *heap* is a part of the memory that is reserved for objects. An object has behavior (methods) and represents part of the system state.

Another part of the memory is reserved for the *stack*. The stack is LIFO list of activation records with one activation record for each method activation.

2. 1 System state: What the system IS

When a class is instantiated, the new object is placed in an available part of the heap. The block of memory that represents the object includes a slot for each instance variable. Every object has a unique and immutable identity. The memory management system maintains a dictionary that binds object IDs to the location of the object on the heap. (Conceptually, that is. There are many different implementations.) An object stays on the heap until it is either deleted programatically or, more common these days, when it is no longer accessible and is removed automatically by a garbage collector. The memory management system is part of the runtime system, often called the *virtual machine (VM)*.

2. 2 System behavior: What the system DOES

Every object has a pointer to the object's class. The class is a container for all methods that can be executed by the object. (Conceptually, the methods could be stored together with the rest of the object on the stack. This would be extremely inefficient since all instances of a class share the same methods. The methods are kept in the class which acts as a shared memory. It is important to note that a method is executed in the context of an object; the class is a mere repository.

System behavior is controlled[1] through a stack of activation records. The stack acts both as a computation stack and a call stack. The activation record for a procedure call typically includes local variables, actual parameters. and the return address. POJO activation records also include a link to the current object and a link to the current method and a program counter within this method. A return address is not needed since the activation record contains all data needed to resume the method execution from the point where it was interrupted.

Individual object behavior is triggered when an object receives a message. The currently running method is interrupted and a new activation record is pushed onto the stack:

1) A method in the sender object creates a message. This message includes the sender object ID, the receiver object ID, the message name, and possible message arguments.

2) The execution of the calling method is suspended

3) The VM adds an activation record on the top of the stack as illustrated in figure 3. (Control data are not shown here)

1.Controlled, because the POJO activation record contains the data needed by the Control Unit in a von Neumann architecture.[Wikipeadia] (See figure 7 (p. 12)

4) The VM follows the class pointer in the receiver object to find its message dictionary. Using the message name as a key, it looks up the corresponding method in the receiving object. The lookup is repeated along the superclass chain if necessary. (A `doesNotUnderstand`-exception is raised on failure).

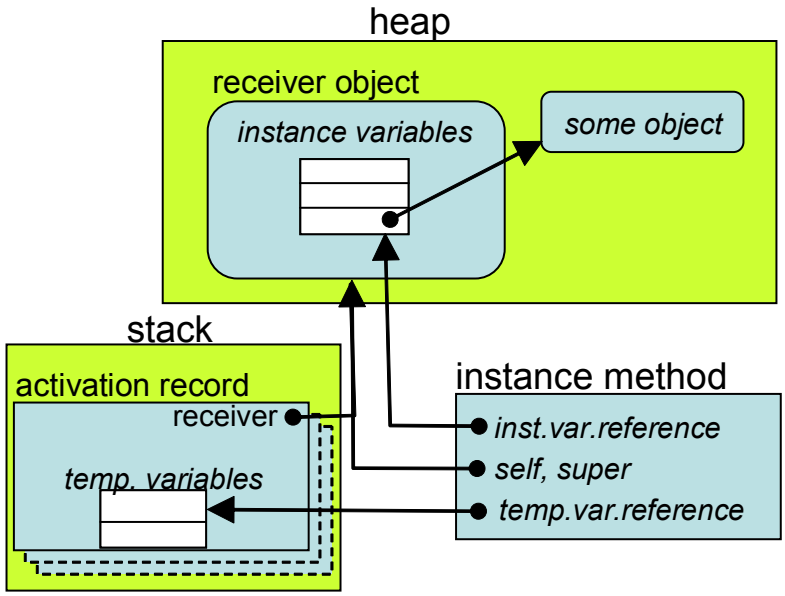
5) The activation record is supplemented with a link to the selected method and a program counter for this method.

6) The execution of the selected method is triggered and is performed in the context of the receiving object. A method may change the object's state. It can also send messages and the process is repeated from point *1)* above.

7) The activation record is removed from the stack upon method completion. The execution of the calling method is resumed from where it was suspended.

Figure 3 illustrates the runtime memory location of variables that are visible from a binary POJO instance method during its execution. There is an activation record for the current method. The stack grows and shrinks on top of this record as the method is executed. The activation record includes a link to the receiver object and slots for the method's temporary variables (actual arguments and local variables).

Figure 3: Runtime POJO memory link structure.



The namespace for the method source code includes identifiers for the temporary variables, the instance variables of the receiver object, and other variables[1]. A compiler transforms the source code into a binary method as a sequence of instructions (*byte codes*) to the VM. Some instructions move values between the top of the stack and various memory locations, some request message sends. and some do other operations.

3 The DCI Execution Model

D - Data Data objects are defined by Data classes and represent system state.

C - Context "A Context describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality." [UML-09]. It answers the questions: "What are the objects? How are they interconnected?"

I - Interaction The algorithms that specify how objects that are identified by the Roles they play, collectively accomplish some desired functionality (e.g., a

1.Static, global and other special variables are also visible; they are not discussed in this article.

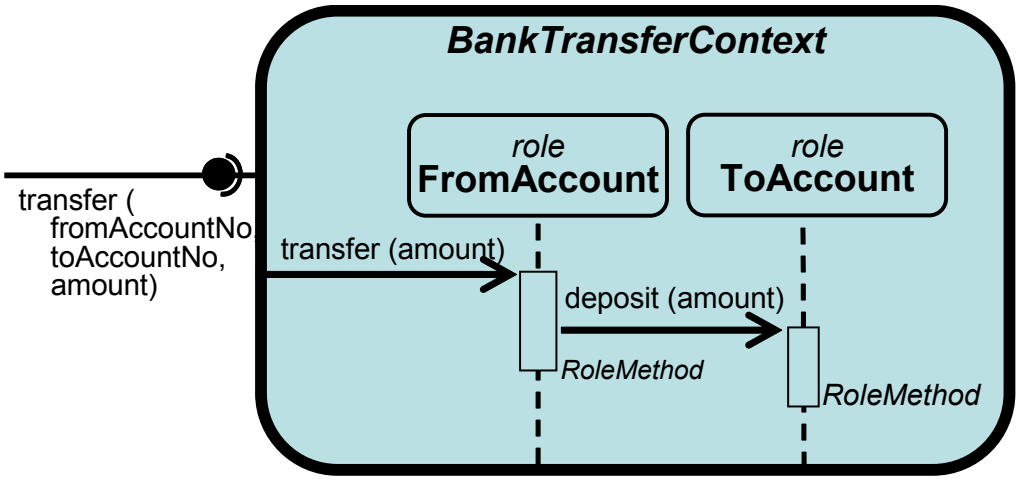
system operation or a use case). It answers the question: “How does the system work?”

The DCI execution model builds on the POJO model. The handling of system state is as described in section 2. 1 above. The system behavior is as described in section 2. 2 with the addition of a new kind of variable; the Role. We will see how this gives rise to an extended runtime model.

3. 1 An Application programmer’s view of DCI

For the purposes of this article, the programmer has decided to let the system shown in figure 2 consist of two interconnected Roles: A `FromAccount` and a `ToAccount`. He has also decides to let the system as a whole be represented by a `BankTransferContext`; the actual transfer takes place within this Context. The user’s conceptual mental model in figure 2 is now refined to become the application programmer’s mental model in figure 4.

Figure 4: Application programmer’s DCI based mental model of the Interaction.



1) The chosen system is the `BankTransferContext`.

2) An object in the system’s environment provides a trigger message; this message is called a *system operation* (or *use case*, or *habit* depending on the required precision).

3) The system’s *Data* objects (the accounts) and possibly other objects are represented by the Roles they play in this Context: `FromAccount`, `ToAccount`. (The Context binds these Roles to the appropriate Data objects by looking up the account numbers to find the account objects.)

4) The vertical timeline below each Role symbol is augmented with a rectangle that represents a *RoleMethod* that will be executed at this point. A RoleMethod is common to all objects that play the Role. (Polymorphism is suspended for RoleMethods). Taken together, the RoleMethods specify the *Interaction* that is executed when the system performs the system operation.

5) The RolePlaying objects for this use case are the bank account objects.

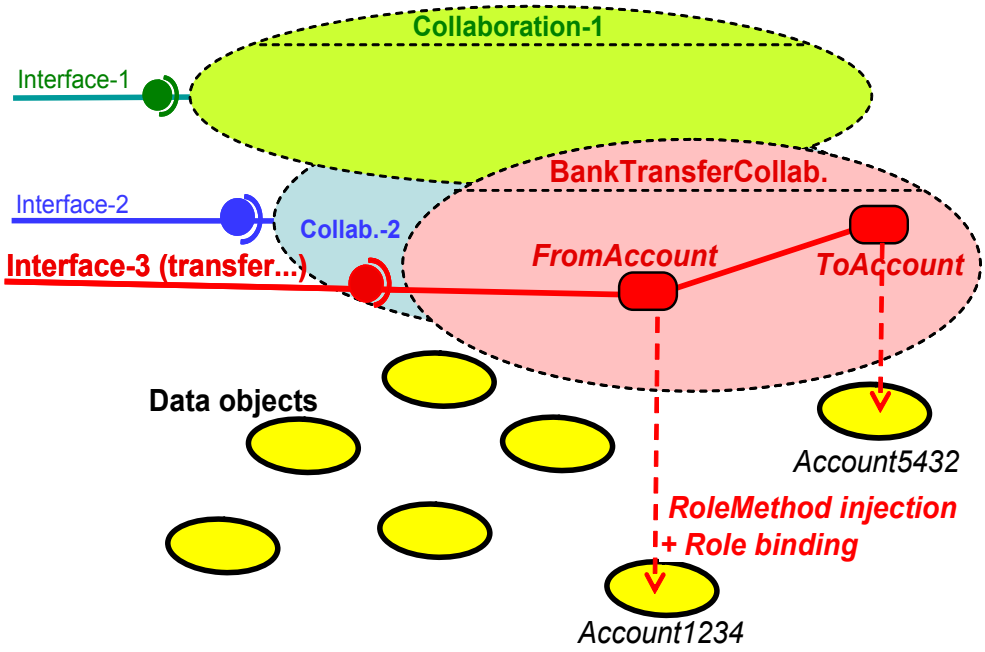
In one of his many clarifying posts on the object-composition@googlegroups.com list, Rickard Öberg wrote:

> *with DCI, the application facade that used to look something like this:*
> *facade.method(id1,id2,id3,param1,param2)*
> *is now replaced by:*
> *context<id1,id2,id3>.interaction(param1,param2)*

A bank offers many different services to its clients, the transfer of funds being one of them. Figure 5 shows how different services are realized by different Contexts. It is only when we look inside a Context that we see that it encapsulates a network of communicating Roles. We have opened the `BankTransferContext` and see that its Roles are momentarily bound to a selection of the bank’s Data objects. We see, for example, that the `FromAccount` Role happens to

be bound to the `Account1234` Data object. Similarly, the `ToAccount` role is played by the `Account5432` object.

Figure 5: DCI separation of concerns: System state and the different system behaviors.



3. 2 A compile time view of DCI

The code for a RoleMethod uses Role names as identifiers. The class of the RolePlayer is unknown, so there are no visible instance variables. For example, the RoleMethod shown as a vertical rectangle under the `FromAccount` Role in figure 4 (p. 6) could be:

```
1. FromAccount>>transfer(amount) {
2.         if (self.balance < amount) self.error ("no funds");
3.         ToAccount.deposit(amount);
4.         }
```

In the first line, `self` refers to the current roleplayer object; the object playing the `FromAccount` Role.

In the second line, `ToAccount` is the receiver of the message `deposit(amount)`. The semantics is that a RoleMethod has a higher priority than an instance method. Here, the RoleMethod is selected for execution:

```
5. ToAccount>>deposit (amount){
6.     self increase (amount).
7. }
```

There is no further message to a role, and the Interaction terminates.

3. 3 A runtime view of DCI

A DCI Interaction is triggered by a message to a Context object. The execution of an Interaction is slightly different from the vanilla execution described in the list on page 4. The process is as follows:

1) A method in the sender object creates a message. This message includes the sender object ID, the receiver object ID, the message name, and possible message arguments.

2) The execution of the calling method is suspended

3) The VM adds an activation record on the top of the stack as illustrated in figure 3. (Control data are not shown here)

4) The VM follows the class pointer in the receiver object to find its message dictionary. Using the message name as a key, it looks up the corresponding method in the receiving object. The lookup is repeated along the superclass chain if necessary. (A `doesNotUnderstand`-exception is raised on failure).

   During the Interaction, messages to an object will be handled in a special way. The object will execute instance methods in the context of the object's class in the usual way. The object will execute RoleMethods in the context of the `CurrentContext`. The method lookup mechanism is as follows: The VM will first search the Role's role methods for a match. If one is found, the object executes this RoleMethod. If not found, the search continues in the object's class as described in bullet *4)* above.

5) The activation record is supplemented with a link to the selected method and a program counter for this method.

6) The execution of the selected method is triggered and is performed in the context of the receiving object. A method may change the object's state. It can also send messages and the process is repeated from point *1)* above.

   There is a special case if the receiver is a Context and the messages is a system operation. The application programmer must write the method for a system operation in a special way:

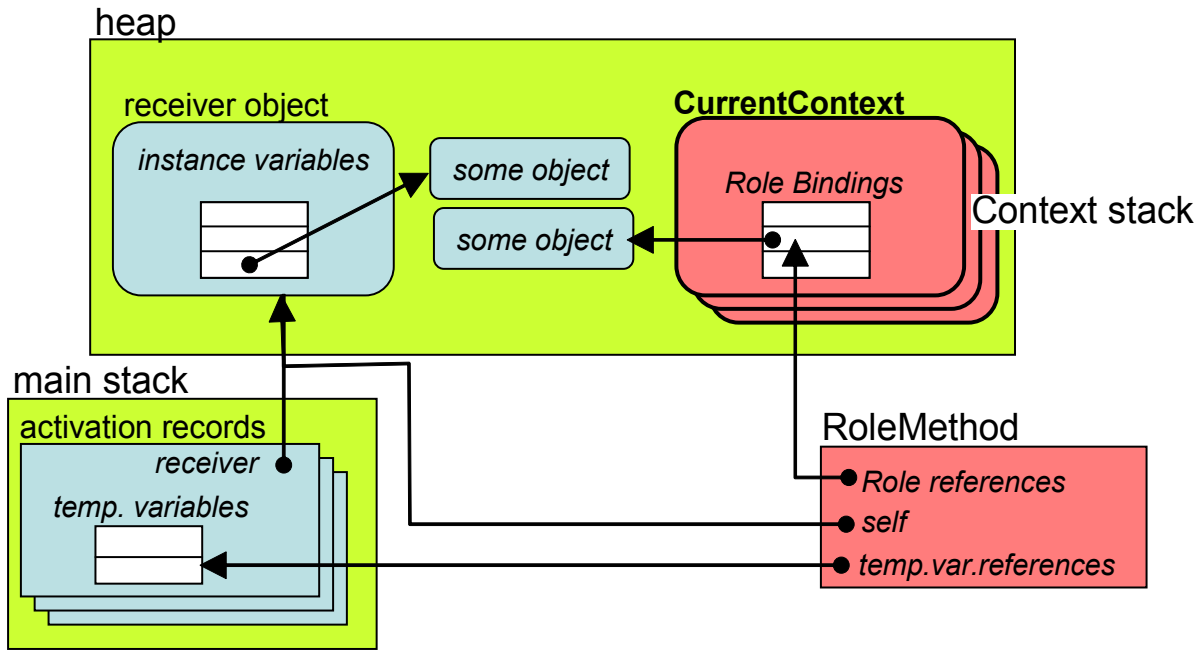
   a. The Context object must be initialized with objects needed for the role/object mapping.

   b. The system operation method is written by the application programmer. It must call a base method that pushes the Context onto the top of a globally available `ContextStack`. This receiver now becomes the `CurrentContext`. The base method then sends a `run`-message to a designated Role, thus starting an Interaction.

   c. At the termination of the Interaction, the `CurrentContext` is popped off the Context stack and execution of the system operation method continues in the normal manner.

7) The activation record is removed from the stack upon method completion. The execution of the calling method is resumed from where it was suspended.

Figure 6: Runtime DCI memory link structure of a binary method.



Instance methods are executed by a receiver object in the context of a class as illustrated in figure 3.

RoleMethods are executed by a receiver object in the context of a Context as illustrated in figure 6 A Role is not associated with a particular class so RoleMethods cannot access the instance variable in the receiver object. Instead, RoleMethods access the visible Roles.

Data objects indirectly through the Role name as shown in figure 6. This figure also illustrates that RoleMethods access temporary variables in the same way as instance methods. Note that `self` refers to the current receiver object; i.e., the object that is playing the current Role.

An important detail is that any object, including a Context object, can play a Role as long as it has the necessary properties. This means that a new Context can be triggered within an executing Interaction.

4 Three Constraints

With DCI, a program is decomposed into independent units of code that reflect important concerns that exist right from the end user's mental model to the innermost parts of the program. It is an imperative goal of DCI that it shall be possible to reason about the code in a particular unit with a minimum of interference from other units. This requirement leads to three constraints that are added to the DCI execution model.

4. 1 The coherent selection of roleplaying objects

One of the great powers of "object orientation" is that polymorphism permits variations on a theme defined by a base superclass. DCI blocks off this feature for programming system operations by ensuring that all objects playing a certain role will play its RoleMethods when required. Instead, the DCI Context is a variation on the same theme by selecting coherent sets of objects to play its Roles. The selection has to be done as an atomic operation in order to ensure that the bound objects represent the Context as a whole. (Independent role binding would endanger the integrity of the Context).

All Roles in a Context
are bound to objects in a single, atomic operation.

A Role can, at most, be bound to a single object at a given point in time. The above constraint says that the binding can only be changed by the atomic binding operation that binds all Roles at the same time.

4. 2 The uniqueness of the CurrentContext

Figure 6 illustrates a Context stack with more than one Context.

Only one DCI Context can be active at a time.

4. 3 DCI only supports single thread execution

May be this constraint will be relaxed in a future version of DCI.

5 Conclusion

The DCI execution model extends the POJO model with a capability for controlling the communication between named objects within a network of communicating objects. With DCI, we see the basic capability of computing in a three-layered architecture:

1) *Data storage* is done in an object's state variables. An object's apparent state can be derived, i.e., computed from other state.
2) *Data transformation* is done in an object's *methods*.

3) *Data communication* is done by message interaction in the context of a network of connected objects.as identified by their roles.

DCI adds data communication
as the third basic capability of computing,
the other two being data transformation and data storage.

6 References

[DCI-Wiki]	*Data, Context, and Interaction*; http://en.wikipedia.org/wiki/Data,_Context,_and_Interaction
[Holbæk-Hanssen-75]	Holbæk-Hanssen, E., Håndlykken, P., Nygaard, K.: "System Description and the DELTA Language". DELTA report No. 4. Second printing. Norwegian Computing Center, 1975.
[IFIP-66]	*IFIP-ICC Vocabulary of Information Processing*; North-Holland, Amsterdam, Holland. 1966; p. A1-A6.
[Kay-93]	Alan Kay:The Early History of Smalltalk; ACM SIGPLAN Notices archive; 28, **3** (March 1993);pp 69 - 95
[OORAM-92]	Trygve Reenskaug: *Working with objects. The OOram Software Engineering Method.* Manning/Prentice Hall 1996. ISBN 0-13-452930-8. Out of print. Late draft may be downloaded here .PDF *also* Trygve Reenskaug et.al.: *OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems.* JOOP 27-41 (October 1992)
[Reed-05]	David Reed: Organization of Programming Languages. Creighton University, 2005
[ReeCop-09]	Reenskaug, T., Coplien J.; The DCI Architecture: A New Vision of Object-Oriented Programming. An article starting a new blog: (14pp) http://www.artima.com/articles/dci_vision.html
[Ree-09]	Reenskaug T.; . [WEB PAGE] http://heim.ifi.uio.no/~trygver/2009/commonsense.pdf
[UML-11]	*OMG Unified Modeling LanguageTM (OMG UML), Superstructure. Version 2.4.1*; Object Management Group; Aug.2011; http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF
[Webster]	http://www.merriam-webster.com
[Wikipeadia]	http://en.wikipedia.org/wiki/ The work is released under CC-BY-SA . See http://creativecommons.org/licenses/by-sa/3.0/