The Roots of DCI

Trygve Reenskaug
Dept. of Informatics, University of Oslo

On 2010.07.05 13:46, Mircea wrote on the object-composition list:

> *I was wondering if you could elaborate on other works that make up the basis of DCI? Were there other works or just real-world observation and work done in BabyIDE?*

The short answer is that the basis of DCI is mainly based on real-world observation of end user requirements. The main DCI concepts are outside the scope of mainstream computer science and this science has, therefore, had almost no influence on the evolution of DCI.

The long answer is very long. But Mircea and others have asked for it, so here it is. The roots of DCI go back to the sixties. It started with my work with shipbuilding and ship design through the sixties that revealed the need for distributed computer systems. See section 1: *User need for distributed components* on page 2.

It seemed very promising to realize the distributed components with object systems, and objects have been at the front of my attention since 1970. There were quite a few stumbling blocks, however, the history is in section 2: *Object Orientation is a promising technology* on page 4.

I retired in 1997, and decided to spend my new freedom to pursue an old hobby horse. GOJO programs are not readable. I believed I knew how to program algorithms and declare information models. My problems stemmed from the third aspect of programming; that of specifying communication. See section 3: *My Retirement Project: BabyUML and DCI* on page 7.

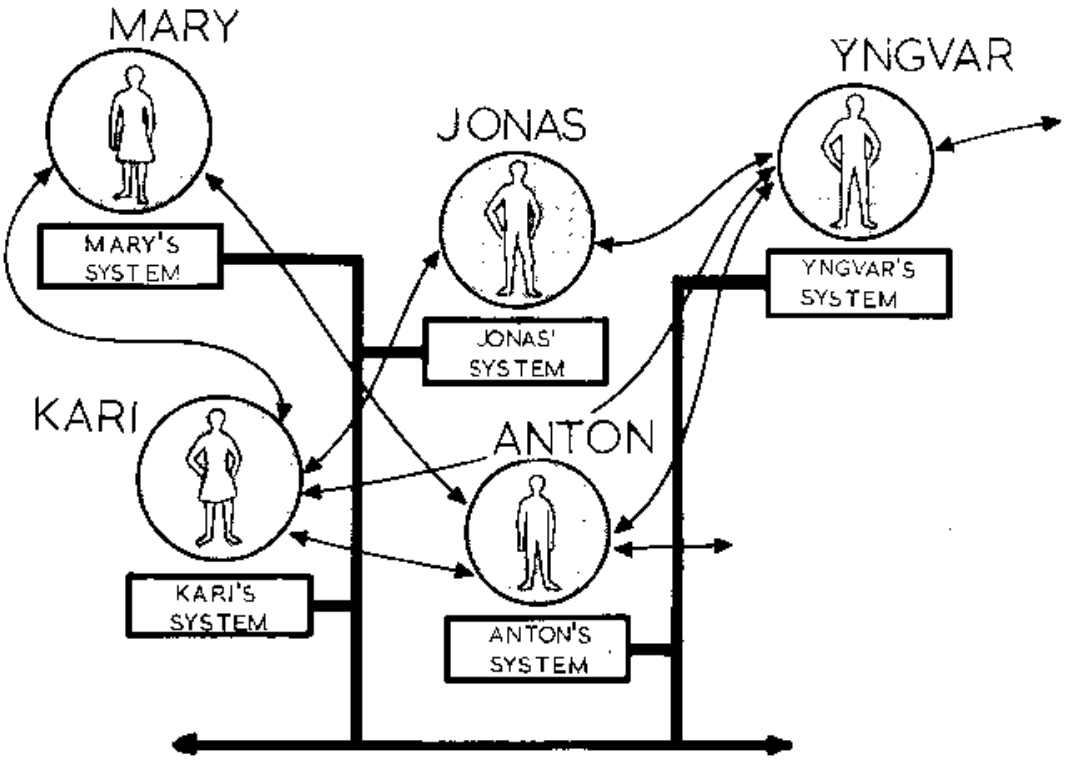
The last section is about the current state: section 4: *DCI Today* on page 8. Read it first if you do not want to read it all.

1 User need for distributed components

Throughout the sixties, I worked on a CAD/CAM system for ships. The system was called Autokon. It was first used in production in 1963, and it was subsequently adopted by most of the world's leading shipyards. The system architecture consisted of a central database holding an evolving product model, this was surrounded by applications that supported the design departments in their various tasks. The lofting department also used a special application to extract product data from the database and punch paper tapes for running numerically controlled machine tools; initially a flame cutter.

An important insight was caused by a blunder. Steel design used Autokon to produce the numerical equivalent of the old 1:50 drawings. A bright lad in Lofting recognized that the product model was now precisely represented in the database. Presto! Control tapes for the flame cutters could be pulled straight out of the database. They cut more than 300 tons of steel before discovering the difference between precision and accuracy. Yes, the data in the database had 40 bit precision. No, dimensions were still as approximate as they had been in the 1:50 drawings. 300 tons of scrap steel and some angry finger pointing was the result.

Figure 1: COMMUNICATION WILL BE THROUGH THE TRADITIONAL CHANNELS AND THROUGH THE NEW COMPUTER SYSTEM



We were blamed, of course, since we had developed the computer system. And I believed we deserved the blame. Previously, each department owned their own design data. Data transfer between departments was strictly controlled; the sender checked and signed before posting, the receiver checked carefully before basing any work on the received material. Our database broke this pattern. There was no data owner and there was no checking. The problem could, and was, of course fixed by changing the work process.

I thought the problem was deeper. The computer system should mirror the line organization and each line department should remain owner of its own data and its own programs. Figure 1 illustrates the architecture[1]. The departments are symbolized by their manager. Traditional communication (speech, paper) is along squiggly lines. Each manager has his own computer (could be virtual). The computer contains the manager's own data(base). The straight lines are communication lines connecting the personal computers called components in the communication network. The idea is recursive. A company can have many departments, a department can have many groups, etc.

I envisioned a matrix architecture where business processes are decomposed into component; each component being owned by the responsible department manager as illustrated in figure 2. (Taken from the same paper as the previous figure).

1. *Figure copied from ADMINISTRATIVE CONTROL IN THE SHIPYARD, ICCAS conference, Tokyo, 1973.* http://heim.ifi.uio.no/~trygver/1973/iccas/1973-08-ICCAS.pdf

Figure 2: TWO DECOMPOSED FUNCTIONS. EACH COMPONENT IS CONTROLLED BY THE RESPONSIBLE DEPARTMENT.

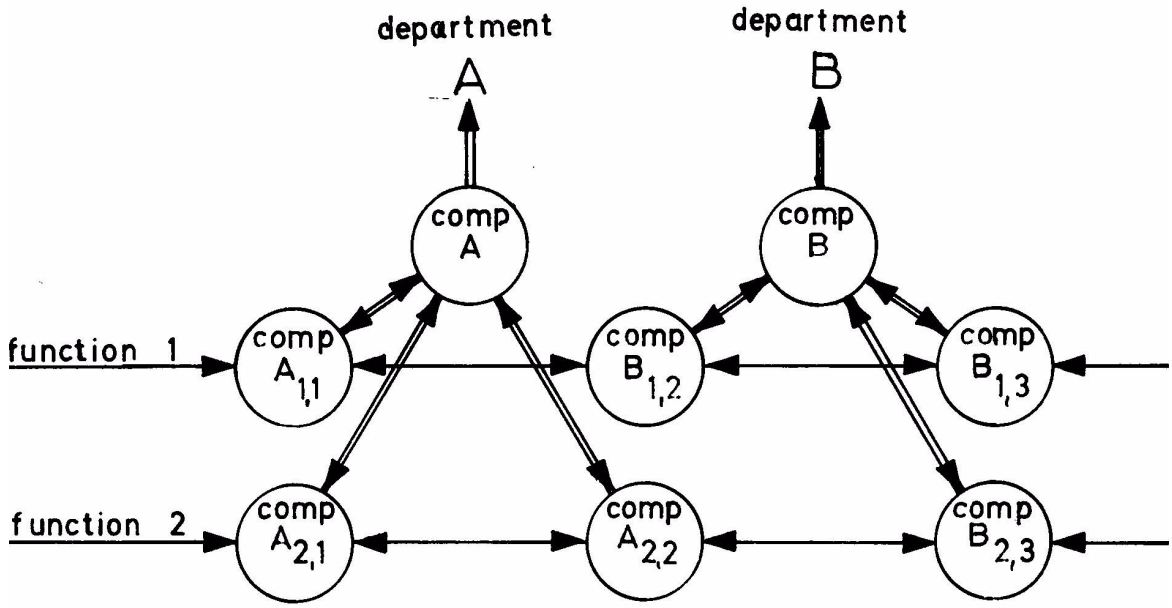


Figure 3 [1]illustrates a first attempt at documenting a business communication process where each component is one of the man-machine combinations shown in figure 1 and figure 2.

These three figures illustrate my mindset in the early seventies. I believed that authority and responsibility ought to go together in a business organization. In my role as a toolmaker, I also believed distributed organizations should be supported by distributed computer systems. Indeed, the need for such systems was so evident that I was convinced they would be commonplace by the end of the seventies. I was wrong.

2 Object Orientation is a promising technology

Around 1970, I got funding for creating a distributed planning and control system based on these ideas. Simula had been invented by Nygaard and Dahl at the Norwegian Computing Center across the yard from my office. What if the components were implemented as objects in a huge Simula program? Great idea, but we hit obstacles. One was that the execution of a Simula program typically lasted seconds or minutes. We needed the execution to last for a year or more. (Persistent objects were still waiting to be invented.) A second obstacle was the real show stopper. Each manager should own his own programs, and Simula insisted that the sender of a message must know the class of the receiver! This broke with the whole idea of communicating components where the communication pattern was standardized while the inside of each component was private. (We even expected that some components would be manual). Exit Simula, and we had a problem.

At that time, the market for new ships collapsed and our project funding disappeared overnight. In retrospect I see that we had a great deal to learn and the possible outcome of the project is not at all obvious. With no project, I got time on my hands and used it to document some of our ideas. The two papers referenced above were written at this time.

---

1. *Figure copied from "PROKON/PLAN-A MODELLING TOOL FOR PROJECT PLANNING AND CONTROL": IFIP Congress, Toronto, Canada, 1977;* http://heim.ifi.uio.no/~trygver/1973/iccas/1973-08-ICCAS.pdf

Figure 3: This example of a resource loading algorithm illustrates the informal, movie-type of documentation used to describe the organized flow of messages resulting from a given user command. .

The Project Component (PC) selects the first resource to be loaded. (Resources are here loaded sequentially. Necessary backtracking is supposed to be done through manual intervention.) When all resources have been loaded, message 148 to the user indicates that the command is completed.

The Resource Requirements Component (RRC) asks the Component of the departmental head (DHC) to give the current capacity limits through message 150.

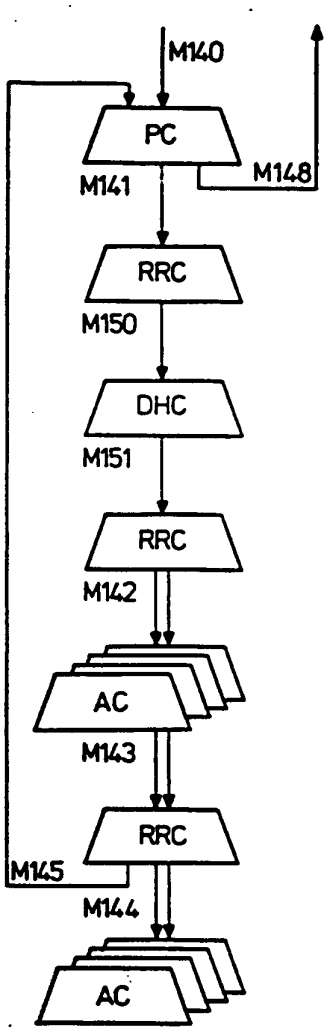
DHC returns the capacity it thinks fit to release for our project through message 151.

RRC then asks all ACs for their resource requirements and their earliest and latest start and finish.

The ACs return the requested information.

RRC performs loading according to its own, built-in rules and reports the planned times for start and finish to the ACs. It also reports back to PC that it has completed its loading.

The ACs note their new planned start- and finish times.



The most important result of the aborted project was that I became a visiting scientist in the Smalltalk group at Xerox PARC in 1978/79. There was, of course, immediate resonance. My ideas of communicating components fitted well with Smalltalk's communicating objects. My focusing on end users and their ownership of their own programs fitted well with Alan Kay's vision of the Dynabook and also with Douglas Engelbart's vision of the computer as an extension of the human brain (Computer augmentation).

In the Smalltalk group, we used to say that there were two approaches to object orientation. One was the East Coast approach with C++ and such: “Object orientation is a smart programming artifact where an object is an instance of a class and also a data structure (struct? record?) with built-in methods for operating upon its data”. The other was the West Coast approach: “Object orientation is much more than that. Object orientation is an entirely new paradigm for thinking about systems. An object is an element for building corresponding models in the human head and in the computer. The essence of object orientation is that objects interact to achieve a task.” The main difference between these views is clearly the difference between machine-orientation and people-orientation. Our forefathers long ago had slaves working for them. I had thousands slaves working for me in the shape of objects in my computer and they were doing exactly what I had told them to do. (Which couldn't always be said of then human slaves).

Alan Kay later wrote an article, "*The Early History of Smalltalk*", where he defined object orientation as follows. (The whole article is well worth reading if you are interested in the foundations of programming[1]).

> *"Smalltalk's design--and existence--is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages. Philosophically, Smalltalk's objects have much in common with the monads of Leibniz and the notions of 20th century physics and biology. Its way of making objects is quite Platonic in that some of them act as idealisations of concepts--Ideas--from which manifestations can be created. That the Ideas are themselves manifestations (of the Idea-Idea) and that the Idea-Idea is a-kind-of Manifestation-Idea--which is a-kind-of itself, so that the system is completely self-describing-- would have been appreciated by Plato as an extremely practical joke [Plato].*
>
> *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 computer all hooked together by a very fast network. Questions of concrete representation can thus be postponed almost indefinitely because we are mainly concerned that the computers behave appropriately, and are interested in particular strategies only if the results are off or come back too slowly.*
>
> *Though it has noble ancestors indeed, Smalltalk's contribution is a new design paradigm--which I called object-oriented--for attacking large problems of the professional programmer, and making small ones possible for the novice user. Object-oriented design is a successful attempt to qualitatively improve the efficiency of modeling the ever more complex dynamic systems and user relationships made possible by the silicon explosion."*

I agree with Alan Kay's definition of object orientation, but there were two questionable assumptions underlying the Smalltalk implementation. One assumption was that if an object was coded "to do the right thing", it would behave properly when interacting with other objects. This is OK in simple cases, but it fails for systems where "the value of the system is greater than the sum of its parts". In such cases, what is "the right thing" depends on the system as a whole, i.e., the context.

There was also an assumption that the class was the ideal unit for code reuse. But classes are often designed together in ensembles and are useless outside their ensemble. An example is what used to be called a framework; an ensemble of classes that complement each other and that are designed to be subclassed together to create a specific application

In short: Classes are not independent, and a higher level construct is needed to specify the whole.

The Model-View-Controller[2] was my first departure from the Smalltalk assumptions and it followed naturally from my object-oriented mental model. It seemed so obvious that I declined an invitation to write an article about it in the special issue of the Byte magazine dedicated to the launching of Smalltalk-80. Instead, I wrote an article called "User-Oriented Descriptions of Smalltalk Systems". The article focuses on the "owner" of the system and his mental model.

---

1. *ACM SIGPLAN Notices; Volume 28, Issue 3 (March 1993) There are many copies on the web, such as* http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html
2. http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf

Structures of interconnected objects, of course, and interacting objects. This article builds on the above references and is clearly one of the roots of DCI.[1]

I returned to the Central Institute for Industrial Research in Oslo in 1979. Our first activities were to build a local Ethernet, to port Smalltalk-76 to a Norwegian computer. and to get funding for a project that aimed at helping Norwegian industry with the wonderful new technology from Xerox.

Role modeling followed naturally; the first OOram tool was demonstrated at the first OOPSLA in Portland, Oregon in 1986.

3 My Retirement Project: BabyUML and DCI

There is a widespread belief that testing was the only way open for debugging OO programs. (Dijkstra: *Testing can show the presence of bugs, not their absence.*) I also like to quote the Gang Of Four:[2]

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

This horrible state of affairs has been with us for thirty years and nobody seem to take it as a challenge to do something about it. Every engineering specification/drawing I have ever seen has two signatures: 'done by' and 'checked by'. In shipbuilding, a representative of the insurers actually checks all important drawings. (Veritas and others). In programming, our weak technology prevents effective checking. Very frightening.

I took up the challenge when I retired i 1997. My retirement goal was to find a way to make code reveal everything about how a works. This clearly implied making object interaction explicit in the code. This should both help making the code reflect the end user's perception of the domain, "no surprises", and to enable code checking (e.g., peer review, a hobby horse of mine[3]). I first believed that a solution was to make UML into a programming language, this effort is covered in my BabyUML page: http://heim.ifi.uio.no/~trygver/themes/babyuml. This page includes articles, reports, and talks.

Three publications from the BabyUML project are cited here for historical reasons; their technology has been superseded by DCI and the BabyIDE programming environment.

A search for usable methods:

Reenskaug, T.: Why Programmers Don't Use Methods And What We Can Do About It.; A column in ObjectEXPERT January 1997; http://heim.ifi.uio.no/~trygver/1997/Why/970329why.pdf

An experiment where I explored the use of object components:

2006; Reenskaug, T: *The BabyUML discipline of programming (where a Program =data + Communication*

---

1. *User-Oriented Descriptions of Smalltalk Systems.Byte Magazine, August 1981* http://heim.ifi.uio.no/~trygver/1981/byte/userorienteddescriptions.pdf
2. *Gamma, E; Helm, R; Johonson, R; Vlissides, J: Design Patterns; ISBN 0-201-63361-; Addison-Wesley, Reading, MA. 1995; pp.22-23.*

3.*The Case for Readable Code*; Expert Commentary; in Klein(ed): *Computer Software Engineering Research*; pp. 3-8; Nova Science Publishers 2007; ISBN: 1-60021-774-5. http://heim.ifi.uio.no/~trygver/2007/readability.pdf

+Algorithms); Expert' voice; Software and Systems Modeling; **5**, 1 April 2006; DOI 10.1007/s10270-006-0005-0; .http://heim.ifi.uio.no/~trygver/2006/SoSyM/trygveDiscipline.pdf

An experiment where I tried to redefine the notion of a class to better support runtime components using metaprogramming (Changing the Smalltalk notion of a class):
Reenskaug, T.: *Programming with Roles and Classes: the BabyUML Approach*; Chapter in Klein: *Computer Software Engineering Research*. 2; pp. 45-88. Nova Science Publishers, New York, 2007; ISBN-13: 978-1-60021-774-6.; http://folk.uio.no/trygver/2007/babyUML.pdf

The goal of the BabyUML project was to bridge the chasm between the code we write at compile time and the networks of communicating objects that do the work at runtime. The BabyIDE interactive development environment bridges this chasm. Its foundation was the new DCI paradigm, one of its keystones was a variant of the Traits stateless methods.

The two last references above were based on the BabyUML component; an object that encapsulated a structure of inner objects and that was characterized by its provided interface. The BabyUML Components were used to organize the data objects in a static structure. A few experiments showed that this was exceedingly cumbersome. Different structures were needed for different system operations.

An important innovation was to replace the BabyUML static data structure with the dynamic DCI Context; a temporary object structure that is created for the purpose of an execution and that exists only during that execution. Further, the participating objects are referenced indirectly through the roles they play in the execution. Finally, Traits were added to specify the interaction explicitly, blocking the ambiguities caused by polymorphism.

4 DCI Today

On the 28 August 2008, I declared that the BabyUML project had reached its goal.

There was no trace of UML in the 2008 release, so the project was renamed BabyDCI. There are two reasons for the 'Baby' part of the name. The 2008 release was an infant that I hoped would grow into something viable and powerful. But also somewhat whimsically: The world's first electronic, digital, stored program computer was called 'The Baby' (University of Manchester, England, 1948). As we all know, much followed after this feeble beginning. The 2008 version of DCI may be the world's first programming paradigm where data communication is a first class citizen of programming, taking its rightful place together with data transformation and data storage. Something big might come out of it.

One of my goals for the BabyUML project is quoted in the "Common Sense" report[1]: "*My hope is that this first BabyIDE implementation shall inspire programmers, developers, and researchers to pick up the baton and run with it*". I am truly grateful for the many people who are making this wish come true.

After 2008, DCI has been brought out in the world under the leadership of James Coplien. Without him, DCI would have remained an obscure curio. The following was my first reaction to the *Lean Architecture* book he wrote with Gertrude Bjørnvig:

> *Where my "Common Sense" report is targeted at the coder, "Lean Software Architecture" paints on a much broader canvas: Working with the end user, end user's mental model, user requirements, system architecture, and right down to actual code. A MUST read.for all who want to understand the true nature of system development.*

---

1. http://heim.ifi.uio.no/~trygver/2009/commonsense.pdf

Thanks to Jim, there is today a significant body of gifted people who are studying DCI, adapting it to different languages, applying it to different requirements, and even developing a new programming language that supports DCI.

Jim has started a DCI mailing list called object-composition@googlegroups.com. The number of active participants on this list is growing, and there has been many threads that have given new insights.

Euclid is said to have replied to King Ptolemy's request for an easier way of learning mathematics that "there is no Royal Road to geometry". Likewise, there is no Royal Road to DCI. An object is an entity that has identity and that encapsulates state and behavior. The class is the common abstraction of objects. Wikipedia defines abstraction as follows: *"In computer science, the mechanism and practice of abstraction reduces and factors out details so that one can focus on a few concepts at a time"*. The *class* abstraction factors out object identity and focuses on the object's inner construction (attributes and methods). In DCI, the *Role* is an opposite abstraction that factors out the object's inner construction and focuses on the object's identity. It is this change of focus that makes it possible to identify the senders and receivers of messages and thus reason about how a network of communicating objects performs a task. And it is this change of focus that makes DCI hard to understand for a class oriented person because object identity is explicitly excluded from consideration. (Opinions about DCI based on class based thinking are weakly founded.

> *Try to explain the notion of color to a completely color blind person. Try to explain the notion of Roles to a class-centered person. The notions are out of scope in both cases.*

The object-composition list is somewhat marred by contributors who loudly demand a royal road to DCI. They are necessarily disappointed, but hopefully also spurred on to renewed efforts. There is no substitute for actual experiments for enlarging one's mental model.

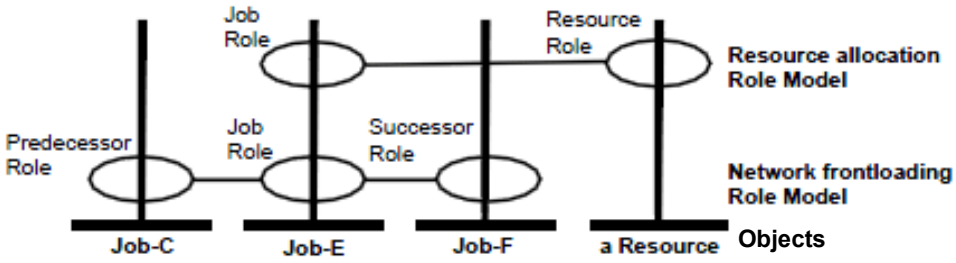
That said, the discussions on the object-composition list have inspired much of my work since its creation in 2008. I have rewritten my Smalltalk/Squeak examples several times, each rewrite exploring a different implementation.

There is an important outstanding issue that is bound to hit us hard at some point in the future. In DCI, there are overlapping namespaces with their attendant inconsistencies. A class definition is a namespace. Method names are unique within this namespace. Different classes can reuse the same method name without conflict. (The class hierarchy doesn't invalidate this argument, only makes it more complicated). A Context is another namespace. Role names are unique within a Context. Different Contexts can reuse the same role name without conflict.

The problem is with the role methods. Role methods belong in the Context namespace. But role methods are injected into Data objects or classes where their names may conflict with methods in the class proper and with other role methods injected from other Contexts. The usual way to handle such problems is with qualified method names such as `<Context name>/<Role name>/<method name>`, but it is not at all clear if this is a good solution for DCI.

An important part of OOram was *role model synthesis*; an operation that merges several models into a composite. Figure 4 illustrates that OOram synthesis constrains several objects to be played by the same object. A copy of figure 1.10 in *Working with objects*.[1]

Figure 4: OOram Synthesis specifies that objects play several roles in a coordinated manner.



In DCI, a corresponding Context merge would mean that roles from different Contexts will be constrained to be played by the same object. Role model synthesis was an essential part of OOram. It is not at all clear if a similar Context merge in DCI will be meaningful or useful. I've made one inconclusive experiment, but I clearly need to refresh my memory with Egil Andersen's PhD. thesis "*Conceptual Modeling of Objects. A Role Modeling Approach*"[2] where he uses state machines to describe the behavior of object systems.

I have long wanted to rewrite BabyIDE using DCI. But before that, there are a few articles that should be written...

My home page for DCI is http://heim.ifi.uio.no/~trygver/themes/babyide where I from time to time post results of my ongoing work.

DCI is evolving and this story is still unfolding, The natural narrator for this continued story will be Jim Coplien. The future looks bright for the DCI paradigm.

---

1. *Reenskaug, T. et.al.: Working with objects. The OOram Software Engineering Method. Manning/Prentice Hall 1996. ISBN 0-13-452930-8; Out of print, but is still available from some bookshops including Amazon as of July 2010. You can alternatively download the last draft before publication. It has not had the benefit of the copy editor's corrections and improvements, but its substance correspond closely to the printed book. .*
   http://heim.ifi.uio.no/~trygver/1995/95Article/951010-paper.pdf
2. http://heim.ifi.uio.no/~trygver/1997/EgilAndersen/ConceptualModelingOO.pdf