Working with objects — in computer and mind

Trygve Reenskaug,
Dept. of Informatics,
University of Oslo
Norway
(“Trygve”)

James O. Coplien
Gertrud & Cope
Copenhagen
Denmark
(“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-1.14.3 - 2/17/15 - Updated section 3.1 + other changes inspired by several comments.
draft 1.9 - Feb. 20-2014- Corrected misprints MB.
draft 1.10 - June 16 2014 - New, large example
draft 1.10.1 - June 28 2014 - Minor changes
draft 1.10.2 - July 11 2014 - Minor changes
draft 1.11.01 - Aug. 3, 2014. Major rewrite in several stages. Concepts presented on the background of an example. System behavior in the forefront.
draft 1.11.02- Body of article rewritten. MVC removed. Example marked blue in preparation for new example.
draft 1.12.01 - Revised from Bruce comments Prokon example retained, its presentation in the paper completely reworked.
draft 1.13 - Revised section 4: Example
draft 1.14 - General revision + extended section 4: example program.

Abstract

After more than 60 years with computers, hundreds of millions of people are dextrous 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 civilized life.

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 novice can write simple code. 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 many different programming languages. The DCI concepts can become a unifying foundation for programming.

A DCI program is specified in two orthogonal *projections*: The *Data* projection describes what the system *is*, its static properties. The *Context* projection describes what the system *does*, its runtime behavior.

Draft

Key Insights

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

1 Introduction

This article is about people. Our target is the general computer user, but we will especially focus on professional users who apply digital systems to improve the performance of their tasks. Their mental models will be grounded in the disciplines that drive their work and our goal is to write

computer programs that feel like extensions of their minds. The lean principle "everybody, all together, all the time"[3] says that the user shall be an active member of the development team.

We propose a new programming paradigm that we call *DCI -- Data, Context, and Interaction* to close the gap between mind and computer. Users who understand simple code can explore a program's capabilities and suggest well-founded improvements. 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 systems actually work. 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.*

The DCI focus on the end user's mental model makes it imperative to distinguish clearly between what is in the user's mind and what is represented in the computer. These IFIP definitions of 1966[11] have withstood 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."

There is no information in a computer system, not even in the World Wide Web; there is only data. Any human who understands the conventions can convert data to information. Such understanding is the first stage of computer literacy.

An essential part of the data existing in any computer system is the code that controls it. Any human who understands the code conventions can build a mental model of how the system works. Such understanding is the second stage of computer literacy.

Fundamentally, a computer offers three simple services: It can store data, it can transform data, and it can communicate data -- so simple, yet so powerful when combined in various ways into comprehensive programs. The DCI challenge is to reflect this fundamental simplicity of computing in the programs we write and use. The human mental model of a computer program shall be based on DCI and be reified into readable code[24]. There shall be no surprises.

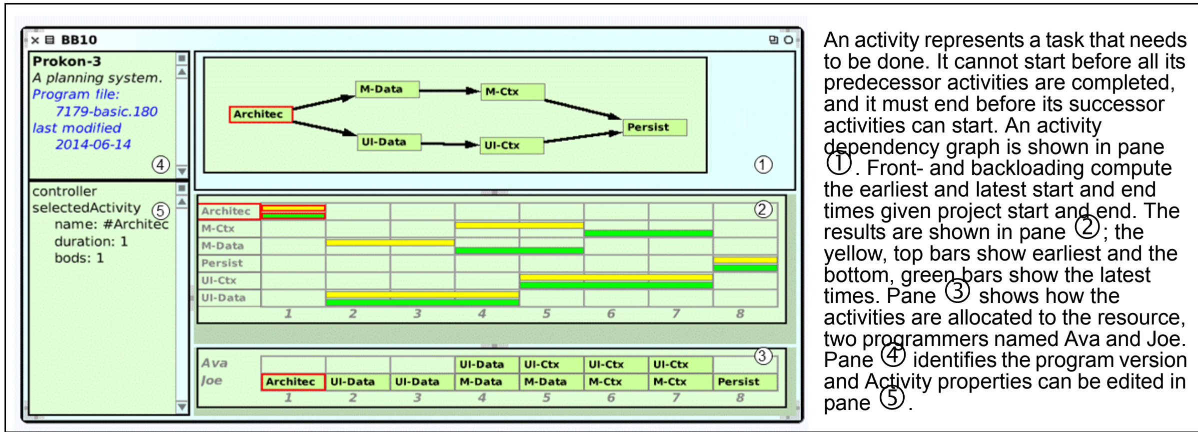
The main goals of DCI are:

> MENTAL MODELS. To reflect the way different users conceptualize the objects of their world so that a program 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.

DCI is a general paradigm for computer programming and is applicable to many different problem areas and programming languages. We will evolve the conceptual model of DCI in the body of this article and simultaneously build a concrete program to illustrate the ideas. The example is *Prokon*, the activity network planning tool shown in figure 1.

Draft

Figure 1: Prokon, an activity network planning tool.



An activity represents a task that needs to be done. It cannot start before all its predecessor activities are completed, and it must end before its successor activities can start. An activity dependency graph is shown in pane ①. Front- and backloading compute the earliest and latest start and end times given project start and end. The results are shown in pane ②; the yellow, top bars show earliest and the bottom, green bars show the latest times. Pane ③ shows how the activities are allocated to the resource, two programmers named Ava and Joe. Pane ④ identifies the program version and Activity properties can be edited in pane ⑤.

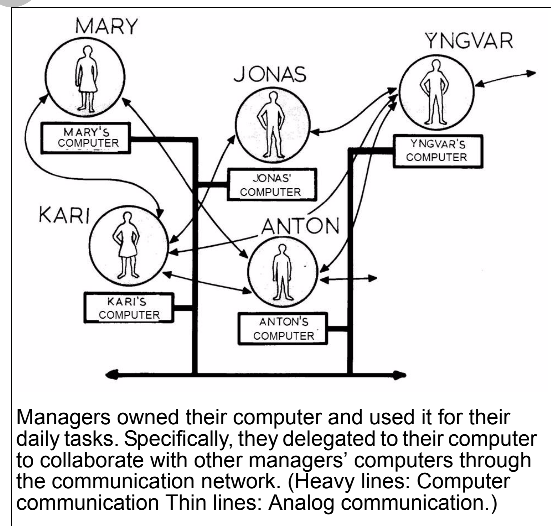
DCI is firmly interwoven with the notion of objects. This notion has matured over the years, and in section 2: *The Roots of DCI* we glean powerful concepts created over the past four decades and see how they contribute to the DCI paradigm. The DCI paradigm presented in section 3: *DCI, the new Programming Paradigm*. Our example program is completed in section 4: *Prokon: Our Activity Network Planning Program*. 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 personal users and schoolchildren. More details about the example in the appendices.

Draft

2 The Roots of DCI

2.1 Prokon's Distributed Systems

Figure 2: Prokon, a distributed system architecture



Managers owned their computer and used it for their daily tasks. Specifically, they delegated to their computer to collaborate with other managers' computers through the communication network. (Heavy lines: Computer communication Thin lines: Analog communication.)

By 1970, it was clear that there was a fundamental lack of balance between the decentralized nature of an organization's distribution of responsibility and authority and the centralized nature of a database-centered system architecture.[19] The Prokon project proposed a distributed system architecture (figure 2) to restore the balance.

As an example, managers use their computer to create plans for their own area of responsibility. They would delegate to their computer to negotiate with other managers to create an overall plan. This created a need for algorithmic control over the communication as a whole. ("Local independence combined with central coordination"[20]).

Communication became a first class citizen of system architecture.

The project had many ideas that pointed towards general principles of system architecture:

- The end user is the defining entity for overall system architecture as well as system details.
- The communication bus connects autonomous computers that encapsulate state and behavior.
- There is algorithmic control over the communication as a whole.

- The architecture is recursive; a senior manager can be responsible for a group of junior managers that collaborate through a subnetwork.

The Prokon project lost its funding and its distributed system was never built. A user interface for managers was written in Smalltalk. It was similar to our example program and led to the MVC programming paradigm (appendix 2).

Prokon gave a glimpse of a possible architecture for our example program: Divide the system into autonomous components. Let each component be responsible for storing its part of the system data and for transforming these data as required. Let these components interact through message interaction. Create system level interaction algorithms that achieve the program's behavior.

2.2 The First Object

Nygaard and Dahl's concept of *objects* was realized in the programming language Simula 67[19]. 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 behavior of each object of the class. (We shall later use the terms *attributes* for the data structure and *methods* for the behavior.)

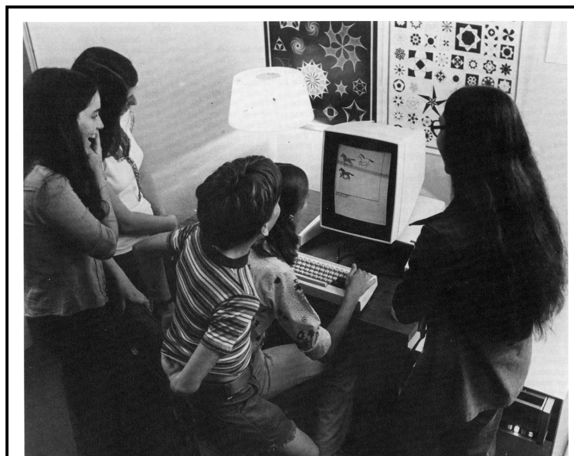
> *The Simula experience indicated that complex physical systems could be naturally reflected in mental and computerized object models.*

Draft

Simula objects could apparently reify the Prokon components. This idea was quashed by the tight coupling of Simula objects through coroutines while Prokon components were loosely coupled through message interaction.

2.3 Kay's Object Orientation

Figure 3: 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.

The children used Kay's ideas to write quite complex programs (See the results on the wall). The research showed that the notion of objects formed a solid foundation for children's programming.

<TBD> Picture © Alan Kay?

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 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]

In most computers, data is represented as binary words in the memory and data processing takes place as defined by IFIP[11]:

> "DATA PROCESSING. The execution of a systematic sequence of operations performed upon data."

This definition reflects how computers are constructed with a data store and a control unit that extracts a stream of instructions from the store and executes them one by one. This hardware model can be found at the core of most programming languages.

Alan Kay broke with this tradition with his definition of object orientation. At the core of Smalltalk there is an object store and data processing is the systematic flow of messages between objects. Smalltalk is a first example of an *object computer*. All information of interest, and this includes computer programs, is represented as objects. Data is stored in objects. Data is transformed by objects. Data is communicated between objects.

> *Kay's object model is a new foundation for thinking about and programming computers. It seems to have a a good fit with the children's mind (figure 3). Simula (section 2.2) and OOram (2.4) suggest that the same applies to professionals in business and industry.*

The object computer lets us hide many low level details in order to create models that are closer to the human mind. There is no loss of substance; we can still store, transform, and communicate data recursively. An object computer can be built in hardware or, which is more common, we can emulate it on a conventional computer.

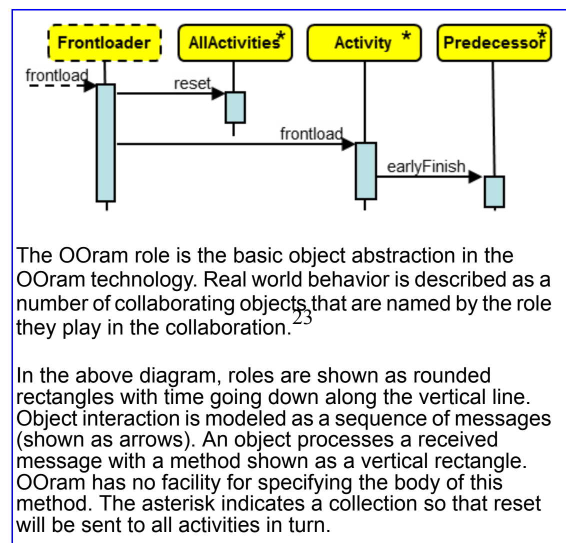
> *We decide to construct our example program as an ensemble of communicating objects.*

Draft

2.4 OOram Role Modeling

OOram, Object Oriented Role Analysis and Modeling, models the behavior of an object system as a flow of messages between participating objects.[23] Its main contribution is the notion of a *Role* that identifies an object in a network of interacting objects according to its use. There is an analogy with the theatre: A human plays a role in a play; an object plays a role in a role model. Both are dynamic; they relate to what the system *does* rather than what it *is*.

Figure 4: OOram role model -- frontloading.



The OOram role is the basic object abstraction in the OOram technology. Real world behavior is described as a number of collaborating objects that are named by the role they play in the collaboration.[23]

In the above diagram, roles are shown as rounded rectangles with time going down along the vertical line. Object interaction is modeled as a sequence of messages (shown as arrows). An object processes a received message with a method shown as a vertical rectangle. OOram has no facility for specifying the body of this method. The asterisk indicates a collection so that reset will be sent to all activities in turn.

The message sequence chart in figure 4 illustrates a very simple algorithm for frontloading in our example (② in figure 1). An object named Frontloader receives a message frontload. This triggers a method that first sends reset to all activities before it asks the activities to frontload|. The activity asks its predecessors for their early finish times.

An OOram role can be played by objects of different kinds so that the actual methods that will be executed is unknown. OOram - like its descendant, the UML collaboration[27] - observes black box objects. We see that activities are asked to frontload; we miss the crucial selection of the sequence of activities visited (section 4.1). OOram is a modeling tool.

3 DCI, the new Programming Paradigm

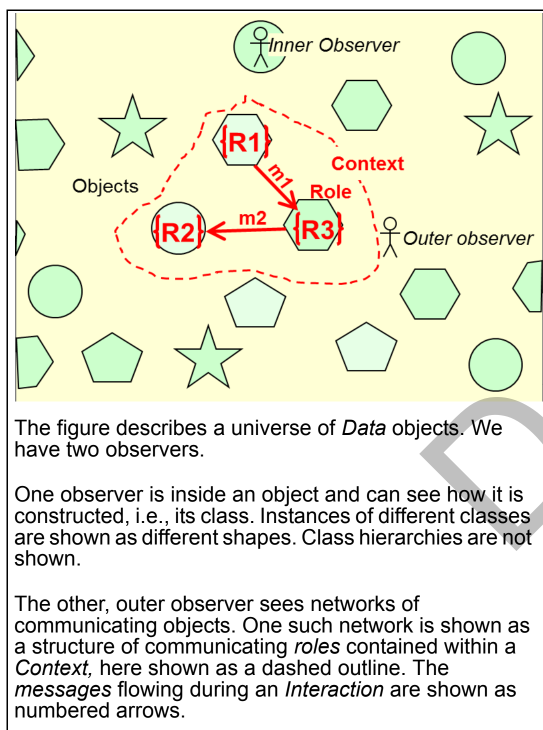
We quoted from the Design Patterns book in the introduction. Here's the full quote[8 pp. 22-23]:

> "An object-oriented program's run-time 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. A program's run-time structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa."

and:

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

DCI -- Data, Context, and Interaction, is the new paradigm that fills the void by saying everything about how a system works at run time. It achieves this by separating the program into largely independent *projections*. The code for the compile-time structure is captured in the Data projection; it consists of classes that have been stripped of code for object interaction. This projection is a reification of the user's information model (section 3.2).



The figure describes a universe of *Data* objects. We have two observers.

One observer is inside an object and can see how it is constructed, i.e., its class. Instances of different classes are shown as different shapes. Class hierarchies are not shown.

The other, outer observer sees networks of communicating objects. One such network is shown as a structure of communicating *roles* contained within a *Context,* here shown as a dashed outline. The *messages* flowing during an *Interaction* are shown as numbered arrows.

Figure 5: Object, role and context

The code for the run-time structure is captured in the Context projection with networks of communicating objects. This projection is a reification of the user's mental model of how the system handles use cases with their scenarios and system operations. The Contexts are kept independent by having a separate Context for each operation. (section 3.3)

Draft

Figure 5 illustrates a universe of objects and how the two projections are visible to observers placed inside and outside the objects.

The essence of object orientation is that objects collaborate to achieve a goal.

3.1 The DCI Object and its Properties

It is time to refine our notion of an *object*. 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 2.4) to define the DCI object. This object has seven basic properties that are essential to the DCI paradigm:

Compile time properties

State — *Like a computer,* the DCI object can store data in its *attributes*. Think of an object as a database record that is encapsulated within the object boundary.

Own 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.

Run time 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. Its concrete 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*. ("Call By Intent")
Communication	*Like a computer,* a DCI object can communicate with other objects through message interaction.
Identity	*Like a computer*, a DCI object has a unique and immutable *identity.* This is essential for reasoning about networks of interacting objects.
Synergic Behavior	An object's behavior is composed from its own behavior together with the synergic behavior that the role acquires because it is a participant in a collaboration. (Role methods, section 3.3)
Processes	We tend to think of an object as running in its own process. This is consistent with the object models discussed in sections 2.1, 2.2, and 2.3. Most object-oriented programming languages are basically single process. Nevertheless, we tend to stick to multi-process mental models and pretend that the message passing is synchronous.

> *A DCI object is encapsulated so that 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 or a realization of any other paradigm. The DCI Object supports multi-paradigm design.*[2]

An object can be an instance of a class, a copy of a prototype[28], or some other construction mechanism. For convenience, we use the word class for all such mechanisms. The class is the predominate concept used in current programming and research. It considers the compile-time object properties as relevant; the run-time properties are ignored.

> *Like a computer, an object does not expose how it reifies the messages in its interface. The object's boundary forms, by definition, an abstraction boundary. In contrast, role methods 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.*

Draft

3.2 The D stands for Data - What the system Is*.*

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." The DCI Data projection uses classes to specify the information part of the user's mental model. A writer of a class builds on domain knowledge and its known conventions and a reader of the class use them to make sense of the code. Both can reason about each class in isolation because its instances are observed from within the object boundary. 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.

This trust is well placed because the methods visible in the Data projection are like C functions, they compute only primitive operations on the data within their domain of responsibility. These methods will not, at the level of the active design discourse, trigger message interaction outside object boundaries.

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 object can even be a Context object that plays a role in an outer Context, thus supporting recursion. Data objects are objects that is destined to play roles in contexts. They are often simpler than their regular counterparts because system behavior has been moved to the contexts. Informally, we say that an object is unaware of its environment. A Data class can be reasoned about and tested as a separate entity.

In our example, Data objects are activities, dependencies and the single resource. Activities are visible in panes ①, ②, and ③ of figure 1 as squares, bars, and annotations. Schemas for the

data objects are shown in figure 6. The corresponding classes follow trivially, and any object-relational impedance mismatch is minimized

Figure 6: The Prokon model objects are like rows in database tables:

CREATE TABLE activities

oop - OBJECTID - PRIMARY KEY	name - VARCHAR(50) - NOT NULL	duration - INTEGER - NOT NULL	earlyStart - INTEGER - NOT NULL	etc.
...	...	...	...	...
...	...	...	...	...

CREATE TABLE dependencies

fromActivity - OBJECTID - NOT NULL	toActivity - OBJECTID - NOT NULL
...	...
...	...

CREATE TABLE allocations

weekNo - INTEGER	activity - OBJECTID
...	...
...	...

There are also a stored procedure in the activities table:
earlyFinish = earlyStart + duration.

3.3 The C and I stand for Context and Interaction - What the system **Does.**

Draft

An outside observer can trace the messages that flow through an ensemble of objects during the execution of an operation. (figure 5) The topology of the trace is a directed graph where the nodes are *roles* and the edges are *connectors*. DCI requires that this topology stays the same for all executions of the same operation. This is the *form* of the execution. The ensemble of objects is mapped on to *roles* within a *Context* (section 2.4) with this selection:

<role player> = **select** <attribute> **from** <Data> **where** <condition>

This gives us a more precise understanding of Data objects:

> *Data objects are objects that are visible in a context for the use as role players. Any object can play a role in a context, but only some of them do so. The distinguishing mark is their use.*

Contexts yield a synergy effect; the value of a context is greater then the simple sum of its objects. This is partly caused by the context’s form, but also by the context’s *interaction algorithm* that computes a system operation.

The word *role* stems from the French rôle roll (as of paper) containing the actor's part[a]. The actor reads from this script when performing the role. In a similar vein, the context decomposes the interaction algorithm into scripts that are attached to its roles. These *role methods* manifest synergetic behavior that is not visible in the data classes; the Context regards objects only in terms of their identities and the interfaces they provide. Their actual construction is irrelevant. This means that we can reason about system operations without having to study the classes. 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. [31]

a. See dictionary.com

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

Typically, roles will be mapped to different objects in different executions. The mapping maintains the consistency between otherwise independent Data and Context projections.

Projections can evolve at different rates and can be implemented and tested by different people.

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.

3.4 Summing up

A class says everything about the inner construction of an object but nothing about how it is used in its interaction with other objects. A context says everything about how the objects are used but says nothing about their insides.

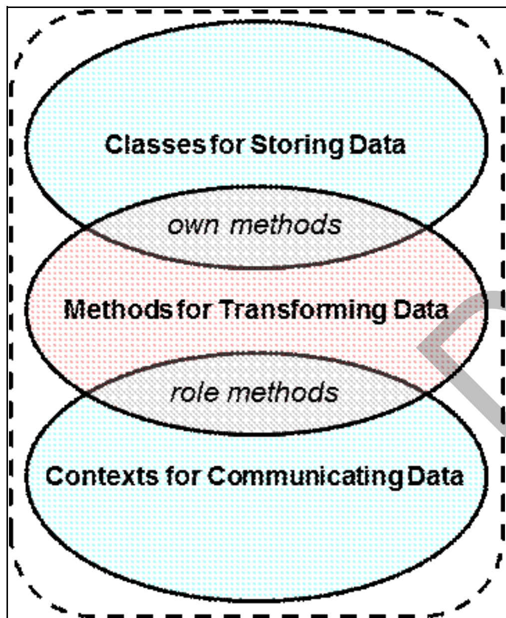


Figure 7: The basic capabilities of computers and objects.

In the introduction, we said that a computer offers three simple services: It can store data, it can transform data, and it can communicate data. Figure 7 shows how DCI supports these capabilities in a balanced manner within a system. Systems store data as specified in the Data classes, networks of communicating objects are specified in the Contexts, and the composite behavior of objects is specified in their own and role methods.The dashed outline indicates that an object can encompass all three capabilities. Recursion implies that an object can play a role in an outer context and that it can encapsulate an inner context.

Draft

Table 1 compares conventional programming with the DCI paradigm. We see that the class serves all purposes in conventional programming while object communication is coded explicitly in DCI programming.

Table 1: Objects collaborate to achieve a system operation/use case scenario.

	Conventional OOP	DCI
What are the participating objects?	Instances of relevant classes.	A Context selects objects to play its roles at runtime.
How are they interconnected?	There is no explicit specification of the communication network.	The network topology is explicitly specified in the Context.
What do they do?	System behavior is fragmented among the classes.	The role methods explicitly drive the interaction within a context.

4 Prokon: Our Activity Network Planning Program

We are now ready to implement the planning program of figure 1. The interface is designed to reflect the user's mental model of a *plan* with its structure of *activities*, *dependencies*, and a single *resource*. We ensure close correspondence between mental model and program by representing the model with an object that contains a similar structure of *activity*, *dependency*, and *resource* objects much as a database contains the corresponding records (figure 6). These objects together with their classes constitute the Data part of the program. This direct implementation of the human information model makes this part of the program easy to understand and protects the user against unpleasant surprises.

Human users understand how to perform operations on the plan: *frontloading*, *backloading*, and *resource allocation*. Each operation is implemented as a DCI context where the participating objects are identified by the roles they play during an execution. Each context is self contained so that it can be studied and tested in isolation.

The program as a whole is described in appendix 1. We will here concentrate on the frontloading and resource allocation operations as illustrations of how DCI handles collaborating objects.

4.1 Frontloading

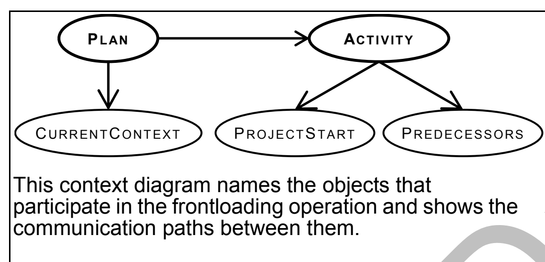


Figure 8: Frontloading Context Diagram

The goal of the frontloading operation is to compute the earliest possible start for every activity given the start of the project. We described a role model for the frontloading operation in section 2.4.We will now replace the unknown methods shown as vertical rectangles in figure 4 with DCI role methods.

Draft

DCI wants everything in one place, so we create a frontloading context that names and encapsulates the objects that participate in the execution of the operation. An activity can start as soon as all is predecessors are finished. We have two main roles: The ACTIVITY under consideration and the PLAN that loops through all the activities. (figure 8).

There are two role methods:

```
ACTIVITY>>frontload
    maxPred = PROJECTSTART.
    for all pred in PREDECESSORS do
        maxPred = max (maxPred, pred.earlyFinish)
    end for
    ACTIVITY.setEarlyStart (maxPred+1)

PLAN>>frontload
    for all act in PLAN.activities do
        act.setEarlyStart (NULL) //set unplanned
    end for
    CURRENTCONTEXT.remap
    while ACTIVITY NOT NULL
        ACTIVITY.frontload
        CurrentCONTEXT.remap
    end while
```

The context object itself is responsible for selecting the objects that are to play the roles. Most selections are trivial. The selection of an activity to play the ACTIVITY role has to be done

systematically. because the algorithm only works for an activity if the earlyFinish-times are known for all its predecessors. This mapping is done in this snippet from the Context remap method:

```
FrontloadContext>>Activity
   select act from activities
      where
         act.earlyStart NOT NULL
      and
         select fromActivity from dependencies
         where fromActivity = act
         and fromActivity.earlyFinish IS NULL
         IS EMPTY
```

The interaction is triggered from outside the context object. This will typically happen somewhere in the user interface:

```
FrontloadContext new (plan).frontload()
```

The plan objects are the Data objects used by the context. This isolates the context from the rest of the system; no other object can play a role in the context. Whatever their state before, all activity objects will have their earlyStart and -Finish times set according to the frontloading algorithm. There will be no other changes; all classes and objects will remain unchanged all through the process.

> *The DCI context tells the truth, the whole truth and nothing but the truth about how an ensemble of objects reify a system operation. The code can be understood, created, tested, and modified as a whole.*

New contexts such as *resource allocation* can be added without increasing the complexity of the system.

> *Why should we package the frontloading algorithm in a context when conventional OO program can do the job? The main reason is that the DCI context concentrates everything about how the system achieves the operation so that its code can be considered in isolation, independently of the program outside it. The isolation leads to better readability because conventional OO distributes code fragments among the classes where it is not easy to discern what is relevant from what is irrelevant. The isolation improves program robustness by making it unlikely that the algorithm code will threaten the integrity of the rest of the program. Conversely, it is equally unlikely that changes in the rest of the code will threaten the integrity of the interaction algorithm*

Draft

4.2 Resource Allocation

There is no simple algorithm for resource allocation so the solution is necessarily pragmatic and user involvement in the programming is essential. Our initial solution is intended as a starting point for discussions with the end users. It leads to the allocations shown in pane ③ in figure 1. We will describe the code for this solution and claim that its simplicity and its weak coupling to other parts of the program makes it feasible for users to understand it and improve it.

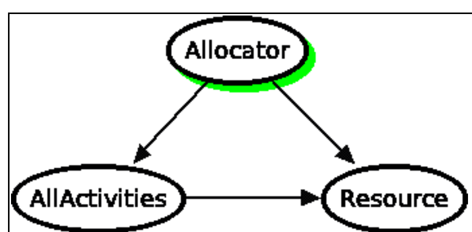


Figure 8: The ResourceAllocationCtx.

Resource allocation can be seen as a dialog between activity objects that seek to complete the project on time, and resource objects that strive for their best possible utilization.[20]

The resource Data model is shown in the allocations table in figure 6.

Figure 8 shows our initial Context for resource allocation with the ACTIVITIES and RESOURCE roles and an ALLOCATOR that coordinates the process.

A "first come first served" strategy is chosen for this initial version of the role methods:

```
ALLOCATOR>>allocateResources
    RESOURCE.reset.
// First decision is to delegate to the activities to control the sequence.
    ALLACTIVITIES.allocateResources.

ALLACTIVITIES>>allocateResources
    for act in self do
        RESOURCE.allocate (act)
    end for

RESOURCE>>allocate (activity)
    tentativeStart = activity.lateStart
    for week = tentativeStart to tentativeStart + activity.duration - 1
        for i = 1 to activity bods
            RESOURCE.allocate (week, activity)
        end for
    end for
    activity.setPlannedStart (tentativeStart).
```

About a dozen lines of code that are well isolated from the more than 1500 lines in the whole program.

5 Related Work

Draft

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[26] provided a way to associate multiple objects through a single operation that engages all of them. Different combinations 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 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[28] 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. In section 3.1, we mentioned that DCI objects can encapsulate different sub-systems. This opens for an interesting study of how DCI can work with other paradigms in the design of large systems. 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-threaded execution model of DCI Contexts.

A particularly promising avenue of research is the teaching of programming to children. An object computer (section 2.3) with the DCI version of object orientation could form a foundation that the children could build upon from their first uncertain steps to a mature mastering of computer programming. There can be a steady progression; there need be no unlearning.

Work remains to further formalize the DCI metamodel. The constraints that Contexts place on object interactions offer the possibility of formal program analyses that were impossible in the past without sacrificing polymorphism.

Draft

7 Conclusion

We started section 3 with a quote[8] saying that an object-oriented program has two structures; a hierarchy of classes and rapidly changing networks of communicating objects. The problem was that “the code won't reveal everything about how a system will work”. Our solution is DCI with static Data projections that specify hierarchies of classes and dynamic Context projections that specify networks of communicating objects that reify use case scenarios and other system operations.

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

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 (section 2.3).
Reasoning.	We work with a DCI program in the orthogonal Data and Context projections. This conceptually simple, yet effective representation enables a developer to reason about one dimension at a time.
Readability.	Readable code is code that is cleanly partitioned and that clearly exhibits the system design.[24] DCI’s orthogonal projections provides such independent partitions.
Reuse.	There are two opportunities for reuse with DCI. One is that the classes in the Data projection 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 be supported in effective development environments. Interesting developments are happening with adapting a number of languages such as C#, Ruby, and C++ 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."*[22]

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 stakeholders can understand critical parts of the program and that some can even write code themselves. 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 stakeholders apply it to their various tasks. Indeed, we claim that DCI is so powerful that it can form a conceptual foundation for expert programmers and that 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, transform, and communicate, the primitives of computing that are captured in the DCI paradigm (section 3.4). We expect that this addition of a new dimension to programming will have a profound influence on program architecture and programming languages. So simple that everybody can understand it, so universal that it can form the foundation for computing in business and education.*

Draft

8 Acknowledgements

Matthew Browne
Gertrud Bjørnvig
Morten Jacobsen
Herman Peeren
Branislav V. Selic
Rune Funch Søltoft
Risto Valimaki
Bruce Horn
<TBD: add more>

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 Reenskaug,T: *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	Parnas, D.L., Clements, P.C.: *A Rational Design Process: How and Why to Fake It;* IEE Transactions on Software Engineering, SE-12, February 1986; pp 251 - 257.
19	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
20	Reenskaug, T.; "*Prokon/Plan-A Modelling Tool for Project Planning and Control*", in Proc. IFIP Congress, 1977, pp.717-721.

Draft

21	Reenskaug, T.: *The original MVC reports*. Xerox PARC 1978; [Web page] http://www.duo.uio.no/sok/work.html?WORKID=52648
22	Reenskaug, T.: *User-Oriented Descriptions of Smalltalk Systems*; Byte Magazine, August 1981.(The special issue on Smalltalk.)
23	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 .[WEB PAGE]PDF
24	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
25	Reenskaug T.; [WEB PAGE] http://folk.uio.no/trygver/2009/commonsense.pdf
26	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.
27	*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
28	Ungar, Smith: *Self, the power of simplicity.* http://labs.oracle.com/self/papers/self-power.html, 1987.
29	http://en.wikipedia.org/wiki/Main_Page
30	Rickard Öberg: What is Qi4j? http://qi4j.org/
31	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.

Draft

APPENDIX 1: PROKON ACTIVITY PLANNING: ARCHITECTURE

Section 4 gave an introduction to our Prokon example program.

Having established an object oriented mental model, we are now ready to design the program. The guiding principle is separation of concerns; we separate the code into largely independent parts that can be reasoned about and tested in isolation. The program architecture builds on the complementary paradigms MVC and DCI. MVC (appendix 2) bridges the gap between system data and a form that can readily be assimilated by the user's mind. 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 mindset.

At the top level, the Prokon program is separated into two main parts according to MVC: *Model*, a representation of domain information, and *User Interface* that bridges the gap between the human brain and the computer (appendix 2). Each MVC part is subdivided according to DCI: *Data-Model* represents the domain information and *Context-Model* with the operations on the model. Similarly, *Data-UI* with the visible user interface and *Context-UI* with update operations and complex user interactions. Table 2 distributes the Prokon classes among the four projections.

Table 2: Projections with their Classes.

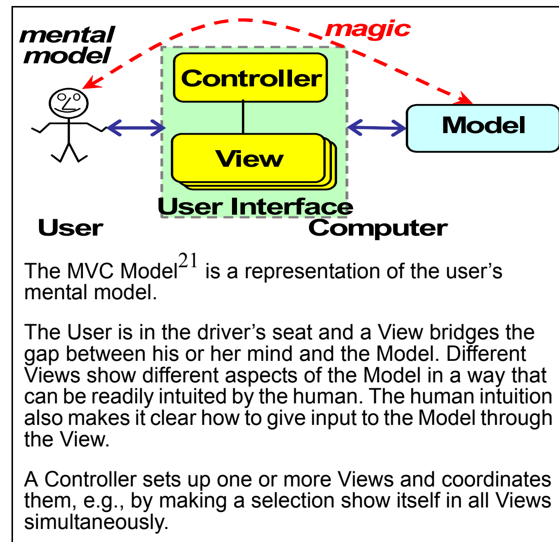
	Model	**User Interface**
Data	Model Activity Dependency Resource	Controller DependencyView DependencyLine ActivitySymbol GanttView ResourceView ActivityTextView BlurbView
Context with Interactions	FrontloadCtx BackloadCtx ResourceAllocationCtx	AddActivityCtx AddDependencyCtx DependencyDisplayCtx GanttDisplayCtx ResourceDisplayCtx

Draft

The 20 classes are organized in four independent projections. each with a clearly understood responsibility. Each class can be independently written, read, tested and maintained. The code is readable.

APPENDIX 2: MVC - THE MODEL, VIEW, CONTROLLER PARADIGM

Figure 9: MVC.



An outcome of research at Xerox PARC on user interfaces was the MVC (Model-View-Controller) paradigm[21] (figure 9). The paradigm sustains Douglas Engelbart's vision of computer augmentation by which computers extend the human intellect and improves 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 user on the other hand. The objects of a model should therefore represent identifiable parts of the problem."*[21]

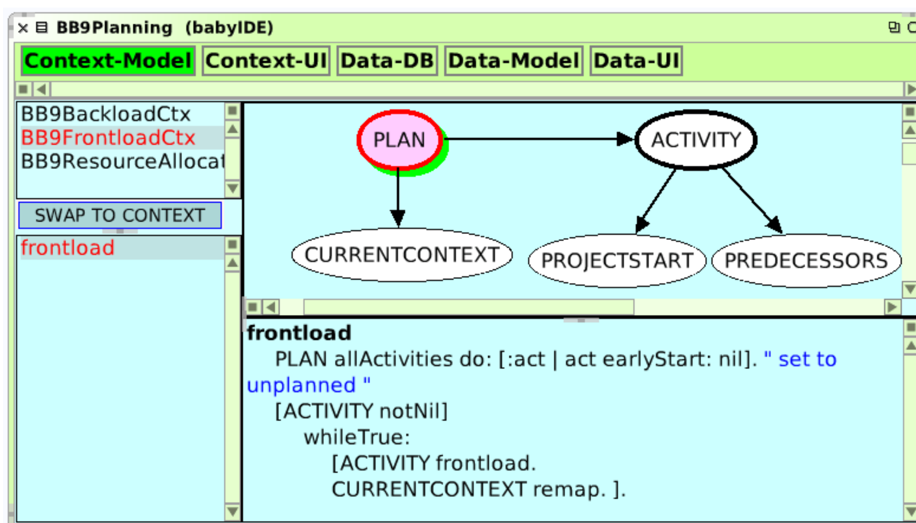
The MVC Model is simply implemented as classes that realize the tables of figure 6 with their container: Model, Activity, Dependency, and Resource.

In figure 1, We see 5 views marked ① through ⑦ in figure 1. They are implemented as 5 View objects. A Controller object creates them and makes the selection of an activity simultaneously visible in all Views.

Draft

APPENDIX 3: THE SMALLTALK CODE

Figure 10: BabyIDE, an IDE for coding DCI programs in Squeak.



The Prokon planning program has been implemented in a DCI extension of Squeak using the BabyIDE tool (figure 10). We see projection flaps along the top; Context-Model is chosen. The contexts are listed top-left, the frontload context is chosen. Roles with their connectors are top-right. The selected role is PLAN, its role methods are list bottom-left and its code is bottom-right.

More details:

- The BabyIDE tool is part of the Squeak 3.10 image at http://fulloo.info/Downloads/BabyIDE.zip
- An HTML listing of the Prokon code is at http://fulloo.info/Examples/SqueakExamples/BB9Planning/readableVersion.html
- The BabyIDE packages can be loaded into a Squeak 4.5 image with the SqueakMap package loader, package BabyIdeAllInOne.

We will here include excerpts of the source code relevant for the frontloading operation.

<TBD. The Prokon program is being updated.>

App 3.1: The Frontloading Operation

projection: Data-Model

```
Object subclass: #Activity
    instanceVariableNames: 'name duration earlyStart lateFinish plannedStart resourceRequirement'
    earlyFinish
        ^earlyStart ifNil: [nil] ifNotNil: [earlyStart + duration - 1]

Object subclass: #Dependency
    instanceVariableNames: 'fromActivity toActivity''

Object subclass: #Model
    instanceVariableNames: 'activities activityPositions dependencies projectStart projectFinish resource'
    initialize
        super initialize.
        activities := OrderedCollection new.
        activityPositions := IdentityDictionary new.
        dependencies := IdentitySet new.
            resource := Resource new.
            projectStart := projectFinish := 0.
    allActivities
        " Return a copy, the activities inst.var. is private to the model object. "
        ^OrderedCollection newFrom: activities
    dependenciesFromActivity: act
        ^ dependencies select: [:dep | dep fromActivity == act]
    dependenciesToActivity: act
        ^ dependencies select: [:dep | dep toActivity == act]
    predecessorsOf: act
        ^ (self dependenciesToActivity: act) collect: [:dep | dep fromActivity]
    recomputeModel
        BackloadCtx new backload: self.
        FrontloadCtx new frontload: self.
        ResourceAllocationCtx new allocateResources: self.
```

Draft

projection: Context-Model

```
Context subclass: #FrontloadCtx
    instanceVariableNames: 'activity model'
    " Find earliest time period for each activity. "
    frontload: mod
        model := mod.
        self triggerInteractionFrom: #MODEL with: #frontload.



    remap
        " Need to find an activity that is ready for planning first since it is used by two role mappings. "
        activity := model allActivities
```

```
        detect:
            [:act |
            act earlyStart isNil and:
                [(model predecessorsOf: act) noneSatisfy: [:pred | pred earlyFinish isNil]]]
        ifNone: [nil].
    super remap.
```

ACTIVITY

```
    ^activity
```

CONTEXT

```
    ^self
```

MODEL

```
    ^model
```

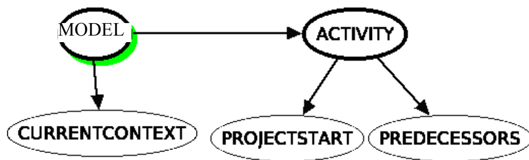
PREDECESSORS

```
    ^activity
        ifNil: [Array new]
        ifNotNil: [model predecessorsOf: activity]
```

PROJECTSTART

```
    ^model projectStart
```

Interaction: FrontloadCtx



roleMethods: MODEL

frontload

```
        MODEL allActivities do: [:act | act earlyStart: nil]. " set to unplanned "
        [CURRENTCONTEXT remap. ACTIVITY notNil] whileTrue: [ACTIVITY frontload].
```

roleMethods: ACTIVITY

frontload

```
     maxPred |
    maxPred := PREDECESSORS detectMax: [:pred | pred earlyFinish].
    maxPred
        ifNil: [ACTIVITY earlyStart: PROJECTSTART.]
        ifNotNil: [ACTIVITY earlyStart: maxPred earlyFinish + 1].
```

Draft