DCI as a New Foundation for Computer Programming

Trygve Reenskaug,	James O. Coplien
Dept. of Informatics,	Gertrud & Cope
University of Oslo	Copenhagen
Norway	Denmark
(“Trygve”)	(“Cope”)

ver.1.3 - Last modified FrameMaker version: November 23, 2013 2:48 pm
draft.1.4.2 - FrameMaker version: Updated and edited from “ver1.3.1.bvs.fdbk.docx”
draft.1.4.5 - Minor updates. (.fm + .pdf)
draft 1.5.1 - Incorporate comments from Risto
draft 1.6.1 - 24.01.07-‘*Data*’ names a kind of projection, not a set of objects. ‘*base* object’ has been renamed to ‘*root* object’.
draft 1.7 - 14.01.09 - Updated 'an object is … ' and minor changes.

Abstract

After more than 60 years with computers, hundreds of millions of people are dexterous at using them. Yet, the source code for a simple app is incomprehensible to almost all. We claim this is wasteful and passé -- Wasteful, because many valuable opportunities are lost; passé because computer programming is rapidly becoming an essential part of primary and secondary education.

We introduce a new paradigm for computer programming called DCI - Data, Context, Interaction. DCI brings programming to the level of everyday concepts and activities. The professional programmer can attack complex problems without undue additional complexity. The software maintainer can preserve system integrity by understanding and honoring the system architecture long after the originators have moved on to other projects. DCI can be embedded in different programming languages that are specialized for different purposes. The DCI concepts can become a unifying foundation for programming in school curricula.

DCI specifies a program as seen in two orthogonal *projections*; the *Data* projection describes system state and the *Context* projection system behavior. Recursion further separates a system into comprehensive levels, each level revealing more details than the level above.

Key Insights

- **A computer can augment the human intellect when the human mental model closely corresponds to the computer model defined by its program.**
- **There is ample evidence that the notion of objects is well matched to the human mind.**
- **An object-based model of a computation can be shared between the end user’s mental model and the design of the program. This gives the user leverage to maximize the value of the computer.**

1 Introduction

We introduce a new paradigm for the understanding and coding of computer programs that we call *DCI - Data, Context, Interaction*. In this article, we focus on professional users who apply computer systems to improve the performance of their tasks. Mental models that are grounded in their disciplines will drive their work. The goal is to create a program that feels like an extension of the user's mind. Users need to learn the rudiments of programming so that they can explore the program’s capabilities and suggest well-founded improvements. The lean principle “everybody, all together, all the time”[3] says that the user shall be an active member of the development team. There shall be no surprises.

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

This is not a trivial goal. Indeed, in the introduction to the Design Patterns book[8 pp. 22.23], the authors write: “it's clear that code won't reveal everything about how a system will work.” It is frightening to read that there are mission critical systems in use today where the code does not reveal how the system actually works. The end users are not alone in their illiteracy; even system maintainers and other experts have problems understanding what goes on in the computer. This problem challenges us to find a way to write code that clearly expresses the system’s runtime behavior. We need a new programming paradigm such as DCI.

The next wave of the digital revolution arrives next year with every English child being taught computer programming.[16] It starts with the 5-year-olds and continues at least until they turn 16. We propose that the conceptual framework of DCI can form a unifying foundation for the range of the concepts to be taught.

DCI is founded on a shared understanding of the nature of *representation* as clarified in the 1966 IFIP vocabulary of Information Processing[11]. It has three definitions that stand the test of time:

> “DATA. A representation of facts or ideas in a formalized manner capable of being communicated or manipulated by some process.”
>
> “INFORMATION. In automatic data processing the meaning that a human assigns to data by means of the known conventions used in its representation.”
>
> “DATA PROCESSING. The execution of a systematic sequence of operations performed upon data.”

The end user’s mental model is in the center of our attention in this article. A program is contemplated in two orthogonal projections. The DCI *Data* projection captures the *information* content of the mental model. The DCI *Context* projection captures its *data processing* properties. The mental model is reified in readable code[23]. By “readable”, we mean code that is cleanly partitioned and that clearly exhibits the designer's intent. More importantly, it means that we optimize the amount of code at hand so stakeholders can reason about the system at a granularity that suits them, without being burdened with extraneous artifacts and with a minimum of loose ends in the reader's mind.

The DCI paradigm is based on the concept of *objects* that was introduced by Ole Johan Dahl and Kristen Nygaard in the mid 1960s.[17] The notion has matured over the years, and in section 2: *Prior Art* we glean powerful concepts created over the past four decades. DCI builds on these concepts, and its main goals are:

> MENTAL MODELS. To reflect the way different users conceptualize the objects of their world so that a program that feels like an extension of its user's mind.
>
> REASONING. To help software developers reason about system state and behavior in addition to the state and behavior of isolated objects.
>
> READABILITY. To improve the readability of object-oriented code by giving system behavior first-class status.
>
> REUSE. To be able to reuse old solutions for new purposes.
>
> REVISION. To cleanly separate code for rapidly changing system behavior (what the system *does*) from code for slowly changing domain knowledge (what the system *is*), instead of combining both in one class hierarchy.

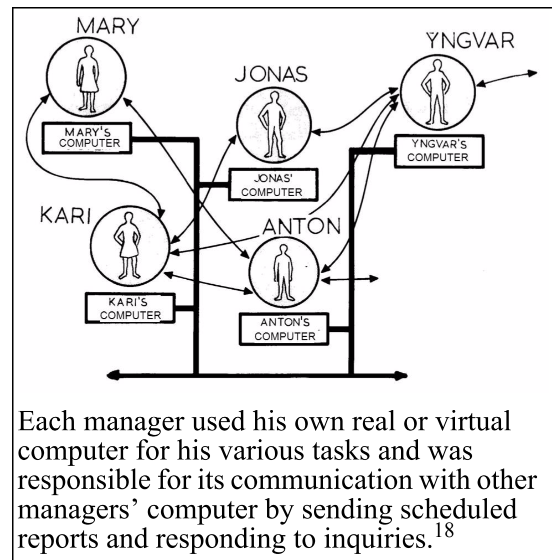
These goals are resolved with the DCI paradigm presented in section 3: *DCI, the new Programming Paradigm*. A simple example follows in section 4. In section 5: *Related Work*, we briefly comment on other efforts that are related to DCI. Suggestions for further work are in section 6. In section 7, we conclude with the vision of DCI as a programming paradigm that spans many programming and modeling languages as well as human users ranging from professionals in business and industry and other stakeholders, to game developers, composers, schoolchildren, and professional programmers.

2 Prior Art

Fundamentally, a computer offers three simple services: It can process data, it can store data, and it can communicate data----So simple, yet so powerful when combined in various ways into comprehensive programs. Trygve has for more than 40 years been working towards making this fundamental simplicity penetrate into the programs we write and use. DCI is the most recent result of this work, but it builds on what he has learned through the years. This section gives some highlights of his experiences that lead up to DCI. He has undoubtedly missed important developments over the years; some of them are mentioned in section 5.

2.1 Prokon's Distributed Systems

Figure 1: Prokon, a distributed system architecture



Each manager used his own real or virtual computer for his various tasks and was responsible for its communication with other managers' computer by sending scheduled reports and responding to inquiries.[18]

By 1970, it was clear that there was a fundamental mismatch between centralized database architecture and the decentralized nature of an organization's distribution of responsibility and authority.[18] The Prokon project proposed a distributed system architecture (figure 1) to restore the balance between an organization's distribution of responsibility and the system architecture.

Four of the major requirements for Prokon were:

1) Managers should be autonomous within their fields of responsibility and own the computer that served them. They were free to decide how to do their work as long as they fulfilled their responsibilities.

2) Business information was distributed between the individual subsystems. This lead to a three-level architecture with a data store at the database level, data processing at the application level, and a communication level at the top that integrates the different subsystems. *Communication became a first class citizen of system architecture.*

3) Managers should understand how their computers work and should oversee its creation. They should preferably be able to write part of the code themselves.

4) The individual subsystems were bound together by overall algorithms that ensured overall completeness and consistency. ("Local independence combined with central control.")[19]

The project was never completed, but it had many ideas that point towards DCI. First, the focus on the end user as the defining entity for both architecture and system details. Second, the importance of the communication bus that connects the autonomous systems that encapsulate state and behavior. Third, the need for algorithmic control over the communication as a whole.

2.2 The First Object

Nygaard and Dahl's concept of *objects* was realized in the programming language Simula 67[17]. The language introduced object modeling as a new and powerful way of thinking about complex systems. Originally designed to simulate real-world phenomena, Simula has also enjoyed application as a general-purpose programming language. The construction of a Simula object is given by a *class declaration* that includes a name, a data structure declaration, and the action pattern of each object of the class. (We shall later use the terms *attributes* for the data structure and *methods* for the action pattern). The Simula experience indicated that complex physical systems could be naturally reflected in object-based mental and computerized models. The Simula objects marked an important step towards the DCI paradigm.

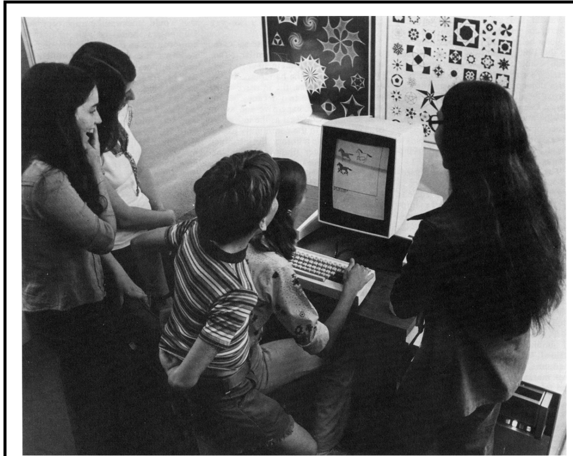
 ©2014 Trygve M. H. Reenskaug and James O. Coplien

2.3 Kay's Object Orientation

From the late sixties, Alan Kay had worked on his vision of a Dynabook: "A personal computer for children of all ages."[12] As part of his work, he introduced a powerful object model that he called *object orientation*:

> "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..."[13]

Figure 2: The Smalltalk experiment.



Kay's group at PARC taught programming to children. The children see each object as a 'turtle' with a pen under its belly.

```
turtle  go: 100; turn: 90; go: 100; turn: 90;
        go: 100; turn: 90; go: 100.
```

makes the turtle draw a square.

Kay's idea enabled the distributed systems of figure 1 to be populated by Smalltalk objects. Research showed that the idea of objects were natural even to children, who could use Kay's ideas to write simple programs. (figure 2) Kay and every one of his colleagues at Xerox Palo Alto Research Center (PARC) had their own Alto computer as shown in the figure. All were connected through a very fast Ethernet network. This realized the hardware dream of Prokon (figure 1).

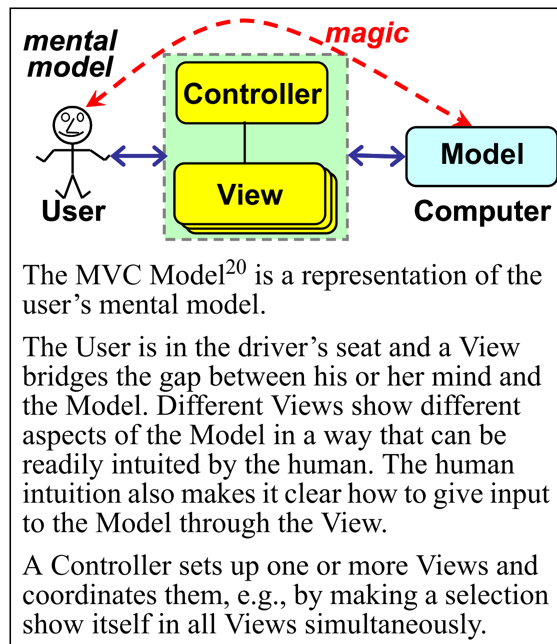
The DCI architecture is object-oriented, but with the addition of algorithms for object collaboration in the DCI Contexts.

2.4 MVC - the Model, View, Controller Paradigm

Trygve had the privilege to work as a visiting scientist at PARC in 1978/79. Here, the Prokon hardware vision was reality and the Prokon communicating computers could be simulated with Smalltalk programs.

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

Figure 3: The Model-View-Controller-User paradigm.



The MVC Model[20] is a representation of the user's mental model.

The User is in the driver's seat and a View bridges the gap between his or her mind and the Model. Different Views show different aspects of the Model in a way that can be readily intuited by the human. The human intuition also makes it clear how to give input to the Model through the View.

A Controller sets up one or more Views and coordinates them, e.g., by making a selection show itself in all Views simultaneously.

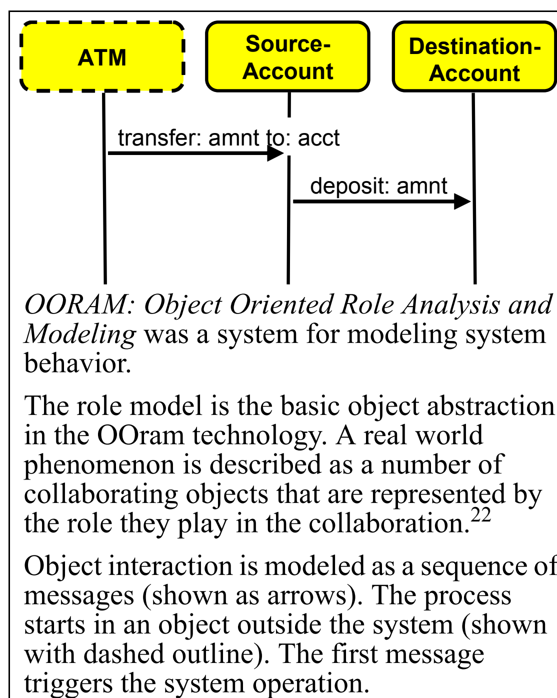
While at PARC, Trygve focused on the problem of a person interacting meaningfully with a complex activity network plan through a small computer screen. The Alto bitmapped display and mouse pointing device opened fascinating opportunities. But the plan was still large and the screen still small. An outcome of the research was the Model-View-Controller (MVC) paradigm. (figure 3) The paradigm sustains Douglas Engelbart's vision of computer augmentation by which computers extend the human intellect and improve human collaboration.[6] This 'magic' is attained when the Model object seamlessly represents the human mental model. A View presents some aspect of it to the user in an intuitively obvious way.

> *"There should be a one-to-one correspondence between the Model and its parts on the one hand, and the represented world as perceived by the owner of the Model on the other hand. The nodes of a model should therefore represent an identifiable part of the problem."*[20]

MVC and DCI are complementary paradigms. MVC transforms the Model data into a physical form that can readily be assimilated by the user's brain. DCI is about creating a program that faithfully represents the human mental model. The two meet when MVC is used to bridge the gap between the human mind and the model implied by the DCI-based computer system.

2.5 OOram Role Modeling

Figure 4: OOram role modeling.



OORAM: Object Oriented Role Analysis and Modeling was a system for modeling system behavior.

The role model is the basic object abstraction in the OOram technology. A real world phenomenon is described as a number of collaborating objects that are represented by the role they play in the collaboration.[22]

Object interaction is modeled as a sequence of messages (shown as arrows). The process starts in an object outside the system (shown with dashed outline). The first message triggers the system operation.

OOram, Object Oriented Role Analysis and Modeling, modeled the behavior of an object system as a flow of messages between participating objects. Its main innovation was the *Role* that identified an object in a network of interacting objects.[22]

In section 4, we will give a simple example where a person uses a bank terminal (ATM) to transfer money from one account to another. Figure 4 shows the OOram scenario diagram of a simplified computer process for this example. Many different kinds of accounts can stand in for both the SourceAccount and DestinationAccount roles.

OOram's essential contribution is the notion of a role as the name of an object according to its use. OOram - like its descendant, the UML collaboration[26] - looks only at the sequence of messages and doesn't exhibit the code that makes this happen. A subsequent series of evolutionary steps has led to DCI with its role methods that make the messages flow according to plan.

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

3 DCI, the new Programming Paradigm

Separation of concerns is a powerful strategy to master complexity. A well-known example is the traditional data-centric architecture with a central *database* and *application* programs arranged around it. The database contains pre-formed data that represent information in the user's mental model. Data processing is realized by different application programs. Each application can access the database through a *bridge* (external view) that is tailored to its particular needs. Further separation of concerns can be achieved with functional decomposition of each application.

The DCI architecture resembles the traditional database-centered architecture in its separation of static data and dynamic system operations. The *Data projection* is like a database description and describes interesting data. The *Context projection* is like a traditional application program and describes the functionality associated with a particular requirement such as a use case scenario. The *Interaction* is part of a Context and is a functional decomposition of the Context functionality. This functionality is realized by the interaction of participating objects that are known by the *roles* they play in the Interaction.

> *DCI sees a program in two orthogonal projections. The Data projection describes system state. The Context projection describes system behavior; there is one of the latter for each use case scenario.*

The DCI object is a specialization of Kay's "an object is like a computer" (section 3.1). The traditional bridge between a database and an application is replaced by a runtime selection in the Context that maps roles to objects according to their use. Typically, the roles will be mapped to different objects in different executions. This mapping maintains the consistency between otherwise independent Data and Context projections. The projections can, therefore, evolve at different rates and can be implemented and tested by different people. We have chosen to discuss the Context with its Interaction first since this is a DCI innovation. (section 3.2) We describe the Data last since this is merely a stripped-down version of the well-known class-oriented programming (section 3.3). Further, Contexts can be tested on preliminary data as long as the objects fulfill the role expectations.

3.1 The DCI Object and its Abstractions

Kay's notion of object orientation (section 2.3) defines an object as a self-contained entity that has all the capabilities of a computer. We add ideas from Prokon (section 2.1) and OOram (section 2.5) to define the DCI object. This object has five basic properties that are essential to the DCI paradigm:

Static properties

State	*Like a computer,* the DCI object has memory called its *attributes*. Think of an object as a database record that is encapsulated within the object boundary.
Behavior	*Like a computer,* the DCI object can process data with its *methods*. Think of them as local procedures that are visible only within the object.

Dynamic properties

Encapsulation	*Like a computer*, a DCI object is encapsulated within an abstraction boundary. The object presents a message interface to its environment just as the computer presents an instruction repertoire. The actual realization in software or hardware is not visible outside the object's boundary. Different objects may invoke different methods for the same message, this is called *polymorphism*.
Communication	*Like a computer,* a DCI object can collaborate with other objects through message interaction. Communication is now a first class citizen of computer programming.
Identity	*Like a computer*, a DCI object has a unique and immutable *identity.* This is essential for reasoning about networks of interacting objects.

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

DCI systems combine objects with the above simple capabilities in various ways to enable millions of different applications.

> *A DCI object is encapsulated, its inner construction is invisible from the outside. Consequently, the object's inside can be anything: A network of communicating objects, a Fortran program, an SQL machine, a state machine, a Petri net, or it can follow any other paradigm. The DCI Object supports multi-paradigm design.*[2]

DCI uses two abstractions[a] on objects, the class or its equivalent and the role. The class is the predominate object abstraction used in current programming and research. We only consider object state and behavior as the only properties relevant here; the communication properties are ignored. The DCI Data projection is expressed with classes or their equivalent. The notion of interacting objects is outside the scope of the Data projection.

The DCI *role* is a new and equally important abstraction on objects. A role names an object as it collaborates with other objects at runtime (OOram, section 2.5). The DCI Context projection is expressed with roles. The object's inner construction, possibly with class and superclasses, is outside the scope of the Context projection.

Object encapsulation demarcates the boundary between the class and role abstractions. Roughly speaking, the class is on the inside and the role is on the outside of this boundary.

> *The role abstraction is the dual of the class abstraction. The role abstraction says nothing about the inner structure of an object but says everything about how the object is used together with other objects. The class abstraction says everything about the inner construction of an object but says nothing about how it is used in interaction with other objects.*

3.2 The C and I in DCI stands for Context and Interaction - What the system Does.

Data classes give rise to the objects that interact to implement system operations. Object interaction takes place within a Context where the objects are identified by the Roles they play. Objects are temporarily extended with Role Methods while they are playing a role. Functional decomposition is used to distribute the Interaction algorithm onto participating role-playing objects and thence to their role methods. Role methods are associated with the roles rather than with classes. This means that we can reason about system operations without having to study the classes of the role-playing objects. As Brian Kernighan characterized C functions, a method should do one thing and do it well. Each should fit on one or two screens of text. [30]

> *A role method creates an ephemeral extension of object functionality while it is needed at runtime. Role methods can, therefore, extend instances of library classes without having access to those classes.*
>
> *An object acquires additional functionality in different Contexts while still retaining its identity.*

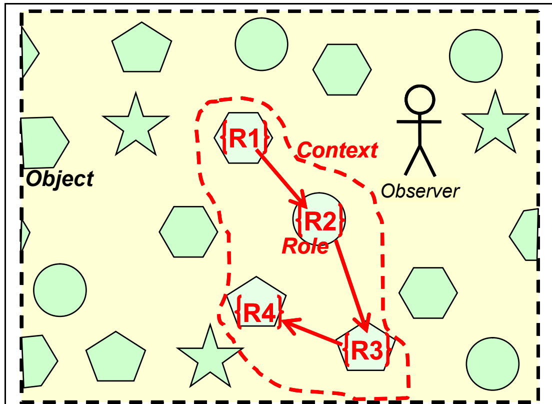
It is through this dynamic role-based extension mechanism that DCI implements polymorphism and its accompanying succinctness. In traditional object-oriented programming, many different kinds of objects could satisfy a given request for service from a client. In DCI we instead think in terms of service aggregates called roles. Many kinds of object can play each role, and many combinations of kinds of objects can play the roles of a given Context. This "many kinds" property is where DCI implements the object notion of polymorphism.

---

a. Wikipedia defines: "Abstractions may be formed by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose"[28]

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

The Context regards objects only in terms of their identities and the interfaces they provide. Their actual construction is irrelevant. The writers of role methods use their domain knowledge to understand and employ these interfaces. They must accept the provided interfaces on trust since they cannot accurately identify the corresponding class-defined methods. This trust is well placed because the methods visible in the Data perspective, like C functions, compute only simple operations on the data within their domain of responsibility. These methods will not, at the level of the active design discourse, trigger ensuing execution sequences across object boundaries.



This figure shows a universe of communicating objects. Instances of different classes are shown as different shapes.

An observer placed in the space between the objects can trace the messages that flow through an ensemble of objects during the execution of an operation. Different executions will typically involve different sets of objects. DCI requires that the topology of the traces must be the same for all executions of the same operation.

The topology is a directed graph where the nodes are roles and the edges are connectors. (Shown colored in the picture). The roles are marked {R1} through {R4}, the connectors are shown as arrows. The ensemble of objects is mapped on to the roles within a *Context*. This is the *form* of the execution

At runtime, a Context musters the objects that shall play its roles and starts the flow of Interaction messages that achieves the required operation.

According to *Dictionary.com*, the origin of the word role is the French rôle roll (as of paper) containing the actor's part. An actor was reading from this roll while performing his part. We analogously extend the roles with *role methods* that extend the behavior of objects while they play the role at runtime.

Figure 5: Object, class, and role.

Figure 5 illustrates how the role abstraction supports the specification of an ensemble of collaborating objects.

The value of a class-oriented system is maximally the sum of its parts. The addition of explicit information about the runtime relationships between the parts can make the value of a DCI system greater than the sum of its parts.

3.3 The D in DCI stands for Data - What the system Is.

Many objects represent ideas in the user's problem domain. Other objects are helpers such as values and collections that are reflected in the programmer's mental model. An MVC View can play a role in an Interaction that updates the View contents; the roles being View and Model. An object can even be a Context that plays a role in an outer Context, thus supporting recursion.

An object is an entity that encapsulates state and behavior. An object can play a role in a context. It is created as an instance of a class, a copy of a prototype, or some other construction mechanism. For convenience, we use the word *class* for all such mechanisms.

Classes can be very simple since all system functionality is moved to the Contexts. Informally, we say that an object is unaware of its environment.

In the introduction, we defined *information* as "the meaning that a human assigns to data by means of the known conventions used in its representation." An implementer of a class uses those conventions and a code reader applies them to make sense of the code. Both see the instances from their inside and reason about each class in isolation. The writer of the class takes responsibility for its correct implementation to permit the writer of a role method to take the object's interface on trust.

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

Like a computer, a root object does not expose how it reifies the messages in its interface. The object's boundary forms, by definition, an abstraction boundary. In contrast, the role methods in a DCI Context are outside the object's abstraction boundary; they are compressions that are open to reading and understanding, rather than abstractions whose correct functioning is left to trust.

4 Money Transfer: A Simplified Example

Assume we shall build software for an Automated Teller Machine (ATM) and that one use case is to support an end user who transfers money from one bank account to another. User, programmer, and bank expert cooperate to capture a shared mental model according to the lean principle "Everybody, all together, all the time".[3] The team elicits the end user's description of what he or she wants to do. One possible response could be: "*Well, I choose an account and a transfer amount, and then I choose another account, and ask the system to transfer that money between the accounts*".

The programmer settles on MVC for the user interface. The implementation of Controller and View is plain programming and will not be discussed here. What remains is the Model, i.e., the banking side of the solution. The bank expert knows that what really matters is not the accounts, but the bank's ledger; an unordered set of immutable bank transaction records. It is the programmer's responsibility to unify the user's account model and the bank's transaction model. He or she decides to let the account objects be caches on the ledger and thus accommodate both. We ignore the ledger in this simplified example and refer the interested reader to a more realistic program in the DCI Home page[5].

The task is to build a bridge between the end user's mental model with its accounts and the bank with *its* accounts. We arbitrarily choose to describe the Data projection first and the Context later. The code is written in Smalltalk because this language has been explicitly designed to be readable by non-experts. Many experts find the unusual Smalltalk syntax a barrier, but it takes little effort to surmount it. (See the explanation before the Smalltalk examples in [5]).

4.1 Data projection

We may ultimately implement a database schema for the bank with appropriate Data classes, but start with dummy classes to give the Context programmer something to use in his or her tests. One Data class is sufficient in this simple example

```
Object subclass: #Account
        instanceVariableNames: 'balance'

" External interface methods. "
balance
   ^balance

decrease: amount
   balance := balance - amount.

increase: amount
   balance := balance + amount.
```

There could be subclasses such as CheckingAccount, SavingsAccount, and LoanAccount. We ignore them here because the corresponding roles can be played equally well by instances of any of them.

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

4.2 Context projection

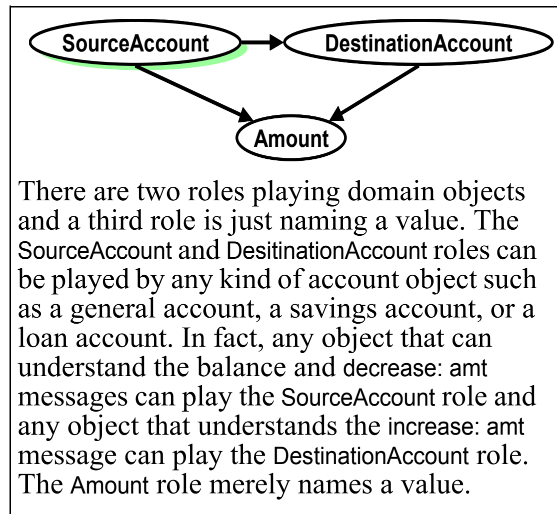


Figure 6: What the system does; the MoneyTransfer Context.

There are two roles playing domain objects and a third role is just naming a value. The SourceAccount and DesitinationAccount roles can be played by any kind of account object such as a general account, a savings account, or a loan account. In fact, any object that can understand the balance and decrease: amt messages can play the SourceAccount role and any object that understands the increase: amt message can play the DestinationAccount role. The Amount role merely names a value.

The use case is reified in the MoneyTransferContext class. The role topology is shown in figure 6. The Context has three roles that stem from the end user mental model: SourceAccount, DestinationAccount, and Amount.

A command from the user is passed through the View and triggers the system operation in the Context: transfer: amt from: account1 to: account2.

The corresponding Context method first binds roles to objects before it triggers the first method in the first role in the Interaction as follows:

```
MoneyTransferContext>>transfer: amt from: account1 to: account2
    self newRoleMap
        at: #Amount put: amt ;
        at: #SourceAccount put: account1 ;
        at: #DestinationAccount put: account2 ;
    self triggerInteractionFrom: #SourceAccount with: #transfer.
```

4.2.1 Interaction

A model of the interaction is shown in figure 4. The actual code shows the details. The Context triggers the flow of messages in the transfer role method:

```
" Role method. "
SourceAccount >> transfer
    self balance >= Amount
        ifTrue:
            [self decrease: Amount.
            DestinationAccount deposit]
        ifFalse:
            [self error: 'Insufficient funds'].

" Role method. "
DestinationAccount>> deposit
    self increase: Amount.
```

5 Related Work

DCI has strong echoes of ideas that came and went before it, many of which attempted to address related problems with object orientation since its early days. In the same sense that DCI breaks the common Cartesian classification found in class-oriented programming, so did many of these earlier concepts and features. Cannon's Flavors[1] offers “mix-ins” as a way to associate multiple lightweight classes and their methods with a single object. However, Flavors has no notion of sequencing the “mix-in” methods and no way to associate stand-alone “mix-ins” in a standalone (i.e., without classes or objects) execution graph.

Steele's multiple dispatch[25] provided a way to associate multiple objects through a single operation that engages all of them. Different combination of object types are mapped onto different method selectors. Multiple dispatch is somewhat like DCI inside-out: no single sequencing of role methods serves all combinations of object types, but rather each combination

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

of object types implicates a method suitable to that combination, which in turn sequences actions upon those instances.

The **self** language of Ungar and Smith[27] has strong facilities to encourage thinking in terms of objects instead of classes, which guards against class-oriented thinking. But, again, there is no focus on a single locus of recurring execution sequence analogous to a DCI Context.

In many ways, DCI implements one deeper level of reflection than its weaker cousin that supports the polymorphism found in most modern object-oriented programming languages. The original vision of Aspect-Oriented Programming (AOP)[14] was also rooted in reflection, and also arranged to factor out scattered implementations of key design concerns into a central concept called an Aspect. However, Aspects tend to focus on multiple insertions (at joinpoints) of a single change (advice) rather than on the coordinated introduction of sequenced methods across an arbitrary set of objects. Its mechanisms tend to be class-oriented rather than encoding any system-level view of what objects should play which roles. AOP tends to operate at the level of the programming language execution model, while DCI tends to operate at the level of business concepts. Aspects tend to erode code readability while DCI enhances it.

DCI is in many ways similar to Actors[10], but in the end is fundamentally different. Both take the triad of store, transform (or process) and communicate as their foundation. Actors is based on a many-to-many addressing model whereas DCI is based on a one-to-many association model between roles and objects and a fixed role method sequencing taxonomy.

ObjectTeams is a separate effort that emphasizes separate run-time entities for roles and the objects that play them.[9] Its goals are similar to those of DCI, but its failure to maintain object identity introduces errors into algorithms that depend on it, as can be demonstrated with a simple program.[4] ObjectTeams converted its terminology to be consistent with DCI terminology in 2013.

6 Future Work

A concrete vision that foresaw today's state of DCI dates back to about 2003, which means that DCI today may be where original object orientation was in the early 1980s. Work remains to further formalize the DCI metamodel. There may be interesting work to be done on concurrency in ways that reflect the original Simula goals of simulated or real parallel time threads, and to evaluate how those play with the single-thread execution model of DCI Contexts. The constraints that Contexts place on object interactions offer the possibility of formal program analyses that were impossible in the past without sacrificing polymorphism.

7 Conclusion

We started this article with a quote[8], saying that an object-oriented program has two structures. The first is the code structure; it is frozen at compile time and consists of a hierarchy of classes. The second structure is orthogonal to the first and “consists of rapidly changing networks of communicating objects”. In that world “the code won't reveal everything about how a system will work”. The problem was clearly that current technology does not offer a concise way of coding the process of computation.

> *Our solution is DCI with its static Data projection showing a hierarchy of classes and the Context projection that realize use case scenarios and other system operations as dynamic networks of communicating objects.*

Section 2 chronicled more than 40 years of gradual evolution to towards the simple solution called DCI. DCI meets the 5 goals that were listed in the introduction:

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

Mental Models. There is ample evidence from a variety of people ranging from professionals to children that object models fit well to the human mind. DCI achieves the 'magic' of figure 3.

***Reasoning**.* We work with a DCI program in the orthogonal Data and Context projections. This conceptually simple, yet effective two-dimensional representation enables a developer to reason about one dimension at the time.

Readability. Readable code is code that is cleanly partitioned and that clearly exhibits the system design.[23] The two-dimensional representation in DCI provides such independent partitions.

Reuse. There are two opportunities for reuse with DCI. One is that the classes are self contained. They are independent of their environment and can be reused for other purposes. The second is that a Context implements a system operation. This Context can be used by another, outer, Context analogously to a subroutine. This reuse reflects the recursive nature of DCI.

Revision. Data and Context are orthogonal; this invites independent evolution. A slowly evolving Data structure in the classes is decoupled from the more rapidly changing business logic in the Contexts. Contexts can be added, deleted, and changed independently of the other parts of the system.

Proof-of-concept implementations in Squeak and Marvin[5] show that the DCI paradigm can be expressed in suitable programming languages and supported in effective development environments. Interesting developments are happening with adapting a number of languages to DCI. The goal is to make a program design conform to an end user's mental model of a system and clearly express it in code that reveals how a system will work.

> *"More than twenty years of experience has shown us that a bad system design can never be hidden from the user, even by a masterfully devised user interface. A quality system, therefore, must be based on sound design that can be described in terms with which the user is familiar."*[21]

Software creates value only when an end user executes it for a purpose. History and common sense argue that users can reap the full value only when they understand how the system works; the ideal being that a stakeholder can write his or her own code. We have shown that DCI is well attuned to the human mind. DCI can, therefore, be a key to user understanding of what goes on in the computer when he or she applies it to his various tasks. Indeed, we claim that DCI is so powerful that it can form a conceptual foundation for expert programmers and it is so simple and universal that it can form a foundation for children learning about computing as a part of their four Rs: Reading, wRiting, aRithmetic, and pRogramming.

> *At long last, communication is a first class citizen of programming. Store, process, and communicate, the primitives of computing that are captured in the DCI paradigm. So simple that everybody can understand it, so universal that it can form the nucleus of computing in business and school.*

8 Acknowledgements

Gertrud
Rune
Morten
Bran
Risto
+++

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

9 References

.

1	Cannon, Howard. *Flavors: A non-hierarchical approach to object-oriented programming.* http://www.softwarepreservation.org/projects/LISP/MIT/nnnfla1-20040122.pdf, 1979.
2	James O. Coplien: Multi-Paradigm Design for C++; Addison-Wesley Professional; 1998; ISBN 0201824671
3	Coplien, J.O., Bjørnvig, G; *Lean Architecture for Agile Software Development*; Wiley, Chichester, UK, 2010; ISBN 978-0-470-68420-7
4	Coplien, James O., and Trygve Mikkjel Heyerdahl Reenskaug. *The data, context and interaction paradigm.* In Gary T. Leavens (Ed.): Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012. ACM 2012, ISBN 978-1-4503-1563-0, pp. 227 - 228.
5	The DCI Home page. http://fullOO.info
6	Douglas C. Engelbart: *Augmenting Human Intellect: A Conceptual Framework.* Summary Report AFOSR-3223 under Contract AF 49(638)-1024, SRI Project 3578 for Air Force Office of Scientific Research, Stanford Research Institute, Menlo Park, Ca., October 1962: www.liquidinformation.org/ohs/62_paper_full.pdf
7	Rune Funch Søltoft: *Marvin*. https://github.com/runefs/Marvin
8	Gamma et.al.: *Design Patterns.* Addison Wesley1995
9	Hermann, Stephan. *Demystifying object schizophrenia.* MASPEGHI Workshop (MechAnisms for SPEcialization, Generalization and inHerItance), at ECOOP'10, Maribor, Slovenia.
10	Hewitt, Carl ; Bishop, Peter; Steiger, Richard(1973). *A Universal Modular Actor Formalism for Artificial Intelligence.* IJCAI.
11	*IFIP-ICC Vocabulary of Information Processing*; North-Holland, Amsterdam, Holland. 1966; p. A1-A6.
12	Kay, Alan (1972). "A Personal Computer for Children of All Ages". http://www.mprove.de/diplom/gui/kay72.html
13	Kay, Alan: *The Early History of Smalltalk*; ACM SIGPLAN Notices archive; 28, 3 (March 1993);pp 69 - 95
14	Kiczales, Gregor et al. *Aspect-Oriented Programming.* Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.
15	Liskov, Barbara. *Data Abstraction and Hierarchy*. SIGPLAN Notices 23, 5 (May 1988).
16	National curriculum in England: computing programmes of study; [WEB PAGE] https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study
17	Nygaard, K. Dahl, O.-J.; *Simula: An ALGOL-based simulation language*; Communications of the ACM **9** (9) (1966): 671. doi:10.1145/365813.365819
18	Reenskaug, T.; *Administrative control in the shipyard.* ICCAS conference, Tokyo, 1973. Scanned by the author July 2003 to http://heim.ifi.uio.no/~trygver/1973/iccas/1973-08-ICCAS.pdf
19	Reenskaug, T.; "*Prokon/Plan-A Modelling Tool for Project Planning and Control*", in Proc. IFIP Congress, 1977, pp.717-721.
20	Reenskaug, T.: *The original MVC reports*. Xerox PARC 1978; [Web page] http://www.duo.uio.no/sok/work.html?WORKID=52648

 ©2014 Trygve M. H. Reenskaug and James O. Coplien

21	Reenskaug, T.: *User-Oriented Descriptions of Smalltalk Systems*; Byte Magazine, August 1981.(The special issue on Smalltalk.)
22	Reenskaug, T.et.al.: *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
23	Reenskaug, T: *The Case for Readable Code*; Klein: Computer Software Engineering Research; Expert Commentary; pp. 3-8; Nova Science Publishers, New York, 2007; ISBN-13: 978-1-60021-774-6. [WEB PAGE] http://heim.ifi.uio.no/~trygver/2007/readability.pdf
24	Reenskaug T.; [WEB PAGE] http://folk.uio.no/trygver/2009/commonsense.pdf
25	Steele, Guy L. "chapter 28", *Common LISP: The Language.* Bedford, MA, U.S.A: Digital Press, ISBN 1555580416, http://books.google.com/books?id=8Hr3ljbCtoAC, 1990.
26	*OMG Unified Modeling Language (OMG UML), Superstructure.* formal/2012-05-07; Object Management Group April 2012; ISO/IEC19505-2:2012(E) http://www.omg.org/cgi-bin/doc?formal/12-05-07.pdf
27	Ungar, Smith: *Self, the power of simplicity.* http://labs.oracle.com/self/papers/self-power.html, 1987.
28	http://en.wikipedia.org/wiki/Main_Page
29	Rickard Öberg: What is Qi4j? http://qi4j.org/
30	Brian W. Kernighan, P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill Book Company, Second Edition 1978, ISBN 0-07-034207-5, pp 59-64.

 ©2014 Trygve M. H. Reenskaug and James O. Coplien