The Common Sense of Object Oriented Programming

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

Abstract. The essence of object orientation is that networks of collaborating objects work together to achieve a common goal. Surely, the common sense of object oriented programming should reflect this essence with code that specifies how the objects collaborate. Our industry has, unfortunately, chosen differently and code is commonly written in terms of classes. A class tells us everything about the properties of the individual objects that are its instances. It does not tell us how these instances work together to achieve the system behavior.

The result is that our code does not reveal everything about how a system will work. This is clearly not satisfactory, and we need a new paradigm as the foundation for more expressive code. We propose that DCI (Data-Context-Interaction) is such a paradigm. DCI separates a program into different perspectives where each perspective focuses on certain system properties. Code in the Data perspective specifies the representation of stand-alone objects. Code in the Context perspective specifies runtime networks of interconnected objects. Code in the Interaction perspective specifies how the networked objects collaborate to achieve the system behavior.

BabyIDE is an interactive development environment that supports the DCI paradigm with specialized browsers for each perspective. These browsers are placed in overlays within a common window so that the programmer can switch quickly between them.

DCI with BabyIDE marks a new departure for object oriented programming technology. It opens up a vista of new and interesting product opportunities and research challenges. Some of these challenges and opportunities are touched upon in a final chapter of this report.

Keywords. Object-oriented programming – Object oriented methods – Static object structures – Dynamic object structures – late binding – collaboration – role model – IDE – BabyUML – BabyIDE

The BabyIDE home page is at
http://heim.ifi.uio.no/~trygver/themes/babyide

The Common Sense of Object Oriented Programming
Table of Contents

1 Introduction and summary

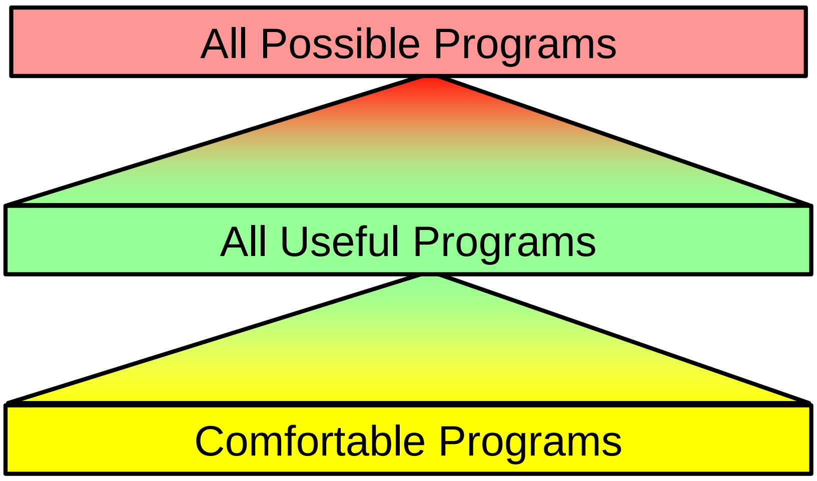
In his 1991 Turing Award Lecture, Tony Hoare succinctly stated the value of readable code [Hoare-81].

"There are two ways of constructing a software design:
One way is to make it so simple that there are obviously no deficiencies
and the other is to make it so complicated that there are no obvious deficiencies.

There is no doubt in my mind that the first way is the best way. I can only trust my code when it so simple that other people can read it and confirm that there are obviously no deficiencies.[1]

Figure 1 can serve as an illustration. The number of different bit patterns that a given computer can execute as a program is enormous; this is symbolized by the upper rectangle in the figure. Most of these programs will either come to an early stop or run forever. The middle rectangle symbolizes the very small subset containing all useful programs. These programs may pass all their tests and some of them might even be bug free. They include cool programs, smart programs, '*guess what it does*' programs, obscure programs. The bottom rectangle in the figure symbolizes the very small subset of the useful programs that are so simple that there are obviously no deficiencies. These programs are clearly readable and I call them comfortable programs because I like to have them around.

Figure 1: All comfortable programs are but a small subset of all useful programs.



Very few object oriented programs are really readable. Polymorphism is a common reason; different objects treat the same incoming message differently depending on their class. Consider the simple statement: *foo delete*. My Squeak image has 46 methods called *delete*; most of them complex and many of them with many side effects. Which one will be called? I could guess what is intended, but I have to identify the class of *foo* to be certain. So I digress into a new tangle of code to find out how *foo* gets its value and the problem repeats itself ad nauseam. As Adele Goldberg once said: "everything happens somewhere else".

The deplorable state of affairs has been clearly expressed in the *Design Patterns* book:

1. I expand on the importance of readability in the short article *The Case for Readable Code* [Readable]

An object-oriented program's runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. The runtime structure consists of rapidly changing networks of communicating objects.[GOF-95] p. 22.

and

…, it's clear that code won't reveal everything about how a system will work. [GOF-95] p. 23

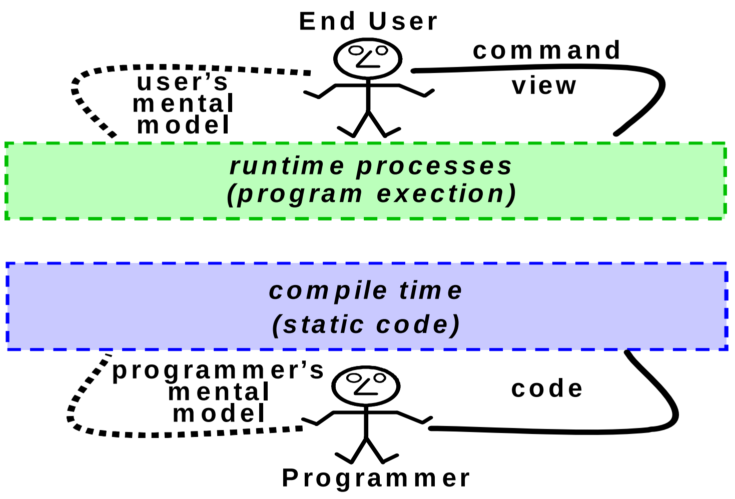
How can we ever hope to build comfortable programs if their code don't reveal how they work? Are we doomed to live with programs that are so complicated that there are no obvious deficiencies? Are we doomed to rely on testing to get a semblance of quality into our programs?

In 1968, Edsger W.Dijkstra wrote a letter to the editor of the *Comm. ACM.* that came to be titled: "*Go To Statement Considered Harmful*". His arguments are relevant to our search for ways and means for readable code. He started the letter with two remarks. We will discuss them one at the time.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to effectuate the desired effect, it is this process that in its dynamic behaviour has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.[Dijkstra-68]

The situation as I see it is illustrated in figure 2. A program generates value by the way its processes support the end user in performing his or her tasks. The end user controls the processes by giving appropriate commands and understands them through his or her mental model. A program quality measure is the extent to which the model exposed through the program interface actually conforms to the user's mental model. We discuss means for satisfying this quality measure in section 5. 1: *MVC: the Model-View-Controller paradigm*.

Figure 2: A program, its creation and its use.


End User
command
view
user's mental model
runtime processes (program exection)
compile time (static code)
programmer's mental model
code
Programmer

A programmer writes the code that the machine transforms into the end user's runtime processes. But we have seen that …, *it's clear that code won't reveal everything about how a system will work*". There is a chasm between the dynamic and the static, runtime and the compile time, and it is hard for the programmer to build a mental model that includes the runtime processes.

We need a new way of writing programs to make them consistently readable. Dijkstra's second remark suggests where to look for it:

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost best to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.[Dijkstra-68]

The execution of a System Operation is triggered by a message from outside the system. The message is received by one of the system's objects that triggers an avalanche of messages that flow through the system. The structure of sender/receiver objects form a tree where branches are sprouting and retracing. The code that controls this flow constitutes the system behavior. This code is fragmented and distributed among the system classes. Dijkstra's second remark suggests that this is not acceptable; we need a programming paradigm that concentrates this behavior code into structures that reflect the runtime structures.

The DCI paradigm is such a paradigm. Programs that are coded according to this paradigm answer the critical questions that need to be answered for every System Operation:

1. What is the network of communicating objects?
2. How are the objects interlinked?
3. How do they interact?

Two earlier articles describe the background for the DCI paradigm. *The BabyUML discipline of programming*[BabyUML-06] explored new programming elements that have lead up to the current solution. The topology of the runtime "networks of communicating objects" mentioned in the above GOF quotation is coded as a conformant network of interconnected Roles. These Roles are bound to classes at compile time and to communicating objects at runtime, thus ensuring conformance between the compile time Roles and the runtime objects.

The fundamental concepts of static networks of interconnected Roles is taken further in *Programming with Roles and Classes: the BabyUML Approach*[BabyUML-07]. This book chapter describes the new DCI paradigm for specifying system state and behavior.

In the DCI paradigm, the programmer's mental model of a program is extended to let the programmer work with the code as seen in several perspectives. The essential ones are:

Data:	The computer representation of the user's mental information model.
Context:	Contexts implement System Operations and specify the networks of interconnected objects that realize them.
Interaction:	Specification of how objects behave in the context of a System Operation.

The DCI paradigm gives maximum leverage for the programmer if it is applied through a suitable development environment. *BabyIDE* is such an environment that has been developed for programming in Squeak.

This report is a documentation of the DCI paradigm and its application through the BabyIDE programming environment. This report is organized as follows:

Section 1: *Introduction and summary*. This section.

Section 2: *Roles and Interactions: Visualizing a simple process*. A visualization of networks of communicating objects. There are two System Operations: The *ShapesAnimation and Arrows-Animation*.

The *ShapesAnimation* visualizes objects that are created and removed in a universe of objects. The objects are shown as shapes that appear and disappear within a window on the screen. Some shapes are stars, some are circles. This illustrates that the class of a shape is unimportant. Two screenshots are shown in figure 3 on page 9.

The *ArrowsAnimation* visualizes object communication as arrows that grow from a sender object to a receiver object. This continues until four messages have connected five objects. The display is then cleared and the sequence is repeated. The point of this animation is that while each sequence of messages appears to be the same at every repetition, the actual objects are different. Two screenshots are shown in figure 4 on page 10.

The section ends with a discussion about how we can make a static description of such repeating behavior in the face of multiple classes and object identities. We find concepts and terms we can use to describe what we see in the animations independent of programming language, tools, etc.The solution is the DCI paradigm where we focus on the common topology of networks of communicating objects. The nodes are called *Roles*, the edges are called *Connectors*. The Roles are bound to objects at runtime. Runtime object communication is specified as compile time Role Interaction.

Section 3: *BabyIDE1, an Environment for true Object-Oriented Programming*. The *BabyIDE* is an interactive development environment that is built on the concepts of Classes, Contexts, and Roles. The *BabyIDE* idea is that a programmer needs to see a program in different perspectives. The perspectives are organized in overlays, and the programmer flips between perspectives at the press of a button. This section describes *BabyIDE1* with its perspectives realized as class and Interaction browsers.

Section 4: *BB5Bank: A Simple Money Transfer Program According to DCI*. This section describes a small and simple program that exemplifies the essential properties of the DCI paradigm.

Section 5: *MVC and DCI - Two paradigms for readable code*. The first paradigm, the Model-View-Controller is included here for completeness. The MVC paradigm separates the parts of a program that are responsible for representing the user's mental information model and the parts that are responsible for providing the illusion that the user is interacting directly with this mental model.

The goal of DCI is to minimize any gap that might exist between the programmer's mental model of a code and the processes that are actually serving the user at runtime. In particular, it concretizes how the system realizes System Operations as networks of communicating objects. Traits[Schärli] is the keystone in the bridge that spans the gap between the code that describes how the system performs a System Operation and the classes that define the participating objects and their methods.

Section 6: *The BB2Shapes Process Visualization Program* documents the program that drives the *BB2Shapes* visualization. The BB2Shapes program is documented in four perspectives: Data, Context, Interaction, and Environment. Environment objects are objects that trigger the System Operations realized by the Contexts.

Section 7: *BB4bPlan: An Activity Network Planning Program with DCI*. An example of DCI programming that illustrates how to utilize DCI as an organizing principle for the Model element in the MVC paradigm.

Section 8: *BB4aPlan: A Conventional Activity Network Planning Program*: The above DCI implementation is compared to a regular OO implementation of the same example. We see that the code is almost the same in the two versions, but that the DCI version disentangles the tangled code of the conventional implementation. We found that in this very simple example, the conventional version of its Model class was burdened with almost 50% more lines of code and 100% more instance variables compared to the DCI version of the same class.

Section 9: *Support for Programming with Roles in Squeak*. It has been necessary with certain modifications and extensions to Squeak to enable it to support the DCI paradigm.

Section 10: *The BabyIDE1 implementation*. The less said about it, the better.

Section 11: *Conclusion*. The BabyUML project has now reached its goal which is to make the runtime behavior of a system concrete and visible in the code. Demonstration examples have been implemented and acted as proof of concept. The results are very promising and BabyIDE1 is now ready to be released for alpha testing.

Section 12: *Further work*. What remains to be done is the formidable task to evolve from the current alpha version to a stable and usable tool. This section also suggests some jobs that need to be done and some research issues that it can be profitable to study.

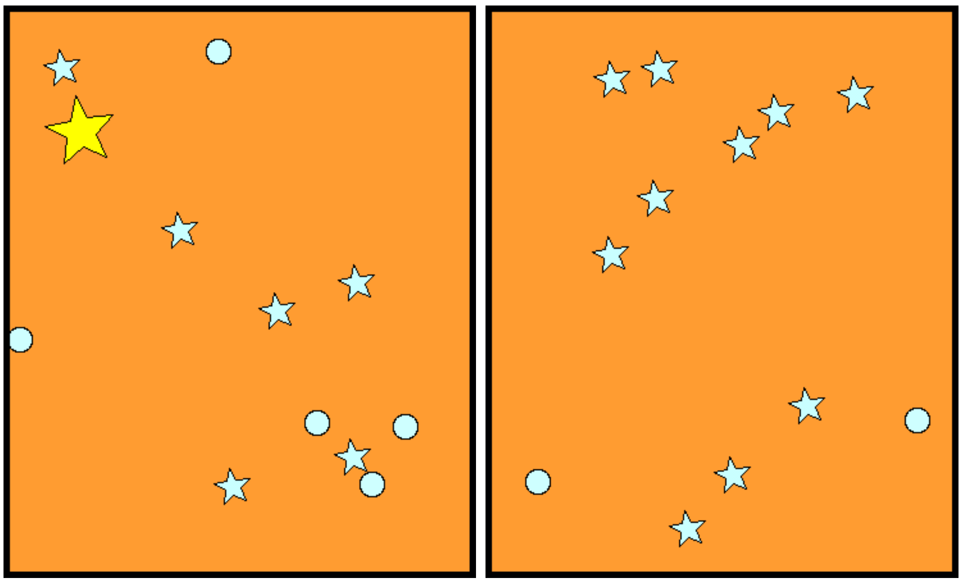
2 Roles and Interactions: Visualizing a simple process

The quote from the Design Patterns book on page 5 says that the code *doesn't tell us how a system will work*. This is clearly a problem that cries for a solution. As a first step towards getting to grips with it, I have written two animations that visualize "*rapidly changing networks of communicating objects*".

The animation program, *BB2Shapes*, shows a system as a colored field with objects as shapes within this field. It has two use cases that visualize the dynamic nature of system behavior, *ShapesAnimation* and *ArrowsAnimation*. These animations are designed to pose a challenge: *Find a way to describe how the visualized system works*. We find the basic class based paradigm inadequate, and introduce the new DCI paradigm; the *Data-Context-Interaction* paradigm.

The *ShapesAnimation* visualizes how objects come and go during an execution by making shapes appear and disappear randomly within the system field. Figure 3 shows two snapshots taken during an execution. The objects can be instances of different classes; this is visualized by stars and circles. We strongly suggest that you actually watch the animation because there is a vast difference between reading about it and actually observing it. Click animate-shapes.mov [1] to watch the movie or run the actual Squeak program[2]

Figure 3: .Two snapshots from the *ShapesAnimation*.

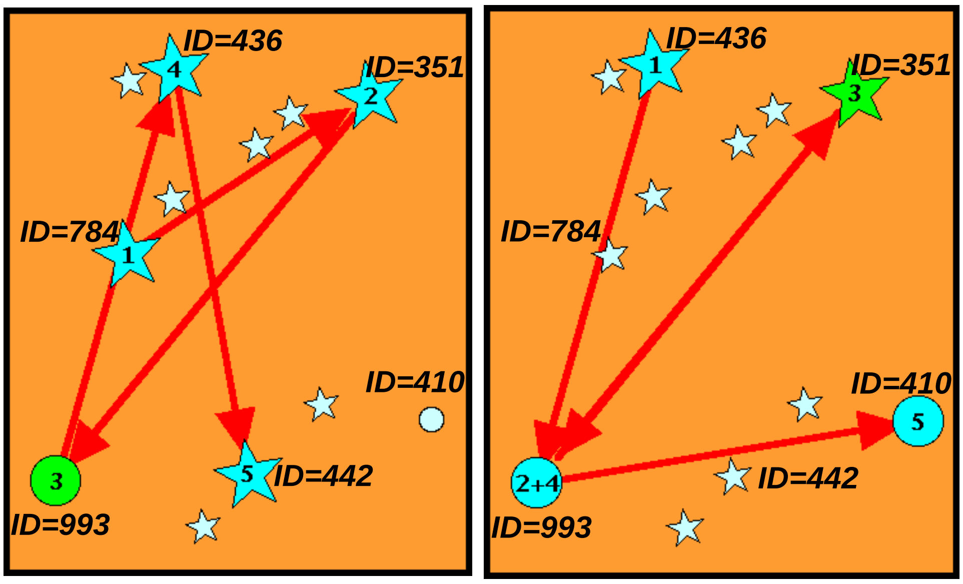


A more challenging example is the *ArrowsAnimation*. This animation visualizes a runtime structure that "*consists of rapidly changing networks of communicating objects*". Figure 4 shows two snapshots. They visualize that messages are sent from object to object by showing arrows that grow from sender to receiver objects. Both snapshots show the system after a train of arrows has connected five objects:

1. http://heim.ifi.uio.no/~trygver/assets/animate-shapes.mov
   http://heim.ifi.uio.no/~trygver/assets/animate-shapes.mpeg
2. Click *Downloads* in the DCI Home Page:
   http://heim.ifi.uio.no/~trygver/themes/babyide/babyide-index.html

1. an object has been selected, enlarged, and annotated with the number 1,
2. an object has been selected, enlarged, and annotated with the number 2.
3. an arrow has been drawn from object 1 to object 2,
4. …and so on up to the last arrow pointing to object 5.

Figure 4: Two snapshots from the *ArrowsAnimation*.



Click animate-arrows.mov [1] to see the *ArrowsAnimation*. Again, static snapshots do not communicate the dynamic nature of the program execution and we strongly suggest that you take the trouble to actually see the video or run the program[2].

The stars and circles are instances of different classes and they appear at random in both animations. Note that the classes of the objects do not appear to matter. In figure 4 for example, object 5 is a star in the left snapshot and a circle in the right one. The classes are irrelevant and do not reveal everything about how the system works. The class based object paradigm is inadequate and we need to extend it to be able to describe a process involving networks of communicating objects.

The *ArrowsAnimation* visualizes the dichotomy between the static system state and the dynamic system behavior. The system state is the same in the two snapshots in figure 4; both exhibit the same objects in the same positions. Dynamically, the process of selecting objects and drawing arrows between them repeats itself again and again, but the process selects different objects every time around the loop.

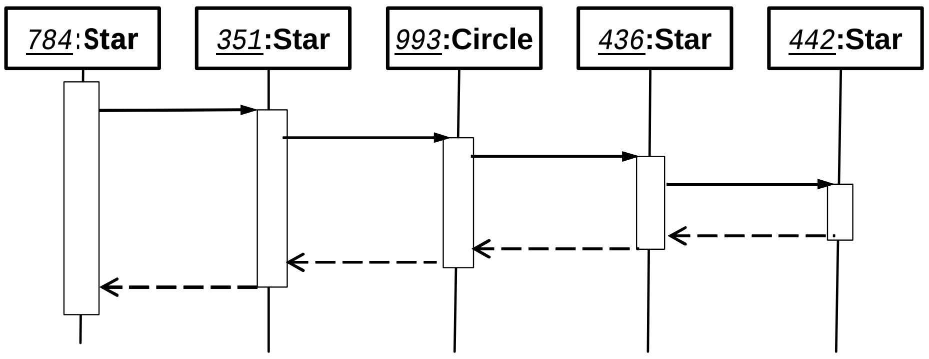
We are keeping things simple and are only considering sequential execution. The execution of a System Operation is triggered by a message to some object. This activates a method that sends a message to the same or to another object. Every message is a part of the sequence of object Interactions that realize the System Operation We can make a *trace* of the execution by observing every message; its sender, its receiver, and its name (message selector). The trace describes a dynamic network of communicating objects. Every visited object is a participant in the process. Every sender/receiver pair tells us that there is a link between them. The link may be per-

1. http://heim.ifi.uio.no/~trygver/assets/animate-arrows.mov
http://heim.ifi.uio.no/~trygver/assets/animate-arrows.mpeg
2. Click *Downloads* in the DCI Home Page:
http://heim.ifi.uio.no/~trygver/themes/babyide/babyide-index.html

manent, or it may only exist for the duration of a particular message transmission. The dynamic network as a whole is likely to be unique to that particular execution and may not exist in its entirety at any time. The network can disappear at the termination of the execution.

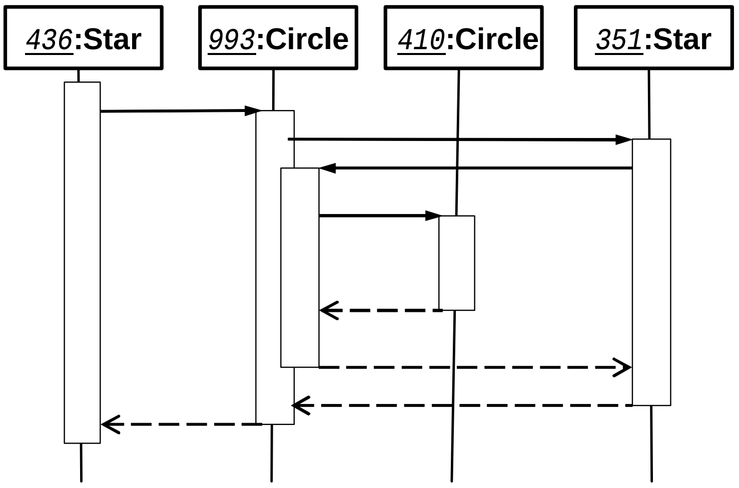
The UML[UML] *sequence diagram* in figure 5 models the process that resulted in the left snapshot of figure 4. A vertical line with a box on top is called a *lifeline*; it represents the history of one of the communicating objects. The box contains the name of the participating object with the format <name:class>. The objects are here named by their object IDs. The unbroken arrows denote message transmissions; the narrow, vertical rectangles denote method executions; the dashed arrows denote method returns.

Figure 5: A Sequence diagram describing the process visualized in the left snapshot of figure 4.



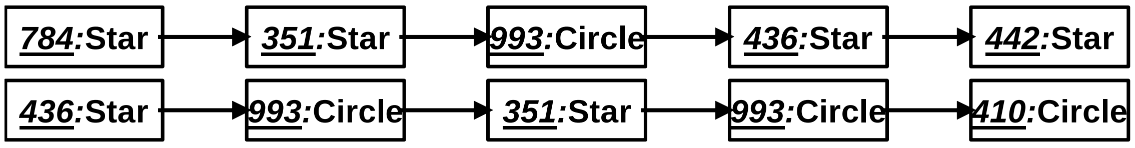
The next sequence diagram (figure 6) models the process that resulted in the right snapshot of figure 4. Some objects are reused from the previous model, but they occupy different positions in the sequence. An interesting feature of the right snapshot is that the shapes marked ‘2’ and ‘4’ are actually the same object, namely the object with ID = 993.

Figure 6: A sequence diagram describing the process visualized in the right snapshot in figure 4.



We revisit these two execution traces, this time recording the objects visited and the observed sender/receiver pairs. We find the two communication networks shown in figure 7.

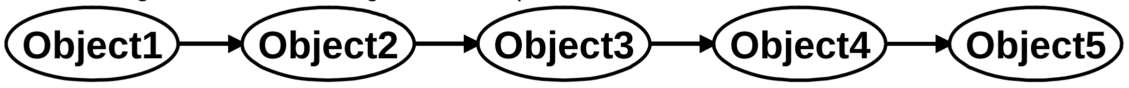
Figure 7: The communication networks of the Interactions shown in figure 5 and figure 6.



$$\begin{tikzcd}
\boxed{784:Star} \arrow[r] & \boxed{351:Star} \arrow[r] & \boxed{993:Circle} \arrow[r] & \boxed{436:Star} \arrow[r] & \boxed{442:Star} \\
\boxed{436:Star} \arrow[r] & \boxed{993:Circle} \arrow[r] & \boxed{351:Star} \arrow[r] & \boxed{993:Circle} \arrow[r] & \boxed{410:Circle}
\end{tikzcd}$$

We observe that the topologies of the two communication networks are the same even if the actual objects in their nodes differ. We observe the actual animation and see that all networks have the same topology. We have drawn this common topology in figure 8.

Figure 8: Context Diagram for the process visualized in the *ArrowsAnimation.*



The nodes in the new diagram are called *Roles* and the edges *Connectors*. The diagram represents the *Context* of the Interaction, where a Role is shown as an ellipse and a Connector as an arrow.[1]

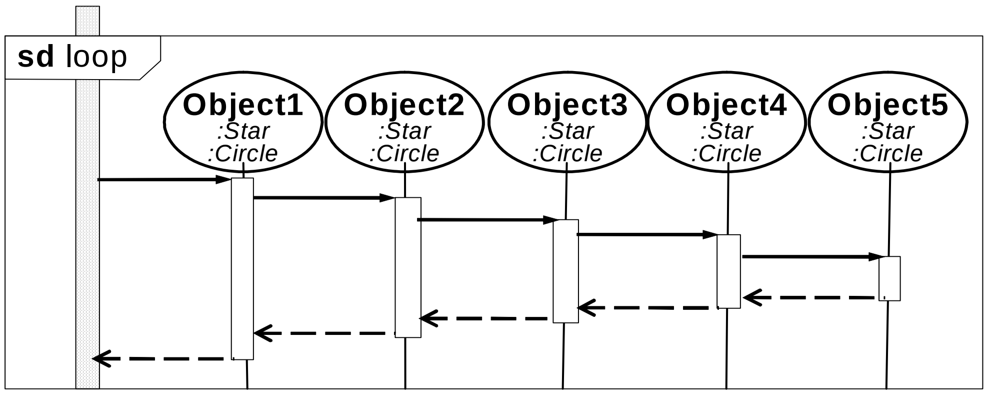
The transition from the rapidly changing networks of communicating objects to static Contexts with their interconnected Roles is very important. Remember Dijkstra's plea for shortening the gap between the static program and the dynamic processes. (page 6). We have here established the compile time, static concepts of Context and Role that closely mirror the ephemeral networks of communicating objects.

The concept of a Role is central to the DCI paradigm. Webster [Webster-08] gives a number of synonyms for the word **role**; all of them applicable to our use of the word. The synonyms are *capacity, function, job, part, place, position, purpose, task, work*:

- *capacity*: Objects playing a Role must have the features needed to play the Role.
- *function*: A Role represents the functionality that objects need to be able to play the Role.
- *job, task, work*: Work is done to perform a System Operation. The responsibility for performing this work is delegated to the communicating objects. A Role represents a responsibility that is delegated to the object that happens to be playing it.
- *part*: An objects plays a Role in a network of communicating objects as an actor plays a Role in a comedy by Shakespeare.
- *place, position*: A Role represents a position in the network of communicating objects.

The description of system behavior is likewise mirrored from the runtime snapshots of figure 5 and figure 6 to the static sequence diagram of figure 9. There is one lifeline for each Role. It describes part of the life of any object that plays the Role at runtime. The methods executed by these objects are shown as narrow, vertical rectangles in the diagram.

Figure 9: Sequence diagram for the ArrowsAnimation.



A Role can be played by instances of different classes, here exemplified by stars and circles. We call these classes *Role Player Classes* and have added their names in the diagram according to the UML notation. A Role Player Class must include the features needed for its instances to play the given Role.

1. Such networks are called role models in [OOram] and Collaborations in [UML].

One important question remains. How do we know that the method executed by an object playing the *Object1*[1] Role will actually send the appropriate message to an object playing the *Object2* Role? Polymorphism leaves the question unanswered.

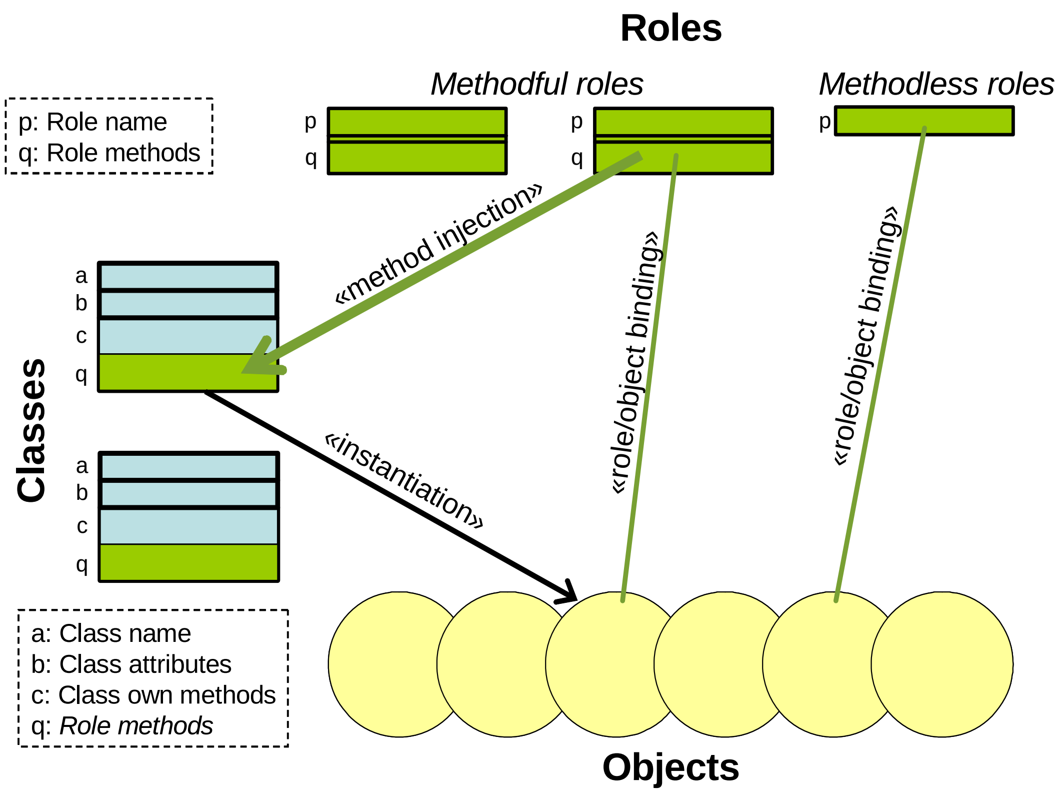
We discussed the problem statement, *foo delete*, on page 4. The DCI paradigm lets us make the statement more explicit. We can write something like *Object2 delete* where the Role name *Object2* is a reference to the runtime object that happens to be playing the *Object2* Role.

A stronger solution is to suspend polymorphism for the methods that are essential for the integrity of the process. We define the methods shown as narrow, vertical rectangles in figure 9 as features of the Roles and force the Role Player Classes to share these methods. The method selector *delete* now causes the execution of a *delete* method that is a feature of the *Object2* Role and is shared by both the *Star* and *Circle* classes. Such methods are called *Role Methods*; they are implemented by Traits and are discussed in section 5. 2 on page 27.

Figure 10 is based on an idea from Jim Coplien and illustrates the essential DCI concepts. The classes are drawn as rectangles with compartments for name, attributes, and methods. There is an additional compartment for the methods that are injected into the class from the Roles.

All Roles are bound to objects at runtime. These bound objects must offer an interface that is a feature of the Role. In addition, some Roles are defined with methods; these methods are injected into the appropriate classes and are thus shared by all objects playing the Role.

Figure 10: The relationships between Roles, Classes and Objects according to James Coplien.



1. We will occasionally underline Role names in running text to enhance readability. This is not part of the current DCI notation and underlining will not be added systematically.

We have arrived at the DCI paradigm for object oriented programming:

- The **D** for Data, e.g., the Star and Circle classes.
- The **C** for Context, e.g., a class that implements System Operations and defines the network topology (an example is shown in figure 8). A Context has methods that bind Roles to objects at runtime.
- The **I** for Interaction are the methods driving the object communication as exemplified in figure 9.

A reader of DCI code can answer the three essential questions: *What are the objects, how are they interlinked, and what do they do*. The code does reveal how the system will work.

In theory, practice is straight forward. All we need to do is to create an interactive development environment (IDE) that supports the notions of Context, Role, Role Methods, and Role binding. A first example of such an environment is the *BabyIDE*; it is described in section 3.

3 BabyIDE1, an Environment for true Object-Oriented Programming

The *BabyIDE1* is a tool designed for working with a program as seen in different perspectives according to the DCI paradigm. Each perspective tells part of the story; all perspectives taken together tell the whole story. The BabyIDE supports the DCI paradigm by providing specialized browsers for three essential perspectives:

Data A *BB1ClassBrowser* for working with the static parts of the domain classes. These static parts specify the state of the system through their own and derived attributes and their associations with other classes. The Data classes can also specify local behavior, that is behavior realized within the encapsulation of its instances.

The dynamic parts of the Data classes are not edited here. They are the methods that specify how instances interact with other instances when realizing System Operations. These Interaction methods are edited in the Interaction perspective and injected from there.

Context A *BB1ClassBrowser* for browsing the Context classes. There is currently one Context class for each System Operation, this class specifies the network topology with its Roles, their Connectors, and the methods that bind Roles to objects during an execution of the operation. (The one-to-one relationship between Context and System Operation will be relaxed in future versions of BabyIDE/DCI).

Interaction A *BB1InteractionBrowser* is based on the Context Diagram.It includes panes for viewing and editing the Context Diagram and for selecting and editing the methods that drive the peer to peer communication when the system performs a given System Operation. This perspective supports editing Role Methods, i.e., methods that are constrained to be common to all Role Player Classes implementing a given Role.

Other perspectives can be added as needed. Each additional perspective is edited in a *BB1ClassBrowser* that shows the classes organized in a sub-category in the Squeak SystemOrganization. Examples are perspectives for environment classes, i.e. classes that include triggers for the execution of System Operations. Other examples are perspectives for the elements of the MVC paradigm.

BabyIDE has been implemented in Squeak[Squeak], a dialect of Smalltalk. Its first implementation is called *BabyIDE1* and is available for download[1].

Figure 11 shows a snapshot of the BabyIDE1 programming environment. The top row includes two buttons and a title. The left button marked X is for closing the window. The second button yields a menu for managing the window as such. The title is in two parts: *BabyIDE1*, the name of this IDE, and the name of the current application. (*BB2Shapes* in this case.)

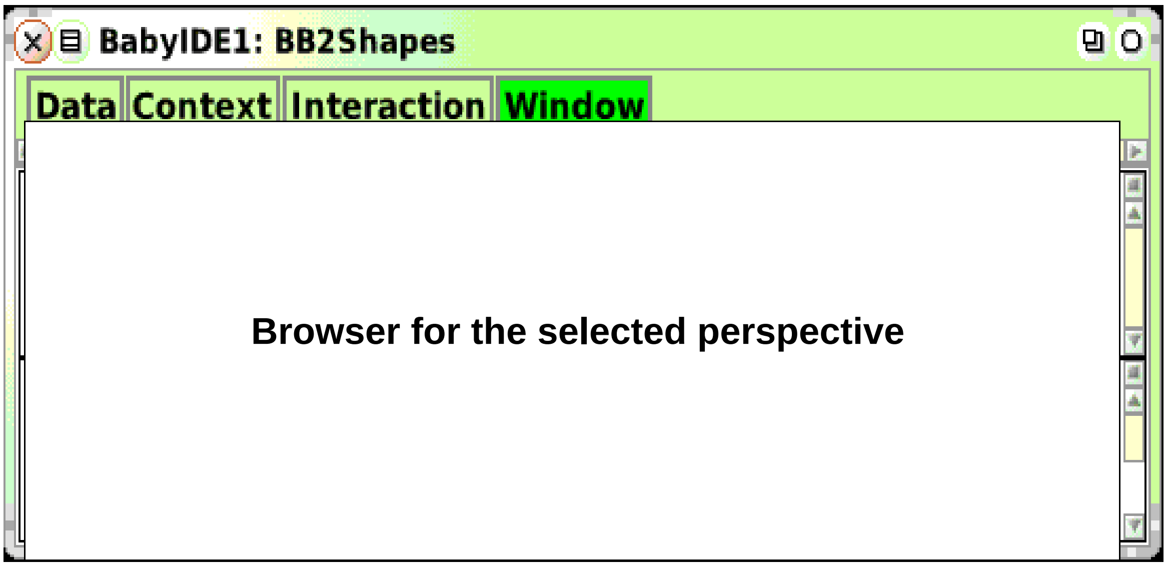
In the second row, we see a strip of four buttons for selecting the required perspective. Each perspective is handled by a specialized browser. These browsers are arranged in independent overlays within the window so that one of them is visible at the time. The programmer switches between perspectives by clicking the corresponding button. The *Data* and *Context* buttons activate a *BB1ClassBrowser* for working with the corresponding classes. The *Interaction* button activates the *BB1InteractionBrowser.* Additional perspectives are edited in the *BB1ClassBrowser.*

1. Click *Downloads* in the DCI Home Page:
   http://heim.ifi.uio.no/~trygver/themes/babyide/babyide-index.html

There is a yellow menu (right-button menu) in the background of the second row. Its commands are

set application	Change the application that is the subject of this IDE.[1]
new perspective	Add another perspective. (The perspective name corresponds to the second part of the class category name in the Squeak class library).
fileOut this App	File out this application as a regular *.st* file for later loading into other images.[2]

Figure 11: BabyIDE development environment.



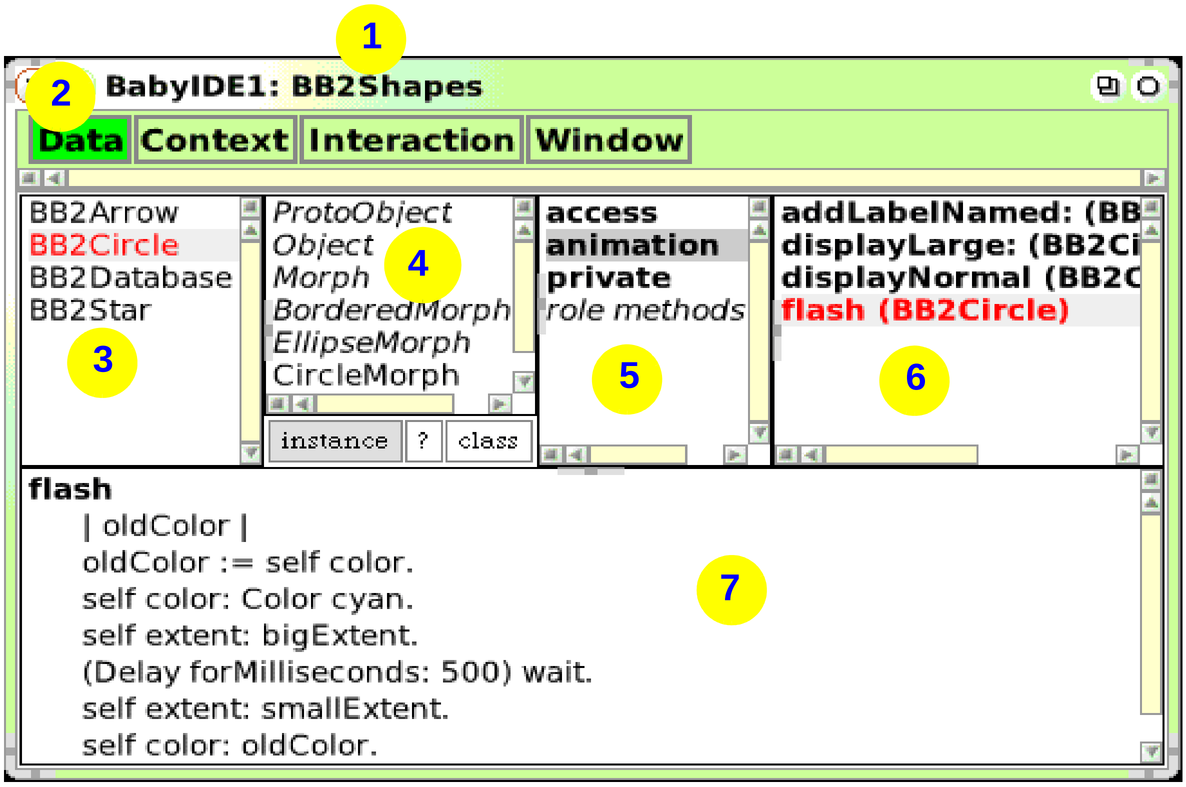
3. 1 *The BB1ClassBrowser*

The *BB1ClassBrowser* is a browser for working with the classes that are in the Squeak class categories with names beginning with *<appname>-<perspectivename>*. An example is shown in figure 12. Its panes are as follows:

1. The application is *BB2Shapes*.
2. The selected perspective is *Data*.
3. A class list showing the *Data* classes. *BB2Circle* is selected.
4. The superclasses of the selected class are shown in a multiple-select list. This is a presentation filter, only features of the selected class and its selected superclasses are shown. No superclass is selected here.
5. The method categories of the selected class and any selected superclasses are shown in a multiple-select list. This is a presentation filter; only methods belonging to the selected class, the selected superclasses and selected method categories are shown. The *animation* category is selected.Notice the Role Methods category. Methods in this category have been injected from the Interaction perspective; they are not edited here.
6. A list of methods in the selected class, superclasses, and method categories. The implementing classes are shown in parenthesis. The *flash* method is selected.
7. A code pane showing the selected method. The code is in italics if the method is not owned by this class. Examples are superclass methods and Role Methods.

1. The application name corresponds to the first part of the class category name in the Squeak class library. Try ‘Collections’ as an app name to see how BabyIDE works on regular Squeak categories.
2. BabyIDE applications cannot be filed out with Monticello because BabyIDE uses a subclass of Traits that does not load properly. BabyIDE1 must be loaded from SqueakMap before loading applications.

Figure 12: BB1ClasssBrowser showing the BB2Shapes>>Data.>>flash.



```
flash
    | oldColor |
    oldColor := self color.
    self color: Color cyan.
    self extent: bigExtent.
    (Delay forMilliseconds: 500) wait.
    self extent: smallExtent.
    self color: oldColor.
```

3. 2 The BB1InteractionBrowser

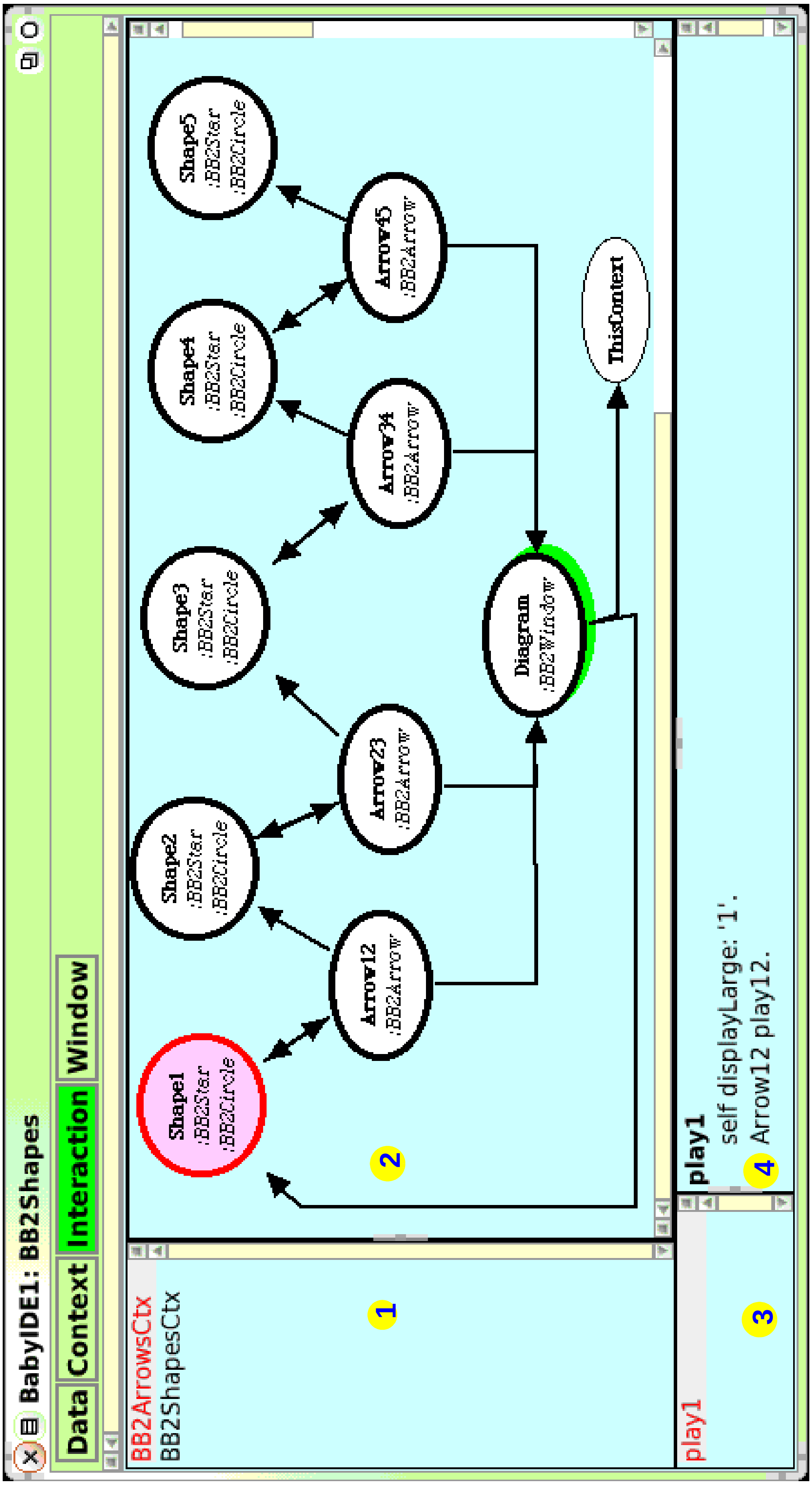
We ended section 2 with a request for an interactive development environment that supports the notions of context, Role, Role Methods, and Role binding. One of the main innovations in *BabyIDE* is the BB1InteractionBrowser; a new graphical browser that let us work with the implementation of a System Operation in the context of a network of interconnected Roles. This in contrast to class browsers where we only see one kind of object at the time.

The Interaction perspective (figure 13) is where we specify how a DCI program realizes a System Operation within a network of communicating objects. There are three critical questions that the code answers for each System Operation:

- *What are the objects?* The objects are represented by the Role they play, the actual objects are selected at runtime by the Role binding methods in the Context.
- *How are they interlinked?* We answer this question by displaying a network of connected Roles in the Context Diagram, see below.
- *How do they work?* We answer this question by augmenting some of the Roles with *Role Methods*[1]. These methods are injected into the classes of the Roleplaying objects. The only variables allowable in the Role Methods are:
  - *self*, i.e. the object that happens to play this Role.
  - *connected Roles*. The connected Roles as specified in the Context Diagram.

1. Role Methods are methods that are used by all classes that implement the Role. This requirement could have been a show stopper for the BabyIDE if it hadn't been for Traits[Schärli]. “A trait is essentially a group of pure methods that serves as a building block for classes and is a primitive unit of code reuse.” We simply give methods to a Role by attaching it to a Trait and let all Role playing Classes be users of this trait.

Figure 13: The *BB1InteractionBrowser*.



BB2ArrowsCtx
BB2ShapesCtx

play1

```
play1
    self displayLarge: '1'.
    Arrow12 play12.
```

Figure 13 shows a *BB1InteractionBrowser*. The panes are as follows:

1. *Contexts/System Operations:* A single-selection list of Contexts. There is one Context for each System Operation in the current browser, so this is also a list of *System Operations*. The *ArrowsAnimationCtx* is selected here.
2. *Context Diagram:* The Context Diagram for the selected System Operation. It shows the Roles of the interacting objects and their connectors. The *Shape1* Role is selected.
3. *Role Methods:* A single selection list of Role Methods for the Role selected in the diagram. The *play1* method is selected.
4. *code:* The code for the selected method.

We see a Context Diagram with the nine shape and arrow Roles. In addition, we see the *Diagram* Role that is bound to the colored background (figure 4 on page 10) and also the *ThisContext* Role that permits Role Methods to access the Context instance.

The *Context Diagram* is an example of graphical code. Roles and connectors can be added and removed as a convenient way of editing the Context.

Roles are played by objects that are specified by classes. *Shape1* is selected in figure 13. A class is bound to a Role with the '*add using class*' yellow-menu command. The instances of that class can then play the selected Role. We follow the UML style and show the Role names extended with every using class name preceded with a colon. We see from the diagram that *Shape2* has two using classes: *Shapes4Circle* and *Shapes4Star*. These classes thus share the *play2* Role Methods and will behave as specified in pane #4.

4 BB5Bank: A Simple Money Transfer Program According to DCI

James Coplien is implementing a DCI infrastructure in C++. He is also working on a simple program that illustrates the DCI essentials. The Squeak code for roughly the same problem is documented in this section.

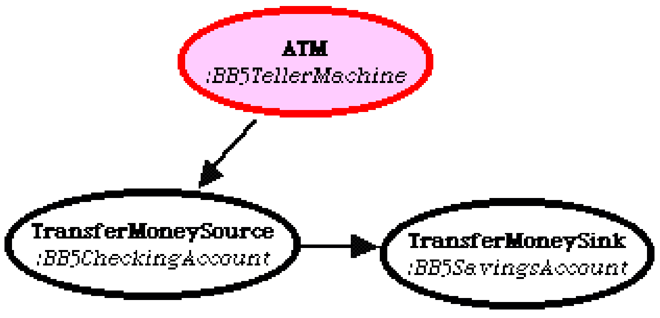
The example story is that a human uses an Automatic Teller Machine (ATM) to transfer funds from her checking account to her savings account. She knows that her checking account is account number 1111 and that her savings account is account number 2222. The amount she wants to transfer on this occasion is $500.

The problem is to program the transfer. We follow the DCI paradigm and organize our code in the three essential perspectives; Data, Context, and Interaction. We can freely choose to begin with the static, Data perspective or the dynamic, Interaction perspective because the two are very loosely coupled through the Context. We here choose to start with specifying system behavior in the Interaction perspective.

4. 1 Money Transfer in the Interaction Perspective

Figure 14 shows our choice of Roles and their connectors. The diagram also shows the Role Player Classes. These classes were not known initially, they were added to the diagram as we decided on the Role binding methods in the Context.

Figure 14: The Money Transfer Context Diagram.



There are three Roles involved in the Interaction:

ATM	The ATM machine itself
TransferMoneySourse	The transfer-from account.
TransferMoneySink	The transfer-to account.

The connectors in figure 14 show that we have not connected *ATM* to *TransferMoneySink* because we have planned for the actual transfer to be done by the *TransferMoneySource*. This is a design decision; the *ATM* could have done it all, but the chosen solution is more illustrative with its decentralized logic.

The Role Methods are as follows:

```
1.    ATM>>transferAmount: amount
2.        TransferMoneySource transfer: amount
```

line 2 Roleplaying objects are executed in a Context. In this case, the Context is instantiated with the *to* and *from* account numbers and binds the

TransferMoneySource and *TransferMoneySink Roles* to the appropriate account objects.

```
3.   TransferMoneySource>>transfer: amount
4.       self withdraw: amount.
5.       TransferMoneySink deposit: amount.

6.   TransferMoneySource>>withdraw: amount
7.       self balance < amount ifTrue: [self notify: 'Insufficient funds'. ^self].
8.       self decrease: amount.
```

There is a subtle distinction between *decrease:* and *withdraw* in this design. The first is something that the object can do by itself. The second is a part of a larger context such as a transaction for transferring funds. Note that *decrease* is specified in the class, *withdraw* is specified in the context of an Interaction and includes a check for legality.

There is the same distinction between the *deposit:* and *increase:* operations in the *TransferMoneySink* Role Methods:

```
9.   TransferMoneySink>>deposit: amount
10.      self increase: amount.
```

This code is independent of how the bank structures its data; we merely presume that it can provide objects that can play the three Roles.

All the above logic is coded as Role Methods. These methods are injected into the Role Playing Classes; these classes are specified with a menu command in the Role symbols shown in figure 14. We see that the *ATM* methods will be injected into the *BB5TellerMachine* class. We also see that the *TransferMoneySource* Role are played by instances of the *BB5CheckingAccount* class and the *TransferMoneySink* Role are played by instances of the *BB5SavingsAccount* class.

4. 2 Money Transfer in the ATM Perspective

Users interact with the ATM and cause things to happen in the system. The *BB5TellerMachine* class is an environment class because it includes triggers for system behavior. We have chosen to put this class in a separate perspective; we could equally well have put it in the Data perspective.

```
11.  Object subclass: #BB5TellerMachine
12.      uses: BB5MoneyTransferContextATM
13.      instanceVariableNames: 'bank'
14.      category: 'BB5Bank-ATM'
```

line 12 This line lists the Roles that inject Role Methods into this class. We see the other end of the using class relation that was specified for the *ATM* Role in figure 14. The Role names are prefixed with the name of the Context class.

The *BB5TellerMachine* class has accessor methods for the bank instance variables; they are not listed here.

The following Data method triggers a transfer:

```
15.  BB5TellerMachine>>transferFom: fromAccountNumber to: toAccountNumber amount: amount
16.      ( BB5MoneyTransferContext
17.                  data: self
18.                  transferFom:: fromAccountNumber
19.                  to: toAccountNumber
20.      )           transferAmount: amount.
```

line 16 - line 20: We request a service from a service provider called *BB5MoneyTransferContext*. There is no indication if this service is implemented with conventional OO or if it is implemented according to the DCI paradigm. We have to go into the receiver in code line 44 on page 23 to find that the receiver is a factory method for the *BB5MoneyTransferContext* class.

4. 3 Money Transfer in the Data Perspective

We define classes for a a rudimentary Bank and the two account classes:

```
21.  Object subclass: #BB5Bank
22.      instanceVariableNames: 'accounts'
23.      category: 'BB5Bank-Data'

24.  BB5Bank>>initialize
25.      super initialize.
26.      accounts := Dictionary new.

27.  BB5Bank>>addCheckingAccountNumbered: aNumber
28.      accounts at: aNumber put:  BB5CheckingAccount new.

29.  BB5Bank>>addSavingsAccountNumbered: aNumber
30.      accounts at: aNumber put:  BB5SavingsAccount new.

31.  BB5Bank>>findAccount: accountNumber
32.      ^ accounts at: accountNumber

33.  Object subclass: #BB5CheckingAccount
34.      uses: MoneyTransferContextTransferMoneySource
35.      instanceVariableNames: 'balance'
36.      category: 'BB5Bank-Data'

37.  Object subclass: #BB5SavingsAccount
38.      uses: MoneyTransferContextTransferMoneySink
39.      instanceVariableNames: 'balance'
40.      category: 'BB5Bank-Data'
```

The *BB5CheckingAccount* and *BB5SavingsAccount* classes have the following accessor methods: *balance*, *increase: amount*, and *decrease: amount*. In addition. they have an *initialize* method that sets the balance to 0. The injected Role Methods are visible in this Data perspective, but they are not edited here.

4. 4 *Money Transfer in the Context Perspective*

This perspective completes the program by specifying methods that bind Roles to objects at runtime.

```
41.    BB1Context subclass: #BB5MoneyTransferContext
42.        instanceVariableNames: 'fromAccountNumber toAccountNumber'
43.        category: 'BB5Bank-Context'
```

line 42 This Context class specifies the context for transfer between any two accounts. An instance of this class is initialized for transferring between the two accounts. The factory method is:

```
44.    BB5MoneyTransferContext class
           >>  data: aTellerMachine
               transfer: amount
               from: fromAccountNumber
               to: toAccountNumber
45.        | ctx |
46.        (ctx := self new)
47.            data: aTellerMachine;
48.            fromAccountNumber: fromAccountNumber;
49.            toAccountNumber: toAccountNumber;
50.            executeInContext: [(ctx at: #ATM) transferAmount: amount].
```

line 44: This is the entry point to the money transfer service. We instantiate the Context class and initialize it with the provided values.

line 50 We trigger the Interaction with the message `transferAmount: amount` to the *ATM* Role. It is received in code line 1 on page 20.

The Context is instantiated with sufficient data for binding the Roles:

```
51.    BB5MoneyTransferContext >>ATM
52.        ^data

53.    BB5MoneyTransferContext >>TransferMoneySink
54.        ^data bank findAccount: toAccountNumber

55.    BB5MoneyTransferContext >>TransferMoneySource
56.        ^data bank findAccount: fromAccountNumber
```

and the Interaction can start in code line 1 on page 20.

4. 5 *Testing the Money Transfer application*

A class, *BB5Testing*, has one class method that tests the application with a single money transfer:

```
57.    BB5Testing>>test1
58.        | teller bank |
59.        teller := BB5TellerMachine new.
60.        bank := BB5Bank new.
61.        teller bank: bank.
62.        bank addCheckingAccountNumbered: 1111.
63.        (bank findAccount: 1111) increase: 2000.
64.        bank addSavingsAccountNumbered: 2222.
```

```
65.
66.        Transcript clear; cr;
67.            show: 'Before: ';
68.            show: '1111: ';
69.            show: (bank findAccount: 1111) balance printString;
70.            show: ' // 2222: ';
71.            show: (bank findAccount: 2222) balance printString.
72.
73.        teller transferFom: 1111 to: 2222 amount: 500.
74.
75.        Transcript cr;
76.            show: 'After: ';
77.            show: '1111: ';
78.            show: (bank findAccount: 1111) balance printString;
79.            show: ' // 2222: ';
80.            show: (bank findAccount: 2222) balance printString.
```

line 63	Account 1111 (the checking account) is filled with $2000.
line 66	*Transcript* is a globally known text window that can be written to from any method at any time.
line 73	The test itself transferring $500 between the given accounts.

The result in the Transcript window is as expected:

Before: 1111: 2000 // 2222: 0

After: 1111: 1500 // 2222: 500

The program appears to work. But testing can only show the presence of bugs, not their absence. This test is the only case that has ever been executed with this program; the DCI promise is that the code shall be readable so that it can be audited. This is probably not true in this paper based, one dimensional form of the program; it is probably true when the program is read through a multi-dimensional IDE such as BabyIDE. You may want to audit the code for yourself. Load the BabyIDE[1] and hope you may experience the intense pleasure of finding bugs in other people's code.

1. Click *Downloads* in the DCI Home Page:
http://heim.ifi.uio.no/~trygver/themes/babyide/babyide-index.html

5 MVC and DCI - Two paradigms for readable code

I have a brain to think with, eyes and ears to observe with, and hands and feet to work with. I use my brain to maintain a model of the world around me so that I can understand how it works and protect me against unpleasant surprises. This model helps me interpret what I see and predict what will happen if I do things with my hands and feet.

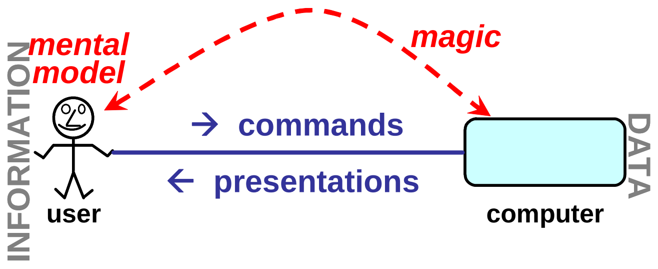
A computer system is part of my environment. A computer system that serves me well will let me build a mental model of its internals that helps me understand its presentations, remember its operations, and predict the effect of those operations.

The ISO vocabulary[ISO-66] distinguishes between *data* and *information* as follows:

> *DATA. A representation of facts or ideas in a formalized manner capable of being communicated or manipulated by some process.*
>
> *Note: The representation may be more suitable either for human interpretation (e.g., printed text) or for internal interpretation by equipment (e.g., punched cards or electrical signals).*
>
> *INFORMATION. In automatic data processing the meaning that a human assigns to data by means of the known conventions used in its representation.*
>
> *Note: The term has a sense wider than that of information theory and nearer to that of common usage.*

Figure 15 illustrates the relationship between user and computer. DATA exists in the computer where it is organized according to a *semantic model* or *schema*. The computer presents DATA in a way that facilitates my transformation of the observed presentation into INFORMATION in my brain. My mental model is the key to this transformation and the model is likely to be updated by the new information.

Figure 15: What it is all about.



If I am the user of an application, the challenge to its programmer is to give me the illusion that I am working directly into my mental model; the computer with its I/O devices recedes into the background. I am the master, the computer is my slave.[1] Douglas Engelbart calls this mode of working *Computer Augmentation* where the computer is used as an intellect-augmentation device. [Engelbart-62]

If I am the programmer of an application, the challenge to the toolmaker who provides my interactive development environment (IDE) is exactly the same, only lifted onto a higher level. The

1. This way of working is the exact opposite of script driven Interaction where the computer leads me through a sequence of predefined steps. My role is reduced to answering the computer's questions. Typical examples are the so called wizzards where I often do not know the underlying semantic model and, therefore, do not know what I should answer to their cryptic questions.

key question is “what is a program”? The answer to this question should be the same for me and for the toolmaker so that I work with tools that reflect my mental model of my program. More precisely, my mental model and the computer representation of my program must follow the same paradigm.

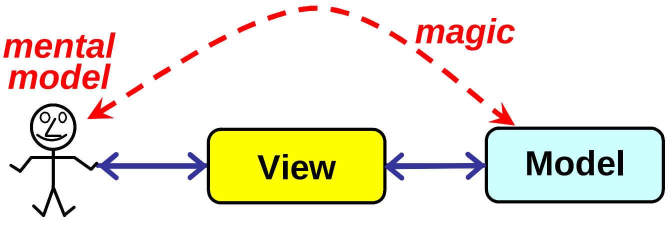
MVC and *DCI* are two paradigms that are designed to be shared by me as a programmer and my program as represented in my code. MVC is old, I implemented the first program designed according to this paradigm in 1978. DCI is new, the first program designed according to this paradigm was the *BB2Shapes* visualization program that is described in section 6 on page 31. The first IDE supporting DCI is the *BabyIDE1* described in section 3 on page 15.

5. 1 MVC: the Model-View-Controller paradigm

The Model-View-Controller paradigm[MVC] separates the parts of a program that are responsible for representing the information in the system and the parts that are responsible for interaction with the user. Figure 16 illustrates this separation.

The DATA that represents my mental model within the computer is called the *Model*. A *View* is responsible for presenting the Model in a way that makes it easy for me to transform the presented data into information in my brain and that helps me understand the effects of commands that I can give through the View.

Figure 16: GUI bridges gap between brain and model.

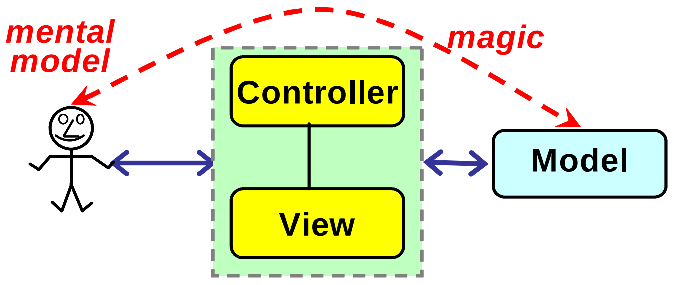


The essential element in MVC is the user. Some implementors miss this essential point and see MVC solely as a programming mechanism. Jim Coplien wants to stress that the human part is the raison d'etre of MVC and renames it to MVC-U: *Model-View-Controller-User.*

The separation between Model and View leads to greatly simplified code due to the separation of the user interface code from the Model code. The separation leads to a software interface between the Model and the View so that the model can support different local or remote views.

Some user tasks require computer-based tools that include several, coordinated views. This coordination is done by a Controller as illustrated in figure 17 and is exemplified in the planning demo in figure 23 on page 39. Typical responsibilities for a Controller includes the coordination of element selection in the Views and the handling of user commands that concern the tool as a whole.

Figure 17: The Model-View-Controller (MVC) paradigm.



The MVC paradigm has proven its value over its 30 years existence. It is often applied as a role pattern without any specific class library support.

The *Activity Planning* application discussed in section 7 on page 39 uses MVC for the user interface architecture and DCI for detailing the Model.

5. 2 DCI: the Data-Context-Interaction paradigm

Some models are complex. The DCI paradigm was originally intended for the detailing of the Model part of MVC, but we now believe that DCI is more general and has many applications outside this scope.

The idea behind the DCI paradigm is to separate the static code that describes system state from the dynamic code that describes system behavior. The code is organized in several perspectives where each perspective focuses on certain aspects of the program. The main perspectives are called ***D****ata*, ***C****ontext*, and ***I****nteraction*:

- The ***D*** for ***Data*** perspective specifies a “micro database” that includes the system domain classes.
- The ***C*** for ***Context*** perspective. A System Operation is executed by a network of communicating objects. Different executions of the same operation may be done by different objects. DCI requires that the networks formed by these objects shall share the same topology.

  A Context class defines this topology as a network of interconnected Roles.

  A Context class defines methods that bind Roles to objects at runtime.[1]

  A Context class has a public interface consisting of the System Operations that can be executed in the Context.
- The ***I*** for ***Interaction*** perspective includes methods that specify how objects collaborate as they realize a System Operation. The Interaction can be coded as *Role Methods* attached to the Roles. Role Methods are injected every class that realize the Role.

We will first describe the part of the paradigm that covers the system state before we add system behavior.

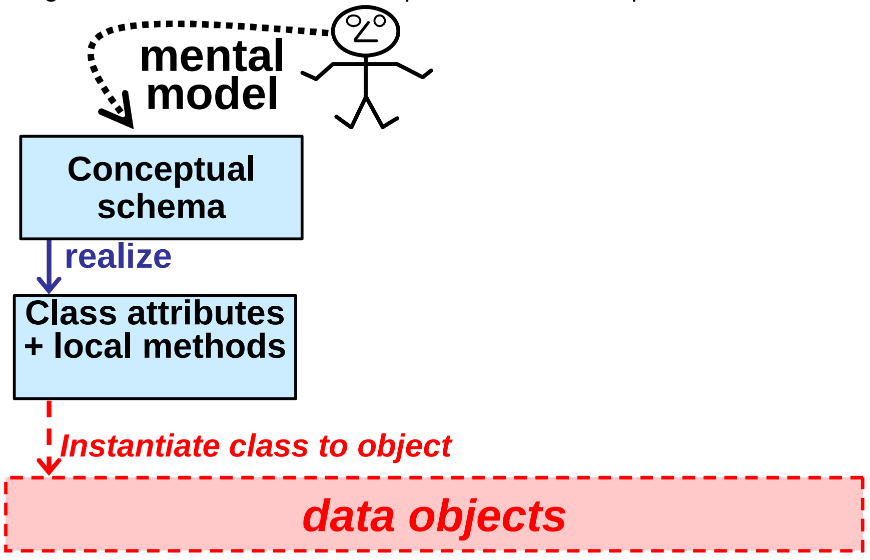
5. 2. 1 The Data perspective

System state is realized by the state of the Data objects and the relations between them. The Data class definitions can be seen in the *Data perspective*. This perspective resolves the magic of figure 15 into concrete code. Figure 18 illustrates this part of the DCI paradigm. The user’s mental model is described in a real or virtual *conceptual schema*. A suitable language for the schema could be something like NIAM[NIAM], ODMG[ODMG], or simply UML class diagrams[UML]. An example is shown in figure 24 on page 40.

The simplest Data description is probably class definitions in the current programming language. This simple solution also opens a capability for specifying local object behavior, i.e., behavior realized within the Data object’s encapsulation. Encapsulated local behavior cannot interfere with the Interactions that implement the System Operations.

1. This will be a *select* operation in database terms, the Context as a whole can be seen as a kind of external view on the Data.

Figure 18: User mental model represented as state part of Data classes.



The Data classes are instantiated to form an object structure that corresponds to the user's Conceptual Schema. What appears as magic to the user is simply a set of instantiated Data classes that are structured according to the schema.

Note that the actual Data objects and their relations are runtime phenomena. These runtime elements are symbolized by a red box with dashed border in figure 18.

5. 2. 2 System behavior: The Context and Interaction perspectives

The system runtime behavior is the system's response to user commands. A user command[1] triggers a method in one of the system's objects. This method sends further messages so that the command is effectuated by a network of communicating objects.[2]

It is easy to write a program that passes all tests but that also hides the secret of how it works behind a tangle of intractable code. I know, because the first version of *BabyIDE* is a good and recent example.

A program that follows the DCI paradigm exposes its inner workings to a reader of its code. In a DCI program, each network of communicating objects is coded as a corresponding network of connected Roles. The establishment and maintenance of the runtime network structure is no longer the responsibility of the participating Data objects, but is centralized in a new element called the *Context*.

The DCI paradigm is based on the concepts of Role Modeling[OOram], Collaborations[UML], and Traits[Schärli]. UML describes Collaboration as follows:

> *A collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality. Its primary purpose is to explain how a system works and, therefore, it typically only incorporates those aspects of reality that are deemed relevant to the explanation. Thus, details, such as the identity or precise class of the actual participating instances are suppressed.*[UML]

1. We use the terms use case, user command, and System Operation interchangeably in our discussion of the DCI paradigm because they all describe phenomena that trigger system behavior.
2. We only consider single thread execution in the current version of the DCI paradigm.

The main difference between the UML Collaboration and the DCI Context is that where UML describe and explain, DCI specify in code.

The system behavior part of the DCI paradigm is illustrated in figure 19. The elements are as follows:

Environment: A class that specifies how the execution of a *System Operation* is triggered by a message to the Context. Environment class examples are *BB5TellerMachine* in the Bank example (section 4 on page 20), *BB2Window* in the Shapes example (section 6 on page 31), and *BB4bController* in the Planning example (section 7 on page 39).

Context: A class that implements one or more System Operations. It also specifies the topology of networks of communicating objects as a similar structure of Roles and Connectors.[1] The Context class also specifies methods that bind Roles to objects at runtime and the methods that trigger Role Interaction.

Interaction: A specification of how objects interact to realize a System Operation. In the code, participating objects are represented by the Roles they play in the Interaction.

Figure 19: System behavior realized in a Context.

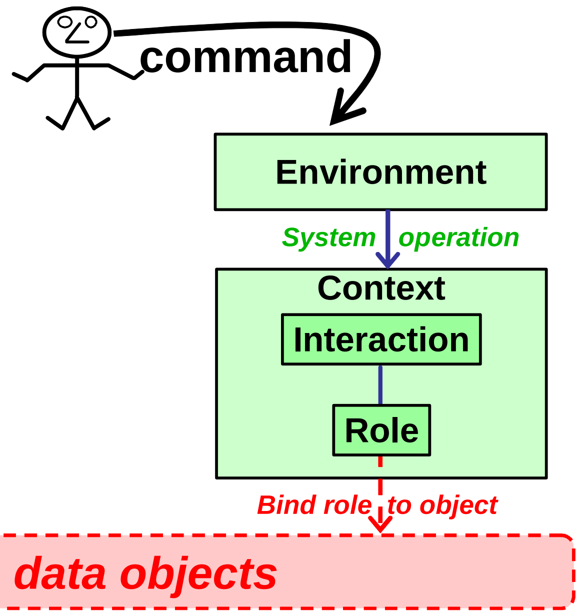


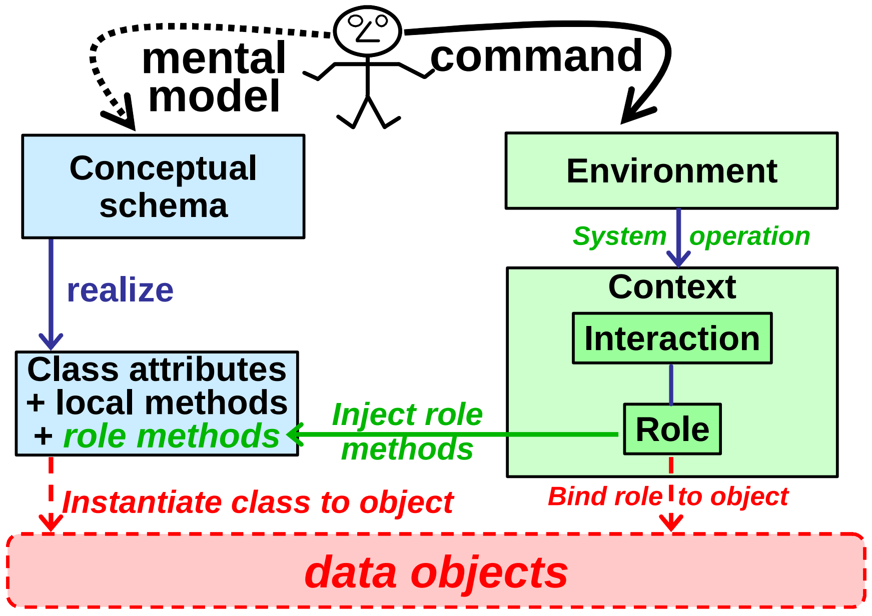
Figure 19 illustrates system behavior up to Interaction messages. Its weakness is that the methods triggered by these messages will conventionally be features of the Data classes. This means that different objects that play the same Role may handle the same message with different methods and violate the network topology specified in the Context.

One way to resolve this ambiguity by enforcing a constraint. *All objects that play a given Role shall process the same Interaction messages with the same methods.* These methods are called *Role Methods* and are features of the Role. The Role Methods are *injected* into the Data classes that may not override them. A code reader can, therefore, trust that there are no surprises hidden in the details of the Data Classes.

1. The notion of communicating Roles is taken from OOram and is described intuitively in section 2.

Figure 20 illustrates the complete DCI paradigm with an injection relation from Role to class. This relation means that the instances of these classes will give priority to the Role Methods above any methods defined in the class itself. [1]

Figure 20: The complete DCI paradigm.



We see that it is clear that the code reveals everything about how a system will work. We have resolved the problem stated in Design Patterns[GOF-95] by replacing unreadable code with code that conforms to the DCI paradigm. The code is readable.

1. Traits are used to implement Role Methods in BabyIDE. Our usage prevents hidden surprises by blocking classes overriding Trait methods.

6 The *BB2Shapes* Process Visualization Program

We discussed the processes that are visualized in the *BB2Shapes* program in section 2 on page 9. We now shift our attention to the program that drives this animation.

We code the *BB2Shapes* program as seen in four perspectives; the three DCI perspectives and a Window perspective. The latter specifies the colored background for the animation and provides the command menu that lets the user control the animation. We find that the *Window* and *Data* classes are almost regular classes as we would write them in our usual programming style. *Almost*, because we have pulled out the methods that relate to the runtime system structure and behavior. The *Window* and *Data* classes are, therefore, much simplified and we can trust that they do not hide ugly surprises.

6. 1 The BB2Shapes program seen in the Data perspective

The browser for the *Data* perspective is a *BB1ClassBrowser* as described in section 3. 1 on page 16. There are four classes:

BB2Star	An instance displays a shape in the form of a star on the screen.
BB2Circle	An instance displays a shape in the form of a circle on the screen.
BB2Arrow	An instance displays an arrow that can grow from a start shape to a destination shape, thus visualizing a message transmission.
BB2Database	An instance maintains a reference to every existing shape and arrow object.

An important feature of the DCI style of programming is that the Data classes do not define the runtime object structure. This can be seen from the Data class definitions:

```
81.   StarMorph subclass: #BB2Star
82.       uses: BB2ArrowsCtxShape5 + BB2ArrowsCtxShape4 + BB2ArrowsCtxShape1 + BB2ArrowsCtxShape2 +
      BB2ArrowsCtxShape3
83.       instanceVariableNames: 'smallExtent bigExtent'
84.       category: 'BB2Shapes-Data'
```

Comments:

line 81:	*BB2Star* is subclass of the library class *StarMorph*.
line 82:	With DCI, a great deal of code is shifted from the Data classes to the Methodful Roles (Traits) listed here[1]. The shifted code will be executed by the instances of the *BB2Star* class, but it is specified elsewhere and is visible here for reference only.
line 83:	The instance variables are only *smallExtent* and *bigExtent*, there is no reference to the arrows that bind shapes together.

1.The Trait name is the Role name prefixed by the name of the Context class.

```
CircleMorph subclass: #BB2Circle
    uses: BB2ArrowsCtxShape5 + BB2ArrowsCtxShape4 + BB2ArrowsCtxShape1 + BB2ArrowsCtxShape2 +
        BB2ArrowsCtxShape3
    instanceVariableNames: 'smallExtent bigExtent'
    category: 'BB2Shapes-Data'

LineMorph subclass: #BB2Arrow
    uses: BB2ArrowsCtxArrow34 + BB2ArrowsCtxArrow45 + BB2ArrowsCtxArrow12 + BB2ArrowsCtxArrow23
    instanceVariableNames: ' '
    category: 'BB2Shapes-Data''

Object subclass: #BB2Database
    instanceVariableNames: 'environment stars arrows'
    category: 'BB2Shapes-Data'
```

All methods defined in the Data perspective are “local” in the sense that they do not have system-wide side effects. Examples:

```
BB2Star>>flash
    | oldColor |
    oldColor := self color.
    self color: Color yellow.
    self extent: bigExtent.
    (Delay forMilliseconds: 500) wait.
    self extent: smallExtent.
    self color: oldColor.
```

The *BB2Arrow* class does not define instance variables for start and end shapes, these values are supplied from the Interaction at runtime:

```
LineMorph subclass: #BB2Arrow
    uses: BB2ArrowsCtxArrow12 + BB2ArrowsCtxArrow34 + BB2ChaosCtxArrow + BB2ArrowsCtxArrow23
        + BB2ArrowsCtxArrow45
    instanceVariableNames: ''
    category: 'BB2Shapes-Data'

BB2Arrow>>growFrom: startShape to: endShape
    | stepMax pt1 pt2 startPoint |
    stepMax := 10.
    startPoint := (startShape attachPointFrom: endShape center) rounded.
    self makeForwardArrow.
    1 to: stepMax do:
        [:stepCounter |
        pt1 := startPoint.
        pt2 := (endShape attachPointFrom: pt1) rounded.
        self
            verticesAt: 1 put: pt1;
            verticesAt: 2 put: (pt1 + (pt2 - pt1 * stepCounter // stepMax)) rounded.
        (Delay forMilliseconds: 20) wait].
```

6. 2 The BB2Shapes program seen in the Window perspective

The *BB2Window* class specifies the environment of the animation behavior; user menu commands start the animation System Operations.

The *BB2Window* class definition is as follows:

```
121.  PasteUpMorph subclass: #BB2Window
122.      BB2ShapesCtxShapesAnimator + BB2ArrowsCtxDiagram
123.      instanceVariableNames: 'data currentState processSemaphore '
124.      category: 'BB2Shapes-Window'
```

Comments:

line 121: The *PasteUpMorph* superclass gives *BB2Window* the capability to show the animation in a window on the screen.

line 122: This line lists the Roles that inject Role Methods into this class. See the comment to code line 12 on page 21.

line 123: *data* is the 'database' that holds Data objects. The Data classes are coded in the Data perspective (section 6. 2. 1).

line 124: The category name is used by *BabyIDE* to find the *Window* classes.

The *BB2Window* class forms the bridge between the system and its environment. The user gives commands through a menu:

```
125.  BB2Window>>yellowButtonActivity: shiftKeyState
126.      | aMenu |
127.      aMenu := (MenuMorph new defaultTarget: self) addTitle: self printString;
128.                  add: 'animate shapes' action: #startShapeAnimation;
129.                  add: 'animate arrows' action: #startArrowAnimation;
130.                  addLine;
131.                  add: 'stop animation' action: #stopAnimation;
132.                  add: 'EXIT' action: #exitDemo.
133.      aMenu popUpInWorld
```

Comment:

line 128, line 129: The *animateShapes* command triggers the *startShapeAnimation* method and the *animate arrows* command triggers the *startArrowAnimation* method. Each of these methods instantiates the appropriate Context and triggers a System Operation:

```
134.  BB2Window>>startArrowsAnimation
135.      | ctx |
136.      currentState = #ARROWS ifTrue: [^self].
137.      currentState := #ARROWS.
138.      processSemaphore wait.
139.      [   BB2ArrowsCtx startArrowsAnimationOn: data.
140.          processSemaphore signal.
141.      ] fork.
```

Comments:

line 136: We don't start the animation if it is already running.

line 139: Command the Context to start the animation. Note that this line is a simple service request. The decision to use DCI rests with whatever object is referenced by BB2ArrowsCtx. It is here a Context class and DCI is triggered in code line 163 on page 35.

line 141: The *ArrowAnimation* loop is executed in a separate process. Without it, the animation process would hog the CPU and block menus and any other work one might want to do while the animation is running. This is a Squeak-technical detail that is irrelevant from the DCI point of view.

Similarly for the other System Operation:

```
142.  BB2Window>>startShapesAnimation
143.      currentState = #SHAPES ifTrue: [^self].
144.      currentState := #SHAPES.
145.      processSemaphore wait.
146.      [   BB2ShapesCtx startShapesAnimationOn: data.
147.          processSemaphore signal.
148.      ] fork.
```

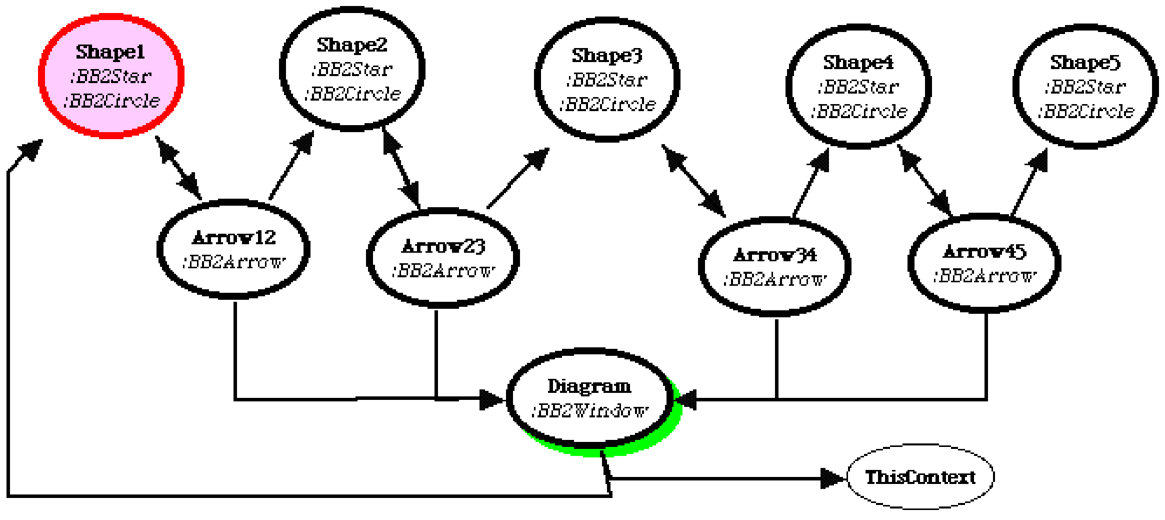
We now follow the System Operations called in line 139 and line 146 into their respective Contexts.

6. 2. 1 The BabyShapes2 ArrowsAnimation System Operation

The result of the ArrowsAnimation is shown in figure 4 on page 10. Here is the code that drives the animation.

ArrowsAnimation in the Context perspective

Figure 21: The ArrowsAnimation Context Diagram.



The Context Diagram is shown in figure 13 on page 18 and is repeated here for convenience in figure 21. The network topology is specified in a class (static) method in the *BB2ArrowCtx* class:

```
149.  BB2ArrowsCtx class>>roleStructure
150.  ^super roleStructure
151.      at: #ThisContext put: #();
152.      at: #Arrow12 put: #(#Shape2 #Shape1 #Diagram );
153.      at: #Shape3 put: #(#Arrow34 );
154.      at: #Arrow34 put: #(#Shape3 #Diagram #Shape4 );
155.      at: #Shape5 put: #();
156.      at: #Arrow23 put: #(#Shape2 #Shape3 #Diagram );
157.      at: #Shape1 put: #(#Arrow12 );
```

```
158.        at: #Shape2 put: #(#Arrow23 );
159.        at: #Shape4 put: #(#Arrow45 );
160.        at: #Arrow45 put: #(#Shape5 #Shape4 #Diagram );
161.        at: #Diagram put: #(#Shape1 #ThisContext );
162.            yourself
```

In addition, using relationships are established between Roles and Classes. They are set in the Interaction perspective as shown in figure 21 and are visible in the Data class definitions, see for example the *uses* parameter in the *BB2Circle* class definition in code line 86 on page 32.

Contexts implement System Operations. Here it is *startArrowsAnimation*, it is called from the Window in code line 139 on page 33. The public Context class (factory) method bridges the chasm from the static realm of regular Squeak code to the dynamic realm of Interactions and Role Methods:

```
163. BB2ArrowsCtx class>>startArrowsAnimation
164. | ctx |
165.     (ctx := self new)
166.         data: aData;
167.         executeInContext: [(ctx at: #Diagram) animateArrows].
```

line 167 This method establishes the current Context and triggers the Interaction by sending the *animateArrows* message to the object playing the *Diagram* Role.

The Role to object bindings are specified in Role binding instance methods named after the Role. Some examples:

```
168. BB2ArrowsCtx>>Shape1
169.     ^data anyShape

170. BB2ArrowsCtx>>Arrow12
171.     ^data newArrow

172. BB2ArrowsCtx>>Diagram
173.     ^ data window

174. BB2ArrowsCtx>>ThisContext
175.     ^self
```

ArrowsAnimation in the Interaction perspective

The Context Diagram for the Arrows animation is shown in figure 21. The following method was triggered from code line 167.

```
176. Diagram>>animateArrows
177.     [self currentState == #ARROWS]
178.     whileTrue:
179.     [   ThisContext removeAllArrows.
180.         ThisContext reselectObjectsForRoles.
181.         Shape1 play1.
182.         (Delay forMilliseconds: 1500) wait
183.     ].
184.
```

Comments:

line 176: This Role Method is a feature of the *Diagram* Role.

line 180: *ThisContext* is the Role name for the current Context (See Role binding code line 174). We here reset all bindings.

line 181: We start drawing the repeating pattern by executing the *play1* method in the *Shape1* Role and the Interaction continues from there.

```
185. Shape1>>play1
186.     self displayLarge: '1'.
187.     Arrow12 play12.
```

Comment:

line 186: *self* is a legal variable; it refers to the object playing the Role. Here, the object playing the *Shape1* Role will be the receiver. (The actual object will, of course, be different at different times. We see from figure 21 on page 34 that the object will be an instance of either *BB2Star* or *BB2Circle*.)

```
188. Arrow12>>play12
189.     Diagram addMorphBack: self.
190.     self growFrom: Shape1 to: Shape2.
191.     Shape2 play2.

192. Shape2>>play2
193.     self displayLarge: '2'.
194.     Arrow23 play23.

195. Arrow23>>play23
196.     Diagram addMorphBack: self.
197.     self growFrom: Shape2 to: Shape3.
198.     Shape3 play3.
```

And so on until *Shape5*.

The *BB2ArrowsCtx* answers the essential questions about the structure of the interacting objects: What are the Roles? How are they connected? The Interaction answers the last essential question: How do they interact? There are no surprises; the Data Classes do not know "*the network of communicating objects*" so no overworked maintainer can upset our grand scheme.

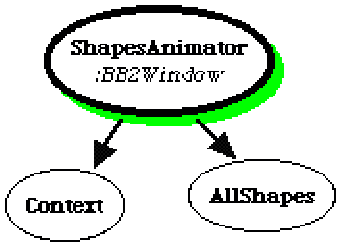
The code is readable.

6. 2. 2 The ShapesAnimation use case

The *ShapesAnimation* involves a single shape at the time. This shape is either added or removed from the pool of shapes.

ShapesAnimation in the Context perspective

Figure 22: The ShapesAnimation Context Diagram.



The Context Diagram is extremely simple as shown in figure 22 and so is its defining code:

```
199.  BB2ShapesCtx class>>roleStructure
200.  ^ super roleStructure
201.      at: #AllShapes put: #();
202.      at: #Context put: #();
203.      at: #ShapesAnimator put: #(#AllShapes #Context );
204.      yourself.
```

The System Operation command is sent from the Window code line 146 on page 34 and received here. We trigger the Interaction in the *ShapesAnimator* Role:

```
205.  BB2ShapesCtx class>>startShapesAnimationOn: aData
206.      | ctx |
207.      (ctx := self new)
208.          data: aData;
209.          executeInContext: [(ctx at: #ShapesAnimator) animateShapes].
```

We see from the diagram that the *ShapesAnimator* Role is *injecting its Role Methods into* the *BB2Window* class so that *ShapesAnimator* Role Methods are equivalent to *BB2Window* instance methods.

ShapesAnimation in the Interaction perspective

We see from figure 22 that the central Role is the *ShapesAnimator*. Its Role Methods can reference the *AllShapes* and *Context* Roles; both happen to be Methodless Roles. Notice that the current Context was called *ThisContext* in the *ArrowsAnimation*, just *Context* here. Role names are local to the Context, we can call them anything we like because the Role binding methods in the Context will bind them to the right objects.

We entered the Context and started the ball rolling in line 209 by sending *animateShapes* to the object playing the *ShapesAnimator* Role:

```
210.  ShapesAnimator>>animateShapes
211.      [self currentState == #SHAPES]
212.          whileTrue:
213.          [   Context reselectObjectsForRoles.
214.              AllShapes size >= 50
215.                  ifTrue: [self deleteShape].
216.              AllShapes size <= 50
217.                  ifTrue: [self addShape]
218.          ]
```

```
ShapesAnimator>>deleteShape
    | shape |
    (shape := Context anyShape)
    ifNotNil
            [Context removeShape: shape.
            shape delete].

ShapesAnimator>>addShape
    | newShape margin newCenter |
    newShape := (Collection randomForPicking next * 10) rounded odd
                            ifTrue: [Context newShape: BB2Star]
                            ifFalse: [Context newShape: BB2Circle].
    margin := newShape extent // 2 .
[       newCenter :=
                (self bounds left + margin x to: self bounds right - margin x) atRandom
                @ (self bounds top + margin y to: self bounds bottom - margin y) atRandom.
        AllShapes
            noneSatisfy:
                [:someShape | (someShape fullBounds extendBy: newShape extent)
                    containsPoint: newCenter]
                ] whileFalse.
    newShape center: newCenter.
    self addMorphBack: newShape.
    newShape flash.
```

These methods send several messages to *self*, the currently bound shape object, They are all local methods and don't interfere with the animation as a whole. There can be no surprises; the important parts are seen in the Interaction perspective and we conclude that the code for this use case is readable.

Using DCI here is a matter of taste, and the decision is local to the **BB2ShapesCtx** class. The advantage is that all the code for the *ShapesAnimation* is pulled out of the *BB2Window* class so that the whole story is collected in the one perspectives. Some programmers might prefer to collect all the code into the *BB2Window* class where it is injected anyway. The disadvantage of such a monolithic solution is that the system behavior code gets mixed up with the other code of the already pretty large *BB2Window* class. It also feels wrong architecturally because the internal system code gets to be mixed with the external environment code.

In conclusion, we prefer to follow DCI so that this use case is implemented the same way as other system behaviors.

7 BB4bPlan: An Activity Network Planning Program with DCI

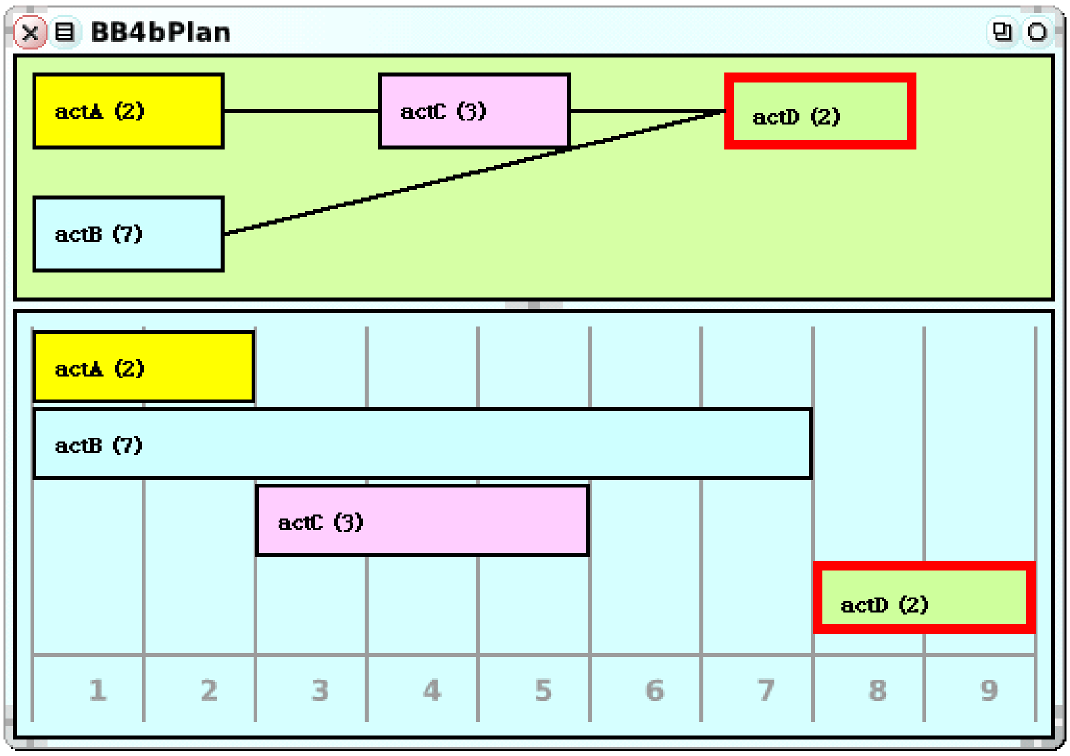
This example is a very rudimentary activity planning application that demonstrates the use of MVC and DCI in combination. There is no user data input; the example network is hard coded.

Two versions of this application will be discussed here. The versions are opened in Squeak from the *World Menu>open>BB4xPlan*, where x is *a* or *b*.

BB4aPlan	A conventional application without DCI.
BB4bPlan	The application coded with DCI.

Both versions are identical from the user's point of view. Both open a window with two Views as shown in figure 23. The top view is a dependency graph that shows the activities with their technological dependencies. The duration of each activity is given in parenthesis after the activity name. The bottom view shows a Gantt diagram with the frontloaded activities laid out along the time axis.

Figure 23: The Activity Planning demo.



Three use cases are realized as menu operations:

build demo network	Build the demo activity network and display it as a dependency graph.
frontload from week 1	*Frontloading* is to compute the earliest start for all activities. An activity can start when all its predecessors are finished.
reset demo	Remove the current network.

We will describe the DCI version, *BB4bPlan*, in this section. In the next section, section 8 on page 53, we compare the DCI version with *BB4aPlan*, the conventional OO programming version. We find that the executed code is essentially the same in both versions, but the code seen by the programmer is organized differently. This leads to very different readability, programmer experience, and programmer's mental model.

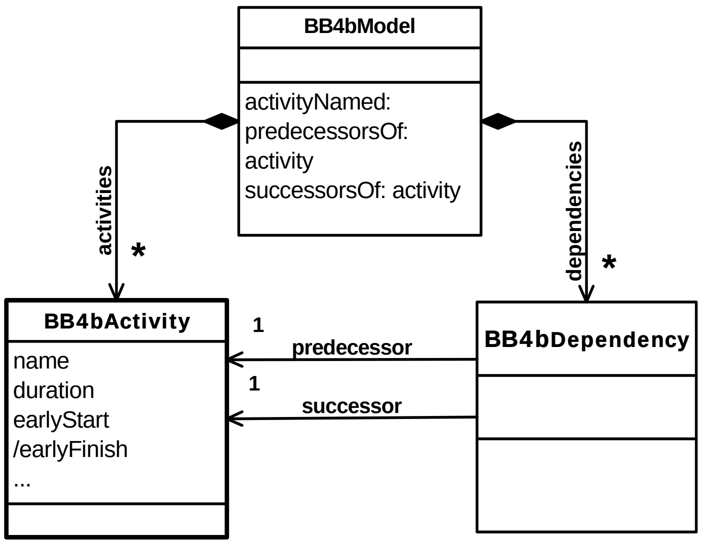
This DCI version is coded in five perspectives:

Data	The MVC *Model*. There are classes for the Model itself and for its parts: *activities* and *dependencies*.
View	The MVC *Views*. There is one class for each view: *BB4bDependencyView*, *BB4bGanttView*, and *BB4bActivityView*.
Controller	The MVC *Controller*. Sets up and coordinates the views. Both views show the same activities so activity selection applies to both views. (Figure 23 shows that *actD* is selected as can be seen in both views).
Context	The DCI Context classes. The *BB4BFrontloadCtx* is responsible for performing the frontloading System Operation. The *BB4bDependencyCtx* is responsible for creating the dependency graph. The *BB4bGanttCtx* is responsible for creating the Gantt diagram.
Interaction	The DCI Interactions. There is one Interaction for each Context.

7. 1 The BB4b Data perspective

The Data classes are dumb classes that know nothing about the implementation of system behavior; they only implement object state and local behavior.

Figure 24: The BB4b class diagram.



The data model is expressed as a UML class diagram in figure 24. The model should be supplemented with a constraint that prevents circular dependencies.

Conventional OO programming would let *predecessors* and *successors* be instance variables in the *Activity* class. We have chosen to move them out into a *Dependency* class for architectural reasons. An additional consideration is that it enhances readability and ensures data integrity.

The corresponding class definitions are as follows:

The BB4bModel class

```
242. Object subclass: #BB4bModel
243.     uses: BB4bFrontloadCtxFrontloader
244.     instanceVariableNames: 'activities dependencies'
245.     category: 'BB4bPlan-Data'
```

line 243 Role Methods have been injected from the *Frontloader* Role in the *FrontloadCtx* in the *BB4b* application. See the Context Diagram in figure 25 on page 44.

Two of the local access methods:

```
246. BB4bModel>>activityNamed: actNam
247.     | act |
248.     act := activities detect: [:a | a name = actNam] ifNone: [nil].
249.     act ifNil: [self error: 'Activity ' , actNam , ' does not exist.'. ^nil].
250.     ^act

251. BB4bModel>>predecessorsOf: succ
252.     | preds |
253.     preds := Set new.
254.     dependencies do: [:dep | dep successor == succ ifTrue: [preds add: dep predecessor]].
255.     ^preds
```

Two local Data definition methods:

```
256. BB4bModel>>newActivityNamed: nam duration: dur color: col
257.     | act |
258.     act := BB4bActivity name: nam duration: dur color: col.
259.     activities add: act.
260.     self changed: #model.

261. BB4bModel>>newDependencyFrom: predNam to: succNam
262.     | pred succ |
263.     pred := self activityNamed: predNam.
264.     succ := self activityNamed: succNam.
265.     (self hasDependencyFrom: pred to: succ)
266.     ifFalse:
267.         [dependencies add:
268.             (BB4bDependency new
269.                 predecessor: pred
270.                 successor: succ).
271.         self changed: #model].
```

The BB4bActivity class

```
272. Object subclass: #BB4bActivity
273.     instanceVariableNames: 'earlyStart duration name color'
274.     category: 'BB4bPlan-Data'

275. BB4bActivity>>earlyStart
276.     ^earlyStart
```

earlyStart is an owned attribute (instance variable) while *earlyFinish* is a derived attribute (a method):

```
277.  B4bActivity>>earlyFinish
278.      ^ earlyStart
279.          ifNil: [nil]
280.          ifNotNil: [earlyStart + duration - 1]
```

The BB4bDependency class

```
281.  Object subclass: #BB4bDependency
282.      instanceVariableNames: 'predecessor successor'
283.      category: 'BB4bPlan-Data'

284.  BB4bDependency>>predecessor
285.      ^ predecessor

286.  BB4bDependency>>successor
287.      ^successor
```

7. 2 The BB4b Controller perspective

There is only one Controller class. It is responsible for setting up the demo model and its views, for coordinating them for common activity selection, and for responding to the menu commands: *build demo network*, *frontload from week 1*, and *reset demo*.

```
288.  SystemWindow subclass: #BB4bController
289.      instanceVariableNames: 'dependencyView ganttView selectedActivity'
290.      category: 'BB4bPlan-Controller'
```

Creating the window with its views is regular Squeak programming, we will not go into the details here. A menu selects the operations:

```
291.  BB4bController>>yellowButtonActivity: shiftKeyState
292.      | aMenu |
293.      aMenu := (MenuMorph new defaultTarget: self)
294.          addTitle: self printString;add: 'build demo network' action: #buildDemoNetwork;
295.          add: 'frontload from week 1' action: #frontloadDemo;
296.          add: 'reset demo' action: #resetDemo.
297.      aMenu popUpInWorld.
```

Menu commands in line 294, line 295, and line 296 call methods that trigger the System Operations. Very simple operations are *resetDemo* and *buildDemoNetwork*; DCI is not needed:

```
298.  BB4bController>>resetDemo
299.      self model: nil.
300.      dependencyView deleteContents.
301.      ganttView deleteContents.
```

```
302.  BB4bController>>buildDemoNetwork
303.        model ifNotNil: [self resetDemo].
304.        model := BB4bModel new.
305.        model
306.                newActivityNamed: 'actA' duration: 2 color: Color yellow;
307.                newActivityNamed: 'actB' duration: 7 color: Color lightBlue;
308.                newActivityNamed: 'actC' duration: 3 color: Color lightMagenta;
309.                newActivityNamed: 'actD' duration: 2 color: Color lightGreen.
310.        model
311.                newDependencyFrom: 'actA' to: 'actC';
312.                newDependencyFrom: 'actB' to: 'actD';
313.                newDependencyFrom: 'actC' to: 'actD'.
314.        self changed: #model.
```

We see that the activity network is hard coded. Code in line 314 activates the Observer pattern. We will later see the result of this *changed: #model* -message in the *update*: methods in the View classes. (code line 363 on page 46, code line 383 on page 47, and code line 392 on page 48).

```
315.  BB4bController>>frontloadDemo
316.        model ifNil: [self inform: 'Define the model before frontloading. \Command ignored.' withCRs. ^ self].
317.        BB4bFrontloadCtx data: model frontloadNetworkFrom: 1.
318.        self changed: #model.
```

line 317 We request the frontload service; it is provided by *BB4bFrontloadCtx*. The request is picked up by the Context class in code line 342 on page 45.

The Controller manages activity selection when triggered by a mouse click on an Activity symbol in a diagram. We could have used DCI here, but have arbitrarily chosen to use the well known Observer pattern. The following method is called from an ActivityView (code line 362 on page 46).

```
319.  BB4bController>>clickAt: act
320.        selectedActivity := selectedActivity == act ifTrue: [nil] ifFalse: [act].
321.        self changed: #selection.
```

7. 3 The BB4b>>frontload System Operation

We now enter the system behavior parts of the program. There is one section for each System Operation. This section is about the *frontloading* System Operation.

We first build a mental model of the operation. The computation of the early start of each activity is called *frontloading*. An activity can start as soon as all its predecessor activities are finished. The earliest possible start for any activity is the start of the project. So the algorithm in pseudocode for computing the early start for an activity is simply:

```
computeEarlyStartFrom (projectStart)
    Activity earlyStart = projectStart.
    for all Predecessors compute
        [Activity earlyStart = MAX (Activity earlyStart, Predecessor earlyFinish)].
```

This is the algorithm as seen from a single activity. We have to traverse all activities, but not in an arbitrary sequence. The algorithm has the constraint that the early finish must be known for all predecessors. We therefore need to take care when we select an activity for frontloading. The following selection algorithm does the trick:

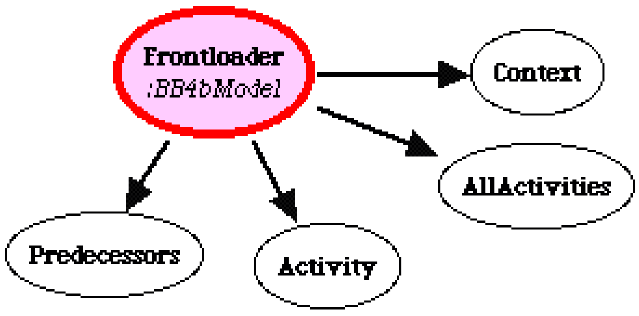
```
select Activity from all activities where
    Activity earlyStart is unknown
    AND all Predecessors earlyFinish are known
```

It is critical that the *earlyStart* and thus *earlyFinish* of all activities is unknown when we start the process. We must therefore remember to reset all activities at the start of the frontloading System Operation.

Based on these considerations, we have chosen a set of Roles that objects play when executing the frontload operation. The Context Diagram is shown in figure 25. The Roles are the Roles that come naturally from the above algorithm. In addition, we need access to the Context itself so that we can ask it to reselect a new Activity for the planning method.

A final decision is to name a *Frontloader* Role that will be responsible for driving the frontloading loop.

Figure 25: Context Diagram for the frontload operation.



Code in the Context perspective specify the Roles and their connectors, code in the Interaction perspective specify how they work. Each depends on the other, so it was a good idea to think about our mental model before we dived into the code.

We arbitrarily choose to code the Interaction first and then the Context.

7. 3. 1 BB4b frontload in the Interaction perspective

The *Frontloader*[1] Role is the only methodful Role in the Context and its Role Method implements the frontload algorithm:

```
Frontloader>>frontloadFrom: startWeek
    AllActivities do: [:act | act earlyStart: nil].
    [   Context reselectObjectsForRoles.
        Activity notNil
    ]   whileTrue:
        [   Activity earlyStart: startWeek.
            Predecessors do:
            [   :pred |
                (pred earlyFinish > Activity earlyStart)
                    ifTrue: [Activity earlyStart: pred earlyFinish + 1]
            ]
        ].
```

1.We underline most of the Role names to make this document more readable.

Comments:

line 323.	Reset model for frontloading.
line 324.	It is the responsibility of the *Context* to select an *Activity* object that satisfies the conditions.
line 326	The planning is a loop traversing all activities, it ends when no acceptable activity can be found. (*Activity* == nil)

The rest of the method is straightforward.

7. 3. 2 BB4b frontload in the Context perspective

We turn our attention to the *BB4bFrontloadCtx* Context class. Its class (static) methods specify the diagram shown in figure 25.

```
334.  BB4bFrontloadCtx class>>roleStructure
335.      ^ super roleStructure
336.          at: #Activity put: #();
337.          at: #Frontloader put: #(#Context #AllActivities #Activity #Predecessors );
338.          at: #Context put: #();
339.          at: #Predecessors put: #();
340.          at: #AllActivities put: #();
341.          yourself.
```

The Context class is responsible for processing System Operation requests. (Here called from the Controller code line 317 on page 43). This is where we decide to implement this operation with DCI and start the ball rolling in the *Frontloader* Role:

```
342.  BB4bFrontloadCtx class>>data: aData frontloadNetworkFrom: startWeek
343.      | ctx |
344.      (ctx := self new)
345.          data: aData;
346.          executeInContext: [(ctx at: #Frontloader) frontloadFrom: startWeek]
```

The object binding method that selects a planable activity is in the core of the frontload implementation:

```
347.  BB4bFrontloadCtx>>Activity
348.      ^ data allActivities
349.          detect:
350.              [:act |
351.              act earlyStart isNil
352.              and: [(data predecessorsOf: act) noneSatisfy: [:pred | pred earlyStart isNil]]]
353.          ifNone: [nil]
```

In simple words: An acceptable activity is an activity that is unplanned (line 351) and none of its predecessors is unplanned (line 352).

The other Role binding methods are straightforward:

```
354.  BB4bFrontloadCtx>>AllActivities
355.      ^data allActivities
```

line 355 The Model is the *data* in this implementation. It was set when the Context was instantiated in code line 317 on page 43.

```
356. BB4bFrontloadCtx>>Frontloader
357.     ^ data
```

The single Role Method beginning in code line 322 on page 44 does not reference *self* so it is independent of the class of the object it is bound to. The Controller class is one candidate; it picks up the user command and triggers the operation and could perform it directly. Another candidate is the Model. We have chosen the latter because the frontload operation is an operation that is defined i terms of the Model objects.

7. 4 The BB4b View perspective

There are three view classes in this application:

BB4bActivityView	This View presents a single model object in a trivial way. We therefore let the View have direct access to its model object.
BB4bDependencyView	This is a composite View, showing all activities with their dependencies as a graph. The View has no direct access to the model; we have chosen to let it delegate to a Context to perform any model-dependent operation. The Context for drawing the dependency graph is *BB4bDependencyCtx*.
BB4bGanttView	The same applies here. The Context is for drawing the Gantt diagram is *BB4bGanttCtx*

BB4bActivityView

We let an Activity View have direct access to its activity object in the model. It has also access to the Controller so that it can report any mouse activity for selection management.

```
358. RectangleMorph subclass: #BB4bActivityView
359.     instanceVariableNames: 'controller activity nameMorph'
360.     category: 'BB4bPlan-View'
```

An activity is selected or deselected when the user clicks its symbol. The selection logic must be in the Controller because it concerns all Views.

```
361. BB4bActivityView>>click: evt
362.     controller clickAt: activity.
```

line 362 The ActivityView delegates the handling of mouse clicks to the Controller. The controller interprets the mouse click as a selection and activates the Observer pattern in code line 319 on page 43. This view is an observer of the Controller and will be asked to update itself when the selection changes:

```
363. BB4bActivityView>>update: aParameter
364.     aParameter = #selection
365.     ifTrue:
366.         [self borderWidth: self borderWidth.
367.         self color: self color.
368.         self borderColor: self borderColor.
```

line 364 This view only reacts to *#selection* updates and ignores all other updates such as *#model* updates.

```
369.  BB4bActivityView>>borderColor
370.      ^ self isSelected
371.          ifTrue: [Color red]
372.          ifFalse: [Color black]

373.  BB4bActivityView>>borderWidth
374.      ^self isSelected
375.          ifTrue: [5]
376.          ifFalse: [2]

377.  BB4bActivityView>>isSelected
378.      ^ controller isSelected: activity
```

We considered programming the selection logic according to the DCI paradigm, but decided against it because the structure of the Controller and View objects is static and explicitly specified in the Controller class. This illustrates that there is no fixed rule; the use of DCI is a design decision.

BB4bDependencyView

The dependency view is the upper view in figure 23 on page 39.

```
379.  PasteUpMorph subclass: #BB4bDependencyView
380.      uses: BB4bDependencyCtxView
381.      instanceVariableNames: 'controller activityViews lines'
382.      category: 'BB4bPlan-View'
```

We see from code line 381 that this view does not have direct access to the Model. (But it can access it through the *controller*).

We will not document the code that instantiates the BB4bDependencyView and adds it to the window because this is regular Squeak code.

The key method is *update: #model* that triggers the creation of the dependency graph and its display:

```
383.  BB4bDependencyView>>update: aSymbol
384.      aSymbol = #model ifTrue: [self refresh].

385.  BB4bDependencyView>>refresh
386.      self deleteContents.
387.      BB4bDependencyCtx data: controller model refresh: self.
```

line 386	deletes (and removes from the View) any old activity symbols and dependency lines.
line 387	Requests the execution of this System Operation. The story continues in the Context perspective in code line 459 on page 50.

BB4bGanttView

The Gantt view is the lower view in figure 23 on page 39. It is similar to the DependencyView in that it accesses the model data through a Context, the *BB4bGanttCtx*.

```
388. PasteUpMorph subclass: #BB4bGanttView
389.     uses: BB4bGanttCtxGanttView
390.     instanceVariableNames: 'controller activityViews lines annotations'
391.     category: 'BB4bPlan-View'
```

line 390 the *lines* attribute holds the grid lines, the *annotations* attribute holds the value texts along the time axis.

```
392. BB4bGanttView>>update: aSymbol
393.     aSymbol = #model ifTrue: [self refresh].
```

```
394. BB4bGanttView>>refresh
395.     self deleteContents.
396.     BB4bGanttCtx data: controller model refresh: self.
```

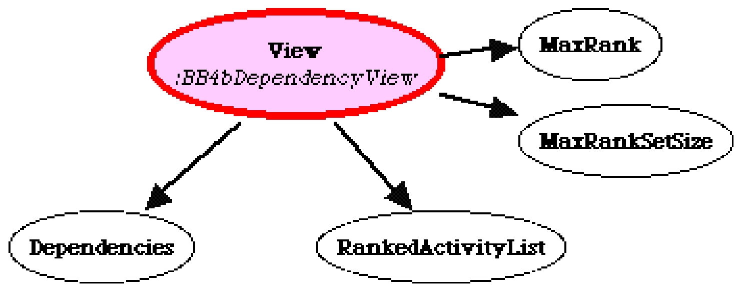
In line 396, we delegate the execution the *refresh* System Operation to *BB4bGanttCtx*. The story continues in code line 505 on page 52.

7. 5 The BB4DependencyView>>refresh System Operation

The dependency view is the top view in figure 23 on page 39. The automatic layout of a graph in two dimensions is far from trivial. We have chosen an oversimplified algorithm that serves as an example of how it helps readability to let a View access its Model through a Context. The algorithm is based on the concept of *rank*; the length of an activity's chain of predecessors. We position the activities in the horizontal direction according to rank and require our Context to deliver the activities sorted on rank. (This algorithm positions the activities nicely, but the dependency lines often overlap and cross activity symbols. This is clearly not acceptable for a real system.)

Figure 26 shows the Context Diagram. There is only one methodful Role, namely *View*. Its Role Methods are injected into the *BB4DependencyView* class.

Figure 26: The Roles needed for refreshing the DependencyView.



7. 5. 1 BB4bDependencyView>>refresh in the Interaction perspective

The Interaction is triggered with a *resetView* message from the Context (code line 464 on page 50). It is picked up here:

```
397. View>>resetView
398.     self addActivityViews.
399.     self addLines.
```

```
400. View>>addActivityViews
401.     | gridX gridY x0 y0 actViewExtent xPos yPos actView |
402.     gridX := self bounds width // MaxRank.
```

```
403.        gridY := self bounds height // MaxRankSetSize.
404.        x0 := self bounds left + 10.
405.        y0 := self bounds top + 10.
406.        actViewExtent :=  100 @ 40. "(gridX-50) @ (gridY-20)."
407.        1 to: RankedActivityList size do:
408.        [:rank |
409.            xPos := x0 + (gridX * (rank-1)).
410.            yPos := y0.
411.            (RankedActivityList at: rank) do:
412.            [:act |
413.                actView := self addActivityViewFor: act.
414.                actView bounds: ((xPos @ yPos) extent: actViewExtent).
415.                yPos := yPos + gridY.
416.            ]
417.        ].
```

line 402-line 406Compute grid.

line 407 Step horizontally on rank.

line 411 Step vertically

line 414 Position the added activity view.

```
418.    View>>addLines
419.        | fromView toView pt1 pt2 |
420.        Dependencies do:
421.        [:dep |
422.            fromView := self activiyViewAt: dep predecessor.
423.            toView := self activiyViewAt: dep successor.
424.            pt1 := fromView right @ ((fromView top + (fromView height // 2))).
425.            pt2 := toView left @ ((toView top + (toView height // 2))).
426.            self addLineFrom: pt1 to: pt2.
427.        ]
```

The above activity layout methods rest heavily on support methods in the *BB4bDependencyView* class. Examples are *addActivityViewFor: anActivity*, *activityViewAt: anActivity*, *addLineFrom: point1 to: point2*. All these methods are local to the class and cannot cause surprises during execution.

7. 5. 2 *BB4bDependencyView>>refresh* in the Context perspective

```
428.    BB1Context subclass: #BB4bDependencyCtx
429.        instanceVariableNames: 'view rankedActivities activityRanks'
430.        category: 'BB4bPlan-Context'
```

line 429 This context needs access to the View as well as *data* that is declared in the superclass. *rankedActivities* and *activityRanks* are caches used in some of the binding methods. They are computed in the beginning of the Role binding method:

```
431.    BB4bDependencyCtx>>reselectObjectsForRoles
432.        self computeRankedActivities.
433.        super reselectObjectsForRoles.
```

```
434.  BB4bDependencyCtx>>computeRankedActivities
435.      rankedActivities :=OrderedCollection new.
436.      activityRanks := Dictionary new.
437.      data allActivities do:
438.      [:act || rnk coll |
439.          rnk := self rankOf: act.
440.          activityRanks at: act put: rnk.
441.          coll := rankedActivities
442.              at: rnk
443.              ifAbsentPut: [SortedCollection sortBlock: [:x :y | x name < y name]].
444.          coll add: act
445.      ].
```

With this done, we are ready for the Role binding methods:

```
446.  BB4bDependencyCtx>>Dependencies
447.  ^ data allDependencies

448.  BB4bDependencyCtx>>View
449.      ^ view

450.  BB4bDependencyCtx>>MaxRank
451.      ^ rankedActivities size

452.  BB4bDependencyCtx>>MaxRankSetSize
453.      | maxSize |
454.      maxSize := 0.
455.      rankedActivities do: [:coll | maxSize := maxSize max: coll size].
456.      ^ maxSize

457.  BB4bDependencyCtx>>RankedActivityList
458.      ^ rankedActivities
```

line 458 This two-dimensional array is cached in the Context.

We pick up the refresh request from code line 387 on page 47.

```
459.  BB4bDependencyCtx class>>data: aData refresh: aView
460.      | ctx |
461.      (ctx := self new)
462.          data: aData;
463.          view: aView;
464.          executeInContext: [(ctx at: #View) resetView]
```

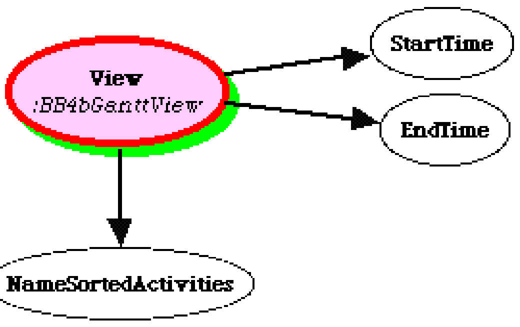
7. 6 The BB4bGanttView>>refresh operation

The Gantt view is the bottom view in figure 23. The layout is simple with time along the horizontal axis and activities along the vertical. There is one row for each activity.

Figure 27 shows the Context Diagram. There is only one methodful Role, namely *View*. *NameSortedActivities* refer to a collection of activities sorted by name.

Roles usually reference runtime objects, but they can also reference useful values. Here, the *StartTime* and *EndTime* Roles reference the start and end times of the project itself.

Figure 27: The Roles needed for resetting *BB4bGanttView*.



7. 6. 1 BB4bGanttView>>reset in the Interaction perspective

The Interaction is triggered with a *resetView* message from the Context sent in code line 510 on page 52. Resetting the view is in two parts: Draw the activity Views as horizontal bars in the right position and draw the grid lines with annotation:

```
465. View>>resetView
466.       self addActivityViews.
467.       self addLines.

468. View>>addActivityViews
469.       | currY maxX maxY gridX gridY x0 width actView |
470.       StartTime = EndTime ifTrue: [^self. "Network not planned. "].
471.             maxX := self width - 20.
472.       maxY := self height - 20.
473.       gridX := maxX // (EndTime - StartTime + 1).
474.       gridY := maxY // (NameSortedActivities size + 1).
475.       currY := 10.
476.       NameSortedActivities do:
477.       [:act |
478.             x0 := act earlyStart - StartTime * gridX + 10.
479.             width := (act earlyFinish - act earlyStart + 1) * gridX.
480.             actView := self addActivityViewFor: act.
481.             actView bounds: ((x0+self left) @ ((currY+self top) + 1) extent: width @ (gridY-2)).
482.             currY := currY + gridY
483.       ].

484. View>>addLines
485.       | maxX maxY gridX gridY y1 y2 y0 |
486.       maxX := self width - 20.
487.       maxY := self height - 20.
488.       gridX := maxX // (EndTime - StartTime + 1).
489.       gridY := maxY // (NameSortedActivities size + 1).
490.       y0 := self top + 10.
491.       y1 := NameSortedActivities size * gridY + self top + 20.
492.       y2 := self bottom - 10.
493.       self addLineFrom: (self left + 10) @ y1 to: (self right - 10) @ y1.
494.       0 to: EndTime - StartTime + 1 do:
495.             [:week || x |
496.             x := week * gridX + self left + 10.
497.             self addLineFrom: x @ y0 to: x @ y2.
498.             self
499.                addAnnotationFor: (StartTime + week) printString
500.                at: (gridX // 2 + x) @ (y1 + 10).
501.        ].
```

7. 6. 2 BB4bGanttView>>refresh in the Context perspective

```
502.  BB1Context subclass: #BB4bGanttCtx
503.        instanceVariableNames: 'view'
504.        category: 'BB4bPlan-Context'
```

The *refresh* System Operation was called from *BB4vGanttView*, code line 396 on page 48. We now trigger the Interaction in the *View* Role.

```
505.  BB4bGanttCtx class>>data: aData refresh: aView
506.        | ctx |
507.        (ctx := self new)
508.            data: aData;
509.            view: aView;
510.            executeInContext: [(ctx at: #View) resetView]
```

and the process continues in the Interaction, code line 465 on page 51.

The Role binding methods are straight forward:

```
511.  BB4bGanttCtx>>StartTime
512.        | time |
513.        time := nil.
514.        data allActivities do:
515.                [:act | time ifNil: [time := act earlyStart] ifNotNil: [time := act earlyStart min: time]].
516.        ^ time ifNil: [0] ifNotNil: [time]

517.  BB4bGanttCtx>>EndTime
518.        | time |
519.        time := nil.
520.        data allActivities
521.            do: [:act | time ifNil: [time := act earlyFinish] ifNotNil: [time := act earlyFinish max: time]].
522.        ^ time ifNil: [0] ifNotNil: [time]

523.  BB4bGanttCtx>>View
524.  ^ view

525.  BB4bGanttCtx>>NameSortedActivities
526.  ^ data allActivities asSortedCollection: [:x :y | x name < y name]
```

This concludes the *BB4bPlan* program documentation. A document is one-dimensional while the DCI paradigm in a multi-dimensional. We have added many cross references to facilitate reading, but it is much easier to read the code in an appropriate tool such as BabyIDE than in a linear document.

8 BB4aPlan: A Conventional Activity Network Planning Program

Application of the DCI paradigm to system behavior leads to readable programs through separation of concern. It is illuminating to compare the DCI-based implementation to a conventional implementation without DCI. Both of these implementations could have been written differently; but this is not a beauty contest. The examples are designed to help understand DCI and the nature of its separation of concern.

The conventional implementation, *BB4aPlan*, was written after the DCI version and the job proved a very simple one with extensive use of copy/paste. The two versions behave exactly the same; both result in the window shown in figure 23 on page 39. The code is also essentially identical; the main difference is in its organization.

The result is remarkable. Data classes are significantly simpler in the DCI version both as measured in lines of code and in coupling. System behavior logic was cleanly factored out in the DCI version; tightly coupled with basic code in the conventional version.

We will now look at the conventional version. There are, of course, no DCI classes in this version. The code in these classes was copy-pasted into the remaining, traditional classes. The result is summarized below.

8. 1 BB4a Data classes

Activity

```
Object subclass: #BB4aActivity
      instanceVariableNames: 'earlyStart duration name color'
…
```

This class is unchanged. Its only touch of system behavior is that it collaborates with the Controller regarding selection, and this is done with conventional programming style in both versions.

Dependency

```
Object subclass: #BB4aDependency
      instanceVariableNames: 'predecessor successor'
…
```

No change.

Model

```
Object subclass: #BB4aModel
      instanceVariableNames: 'activities dependencies rankedActivities activityRanks'
```

The instance variables *rankedActivities* and *activityRanks* had to be added. They stem from *BB4bDependencyCtx* where they were caches used by the algorithm for sorting the model activities according to rank. They are used in two methods; *computeRankedActivities* and *rankOf: act*. This is an example of state (instance) variables and methods that have nothing to do with the

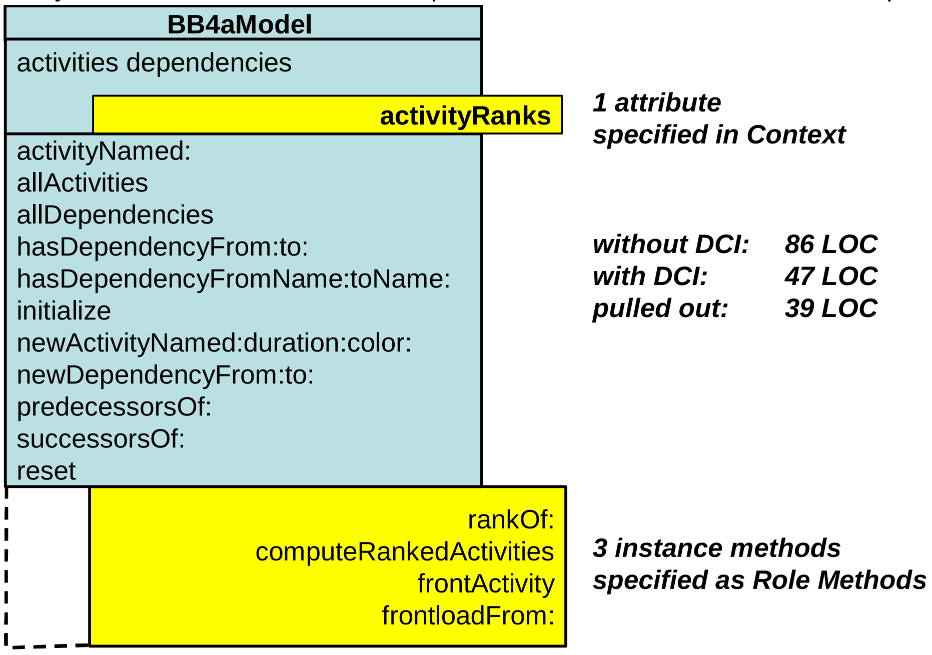
Model object as such; they are only meaningful during the execution of a particular System Operation.

The Model class has also been burdened with the methods that implement the *frontload* System Operation. They used to be in the *BB4bFrontloadCtx* and in the Role Methods that were injected from its *Frontloader* Role. (Compare with the DCI Role Method in code line 322 on page 44):

```
535. BB4aModel>>frontloadFrom: startWeek
536.     | frontAct |
537.     self allActivities do: [:act | act earlyStart: nil].
538.     [frontAct := self frontActivity. frontAct notNil]
539.     whileTrue:
540.         [frontAct earlyStart: startWeek.
541.         (self predecessorsOf: frontAct) do:
542.             [:pred |
543.             (pred earlyFinish > frontAct earlyStart)
544.                 ifTrue: [frontAct earlyStart: pred earlyFinish + 1]].
545.         ].

546. BB4aModel>>frontActivity
547.     ^self allActivities
548.         detect:
549.             [:act |
550.             act earlyStart isNil
551.             and:
552.                 [(self predecessorsOf: act) noneSatisfy: [:pred | pred earlyStart isNil]]]
553.         ifNone: [nil]
```

Figure 28: System behavior methods were specified in the Context and Interaction perspectives.



This illustrates an important characteristic of DCI; Data classes are not encumbered with instance variables and methods that are private to particular system behaviors. The difference is striking even in this very simple example. Figure 28 illustrates how DCI distinguishes between code that specifies what an object *is* from what an object *does* in collaboration with other objects.

The *BB4bModel* class source code in the DCI version is 47 lines of code without injected methods. Compare with the conventional version, *BB4aModel,* which is 86 lines. This is an increase of 39 lines and we still haven’t done backloading and resource allocation and other operations that will be parts of a real planning system.

8. 2 The BB4a Controller class

```
SystemWindow subclass: #BB4aController
    instanceVariableNames: 'dependencyView ganttView selectedActivity'
…
```

The class definition is unchanged and there are no additional methods. The Model is now responsible for frontoading:

```
BB4aController>>frontloadDemo
    model ifNil: [self inform: 'Define the model before frontloading. \Command ignored.' withCRs. ^self].
    model frontloadNetworkFrom: 1.
    self changed: #model.
```

8. 3 The BB4a View classes

ActivityView

The ActivityView is unchanged; it does not participate in System Operations:

```
RectangleMorph subclass: #BB4aActivityView
    instanceVariableNames: 'controller activity nameMorph'
…
```

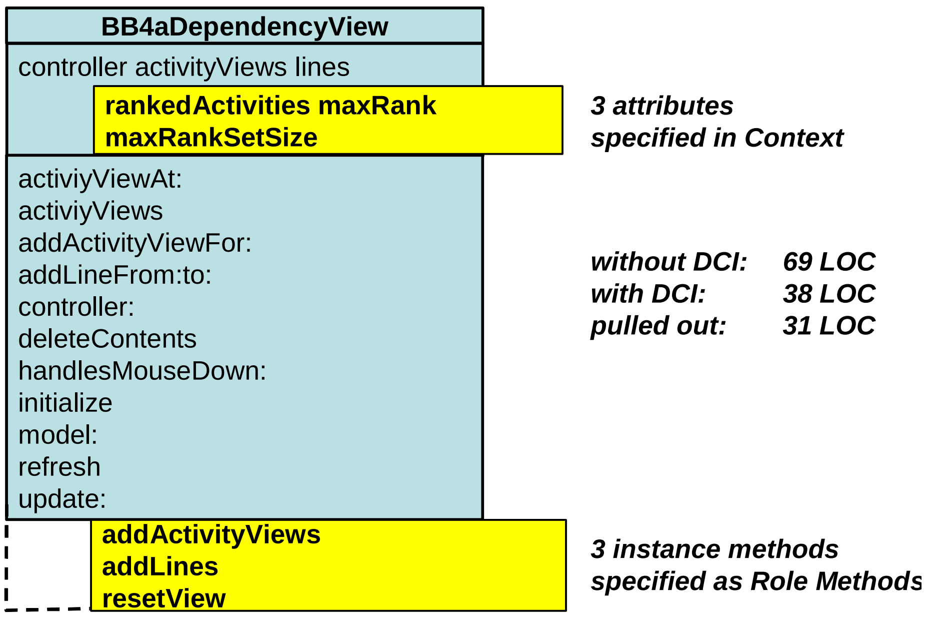
DependencyView

The conventional *DependencyView* cannot depend on a Context to serve up the model data in an appetizing form and must do everything itself.

```
PasteUpMorph subclass: #BB4aDependencyView
    instanceVariableNames:
            'controller activityViews lines rankedActivities maxRank maxRankSetSize'
    category: 'BB4aPlan-View'
```

The spurious instance variables *rankedActivities*, *maxRank*, and *maxRankSetSize* are all incidental to the layout algorithm and irrelevant to the *DependencyView* as such. The layout algorithm was very weak in this demo program and is likely to be changed. This will entail redefinition of the instance variables in the conventional version, while it will be local to the Context with its Roles and Role Methods in the DCI version. A significant difference.

Figure 29: DCI moves system behavior methods to Context and Interaction perspectives.



The BB4aDependencyView class contains instance variables and methods that were pulled out in the DCI version. The result is that the code for the conventional class is 31 lines longer than the DCI version. This is an increase of 80% - and this is for a single System Operation. No wonder conventional code can be so cluttered as to be unreadable. (The additional lines were in both versions of the class, but they were injected from the Methodful Roles in the DCI version so that they were easily be filtered out in the browser).

GanttView

We see similar differences in the *GanttView* class:

```
PasteUpMorph subclass: #BB4aGanttView
    instanceVariableNames:
        'controller activityViews lines annotations endTime startTime nameSortedActivities
    category: 'BB4aPlan-View'
```

The additional instance variables are *endTime*, *startTime*, an *nameSortedActivities*. Role Methods and other system behavior related methods makes the GanttView class grow from 39 lines to 83 lines. An increase of 82%.

9 Support for Programming with Roles in Squeak

Computer system processes create value when an end user apply them for profitable tasks. The processes we are concerned with take place in networks of communicating objects. The networks are ephemeral; they arise spontaneously and depend on the momentary state of the system and the current task.

Code is conventionally expresses in terms of classes and the structures found in class hierarchies. Compile time, class based code cannot, in general, reveal everything about the runtime value-creating processes.

A system can perform many different tasks. Each task is executed by a Context that establishes a network of interlinked objects. The Context is instantiated at the beginning of a task and looses its utility when the task is completed. We say that the Context instance is a dynamic namespace because it is created to support an execution.

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

The DCI paradigm impose a restriction on the allowable networks of communicating objects by insisting that all networks that realize a given task shall share a common topology. This makes it possible to create a compile time, static structure that describes all the ephemeral runtime networks.

We use two mechanisms to make this scheme work. Firstly, we describe the topology as a structure where the nodes are called *Roles* and the edges are called *Connectors*. We can then write the code that controls object communication in terms of these Roles and Connectors. Secondly, we need to bind the compile time Roles to the objects that actually communicate at runtime.

The BabyIDE1 Squeak programming environment includes support for programming with Roles. We will discuss the support classes in this section. The support classes are:

Object subclass: #BB1Context
The main DCI support class. See *section 9. 1. 1*.

Trait subclass: #BB1RoleTrait
Extends Trait with a reference to the Context class.

Object subclass: #BB1ReferenceClass
We need to have class references that are richer than the plain class name.

Object subclass: #BB1ReferenceSelector
We need to message references that are richer than the plain message selector.

VariableNode subclass: #BB1RoleNode
Part of the compiler extension. See section 9. 3 on page 61.

In addition, there are a few subclasses of Squeak library classes that adapt the library classes to the needs of BabyIDE1:

ContextVariablesInspector subclass: #BB1ContextVariablesInspector
Adds current RoleNames and Role Values to list of candidates for inspection.

Object subclass: #BB1Iterator
Part of an unfinished experiment.

PluggableButtonMorph subclass: #BB1PluggableButtonMorph
PluggableButtonMorph needs to identify a button when it is pushed.

PluggableTextMorph subclass: #BB1PluggableTextMorph
We need to control fonts and to get at the internal TextMorph.

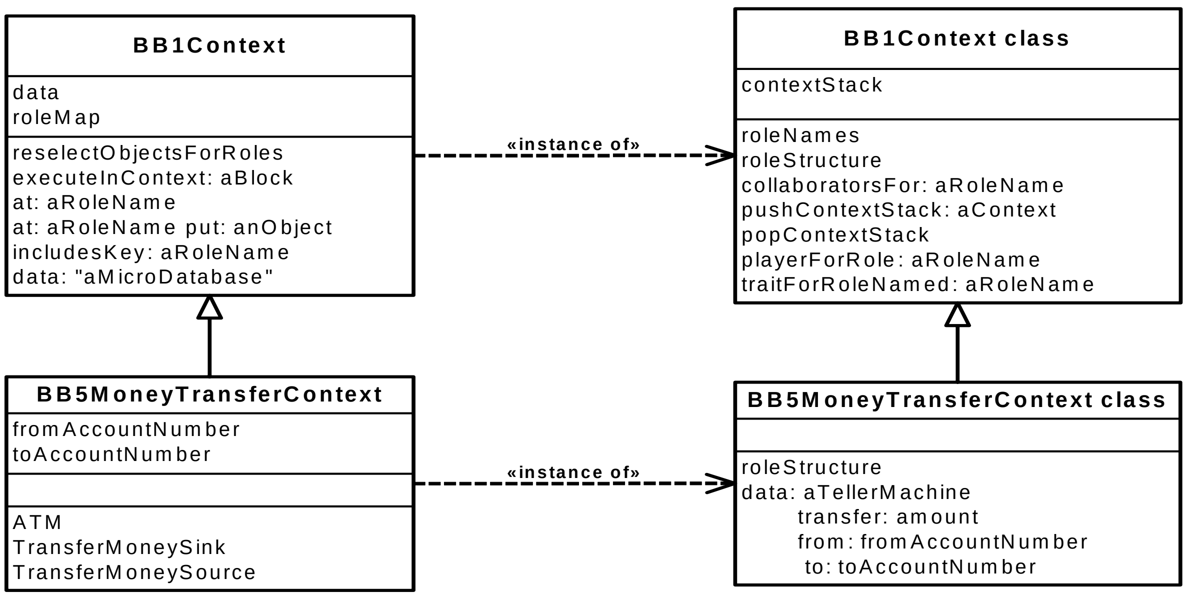
There are also some method changes in the compiler. They are described in section 9. 3 on page 61.

9. 1 The BB1Context class

The main support class is the *BB1Context* class. Its static, metaclass side specifies the network topology as a *Context Diagram* with Roles and Connectors. The static, metaclass side of its subclasses also define factory methods for the System Operations that are implemented in the Context. This class side is modeled to the right in the UML class diagram in figure 30

The instance side of *BB1Context* and its subclasses is a dynamic namespace that binds Roles to objects and that hold any variables that are private to the Interaction methods. This instance side is shown to the left in the UML class diagram in figure 30.

Figure 30: *BB1Context* class diagram with an example subclass.



9. 1. 1 Binding Roles to objects at runtime

Compile time code is written in terms of interconnected Roles, runtime processes take place in networks of communicating objects. We therefore need to bind the Roles to the objects that actually do the work. The binding is done in the Context instance, let’s call it *ctx* for the time being. Binding from Role to object is like a dictionary lookup:

```
570. ATM>>transfer (amount)
571.     (ctx playerForRole: #sourceAccount) withdraw amount
572.     (ctx playerForRole: #destinationAccount) deposit amount
```

This leaves two open questions. The first is to find the Context instance, *ctx*. The other is to set up the binding dictionary in the Context.

It would also be nice if we could replace the clumsy *(ctx playerForRole: #sourceAccount)* with the Role name itself, *SourceAccount*. This entails hacking the Squeak compiler; the hacks are described in section 9. 3 on page 61.

9. 2 Finding the Context instance

A Context is instantiated at the beginning of a task and is disbanded at its completion. Conceptually, the Context could live on the stack where it would be available to all method executions during the performance of the task. We are currently only considering sequential execution so subtasks can be executed within subcontexts.

A simpler mechanism is to let every Context class have a stack for keeping its active instances. This metavariable is defined in superclass for all Contexts, *BB1Context*:

```
573.  BB1Context class
574.      instanceVariableNames: 'contextStack'
```

line 574 *contextStack* is specified in the metaclass so that every subclass of *BB1Context* gets its own instance variable. (Just as every instance of regular class gets its own set of instance variables.)

The *contextStack* is handled by two simple methods:

```
575.  BB1Context class>>pushContextStack: aContext
576.      contextStack addLast: aContext.

577.  BB1Context class>>popContextStack
578.      contextStack removeLast
```

The runtime Role binding lookup is simple and fast:

```
579.  BB1Context class>>playerForRole: roleName
580.      ^ contextStack last at: roleName
```

Tasks are executed within a Context instance so that Roles can be bound to objects during the execution. This gets us to the instance side of the *BB1Context* class:

```
581.  Object subclass: #BB1Context
582.      instanceVariableNames: 'data roleMap'
583.      category: 'BB1IDE-Support'
```

line 582 All Contexts have a *data* variable where the Role binding methods can find their objects. Some Contexts have additional variables as shown in the Bank example below.
roleMap is the *Role -> object* binding dictionary.

The *roleMap* Dictionary is filled with an entry for each Role:

```
584.  BB1Context>>reselectObjectsForRoles
585.      | messName |
586.      roleMap := IdentityDictionary new.
587.      self class roleNames
588.      do:
589.          [:roleName |
590.          roleMap at: roleName put: (self perform: roleName ifNotUnderstood: [nil])].
```

line 590 The Context has a method for every Role. This method has the same name as the Role and returns the object that at is currently playing the Role. This line calls the method and puts the resulting value into the *roleMap* dictionary.

The Roles are bound to objects at the beginning of an execution:

```
591. BB1Context>>executeInContext: aBlock
592.     self reselectObjectsForRoles.
593.     [   self class pushContextStack: self.
594.         aBlock value]
595.     ensure: [self class popContextStack].
```

line 592 Build the namespace dictionary by executing the Role Binding Methods.

line 593 Push this Context on the Context stack in the class, see code line 575.

line 594 Let the bound objects perform the task.

line 595 This line ensures that the Context stack will be popped even if the execution terminated on an error.

Let’s take the Bank Transfer from section 4 on page 20 as an example:

```
596. BB1Context subclass: #BB5MoneyTransferContext
597.     instanceVariableNames: 'fromAccountNumber toAccountNumber'
598.     category: 'BB5Bank-Context'
```

line 597 The Context holds variables that are private to the execution.In addition, the superclass *data* variable is a reference to the Teller machine.

We saw in code line 571 and code line 572 on page 58 that methods access the Roleplaying objects with the message *playerForRole:* to the Context class.

```
599. BB1Context class>>playerForRole: roleName
600.     ^ contextStack last at: roleName

601. BB1Context>>at: roleName
602.     ^ self
603.         at: roleName
604.         ifAbsent:
605.             [self error: roleName , ' is not defined as a role in this Context.'].

606. BB1Context>>at: roleName ifAbsent: absentBlock
607.     | value |
608.     value := roleMap
609.                 at: roleName
610.                 ifAbsent: [absentBlock].
611.     value == self symbolForLazyBinding
612.         ifTrue: [value := self perform: roleName.
613.             roleMap at: roleName put: value].
614.     (value isBlock
615.             and: [value numArgs = 0])
616.         ifTrue: [^ value value].
617.     ^ value
```

Some fancy mechanisms for future exploration here, but this method is currently used as a simple dictionary lookup.

The money transfer operation is triggered by a message to the *BB5MoneyTransferContext* class. We use it as an example of the BB1Context subclasses:

```
618.  BB5MoneyTransferContext class>>
                  data: tellerMachine
                  transfer: amount
                  from: fromAccountNumber
                  to: toAccountNumber
619.      | ctx |
620.      (ctx := self new)
621.          data: tellerMachine;
622.          fromAccountNumber: fromAccountNumber;
623.          toAccountNumber: toAccountNumber;
624.          executeInContext: [(ctx at: #ATM) transferAmount: amount].
```

line 624 The Context instance is put on the execution stack when we start a System Operation, see code line 591. This Context will be available to all methods that are activated during the execution irrespective of actual objects or classes. The Context will be popped from the stack at the completion of the operation.

9. 3 Use Role names in code

We would like to write code such as[1]:

```
625.  ATM>>transfer: amount
626.      SourceAccount withdraw: amount.
627.      DestinationAccount deposit: amount.
```

The Compiler must then expand a Role reference such as *SourceAccount* to the Role name in the expression *(ctx playerForRole: #SourceAccount)*. This has been done by hacking a method in the *Encoder* class in the Squeak compiler.

```
628.  Encoder>>init: aClass context: aContext notifying: req
629.      | node n homeNode indexNode |
630.      requestor := req.
631.      class := aClass.
632.      nTemps := 0.
633.      supered := false.
634.      self initScopeAndLiteralTables.
635.      n := -1.
636.      ((class isKindOf: BB1RoleTrait) and: [class roleContextClass notNil])
637.      ifTrue:
638.          [(class roleContextClass
                  collaboratorsFor: (class roleContextClass roleNameFromTraitName: class name)) do:
639.              [:roleName |
640.                  self scopeTableAt: roleName
                          put: (BB1RoleNode new
                                  asVariable: roleName
                                  contextName: class roleContextClassName)
641.              ]
642.          ].
643.      "Here follows old code for other kinds of variables"
```

line 636 Here is an addition that treats *BB1RoleTraits* specially, but only if the RoleTrait has a link to its Context.

1.We underline Role names in running text to simplify reading; we capitalize Role names in Squeak to distinguish them from regular variable names and method selectors.

line 638	Add a variable node for every Role that is visible from the current Role.
line 640	The variable node is an instance of *BB1RoleNode*, one of our support classes.

The *BB1RoleNode* holds the data it needs for generating the required Context lookup code:

```
644.  VariableNode subclass: #BB1RoleNode
645.        instanceVariableNames: 'receiver arguments selector'
646.        category: 'BB1IDE-Support'

647.  BB1RoleNode>>asVariable: roleName contextName: ctxNam
648.        | arg1 |
649.        comment := nil.
650.        receiver := VariableNode new
651.                    name: 'self'
652.                    key: ctxNam -> (Smalltalk at: ctxNam)
653.                    code: -4.
654.        selector := SelectorNode new comment: nil;
655.                     key: #playerForRole: code: -5.
656.        arg1 := LiteralVariableNode new.
657.        arg1 key: roleName asSymbol code: LdLitType negated.
658.        arg1 name: roleName.
659.        arguments := OrderedCollection with: arg1
```

This is getting pretty deep into obscure-land. It generates the required message when the receiver (the Context class name) is declared as in line 652 and the message selector is *#playerForRole*: as declared in line 655 and the Role name is declared as in line 657. This solution seems to work both during execution and in debuggers. This is as far I have dived into the Compiler complex. The solution may be inelegant and it may hide unpleasant bugs. But it seems to serve its purpose in BabyIDE1, and that's all that is required for the time being.

9. 4 Methodful Roles

A powerful DCI mechanism is the Methodful Roles. It includes the capability to define Role Methods, i.e. methods that are associated with a Role and are injected into the Role Playing Classes.

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

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

Vanilla Traits are stateless methods that are always executed in the context of a class. A Trait Method can access its instances through messages to *self*.

A methodful Role can be created by associating a Role with a Trait. The trait methods are injected into all classes that implement the Role. Role Methods may send messages to *self* so

1.Schärli, N; Ducasse, S; Nierstrasz, O; Black, A.; "*Traits: Composable Units of Behavior*," Proc. ECOOP'03, LNCS, vol. 2743; Springer Verlag, July 2003, pp. 248—274. [DOI] 10.1007/b11832 [web page] http://www.iam.unibe.ch/~scg/Archive/Papers/Scha03aTraits.pdf that the Role playing objects can do different things depending on their class. The Role Methods suspends polymorphism at the Interaction level and ensures that the Interaction is under full control. A Role can be realized by several Role Playing classes. This permits the variability provided by polymorphism at the local level.

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

Our Trait subclass, *BB1RoleTrait*, binds the Trait to a Context so that the Trait compiler can compile methods that reference Roles as described in section 9. 3 above.

10 The BabyIDE1 implementation

BabyIDE1 has been created through exploratory programming. The code is incomplete, unreadable and probably buggy. The sooner it is replaced by a *BabyIDE2* that is properly designed and coded the better.

In spite of all its flaws, *BabyIDE* has two important characteristics to recommend it. It exists, and it illustrates what programming according to the DCI paradigm is all about.

11 Conclusion

It started with a dream; *let me make my programs so simple that there are obviously no deficiencies*. I found that my "object oriented" programs did not reveal everything about how the system worked. I simply refused to continue writing unreadable programs. The present wasn't good enough and I had to do something about it. I started the BabyUML project with the conviction that object oriented programs could and should be readable.

The background for the project name was somewhat whimsically as follows:

Baby: The world's first digital stored program computer was the *Manchester Small Scale Experimental Machine—"The Baby"*. This Baby was small; it was a truly minimal computer with an operations repertoire of just 7 instructions. It executed its first program on 21st June 1948. The first truly object oriented program was written using the BabyIDE tool exactly fifty years later. "The Baby" was insignificant in itself, but it marked the beginning of a new era. The first programs written with BabyIDE and the DCI paradigm are likewise insignificant …

UML. The UML part of the name meant that I expected to adapt many concepts from UML. It didn't turn out that way, and my new project is called *BabyIDE* to stress that the focus has been shifted from proof of concept to tools for practical programmers.

I have extended my universe of discourse to separate code for system state from code for system behavior. I have augmented the old class oriented code with readable, object oriented code.

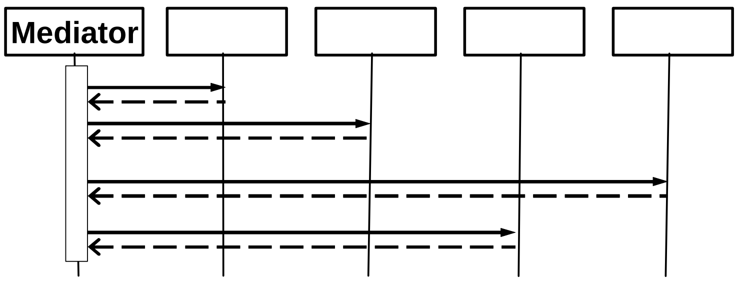
The process visualized in the *ArrowsAnimation* program is a challenge to conventional OO programming because it involves a network of communicating objects where the class and identity of objects occupying a given node in the network varies over time.

One recommended programming style is the Mediator pattern:

> *"Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."*[GOF-95]

The process visualized in the *ArrowsAnimation* could be implemented as a Mediator class. All the animation logic would be in this class; star, circle, and arrow classes would be pure state holders with no knowledge of the animation. A sequence diagram modeling the process is shown in figure 31.

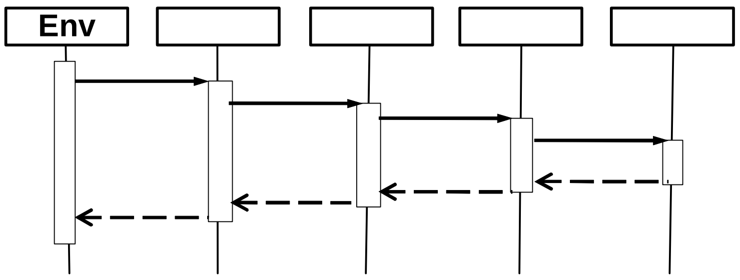
Figure 31: Extreme centralization. The Mediator pattern.



The Mediator class specifies all object communication; there is no peer-to-peer collaboration. The Mediator class gets very complex if the communication pattern is complex.

Extreme decentralization is too chaotic, extreme centralization is too rigid. DCI leads to distributed logic under full control. The sequence diagram in figure 32 illustrates how DCI supports distributed interaction logic. There is an important restriction. The methods shown as narrow, vertical rectangles in the diagram are *Role Methods*; they are shared by the classes of all objects that can play the Roles.

Figure 32: Sequence diagram illustrating DCI distributed logic.



Is DCI a procedure oriented paradigm in disguise? My answer is no. Both state and behavior is here distributed among the objects. I see DCI as even more object oriented than regular OO. In DCI, we work with networks of collaborating objects; in regular OO, we work with classes and can only see one object at the time.

The current state of the BabyIDE project is that an alpha version ofBabyIDE1 can be downloaded from SqueakMap and the toy examples can be downloaded as changes files from the Downloads section in my DCI Home Page:

http://heim.ifi.uio.no/~trygver/themes/babyide/babyide-index.html.

The results we have seen so far are promising. We have observed that the DCI implementation of the *BabyShapes* animation program is radically more readable than the many versions preceding it. The two implementations of the Network Activity Example show that the explicit code for system behavior that is achieved with DCI is clearly more readable than conventional class oriented code.

Random highlights:

- System state and behavior is at least as important as object state and behavior.
- A Context captures what is common between networks of communicating objects that realize the same System Operation in different executions.
- We only permit runtime processes where the networks of communicating objects have a common topology.
- We suspend polymorphism for methods that are essential for the integrity of an Interaction.
- An important feature of the DCI style of programming is that the Data classes do not define the runtime object structure.

I look forward to see more programs written according to the DCI paradigm and am confident that they will confirm that their code will indeed reveal everything about how they will work.

Promising work is being done on implementations of DCI in other languages such as C++, Scala, various extensions of Java (Composite Oriented Programming/Qi4j), Ruby, Phyton… A crucial next step could be to implement IDEs for these languages to facilitate the development of real applications in these languages.

An e-mail list, object-composition@googlegroups.com, is a meeting place for people interested in object composition in general and DCI in particular.

The DCI paradigm outlaws many working programs as was illustrated in figure 1 on page 4. I will miss many cherished programming constructs that I can no longer use since they lead to unreadable programs.

I take comfort from history. Some time in the sixties, Dijkstra considered the GOTO statement harmful. I believed him and got very depressed. I just couldn't see how I could write programs without using the very statement that gave programming its real power. Yet, I painfully changed my way of thinking and got to like GOTO-less programs better than the spaghetti code of old. I was introduced to Smalltalk some years later and was asked if I missed the GOTO statement. I hadn't noticed that Smalltalk had no GOTO!

My conclusion is that while hardened programmer may find it strange to extend their attention with runtime behavior, I am confident they will find such a shift very profitable and well worth the effort because the resulting code can be audited before it is tested and understood by maintainers.

The BabyUML/BabyIDE projects have created what may be the world's first integrated development environment based on a truly object oriented programming paradigm (Simula, Smalltalk, C++, Java, and others are based on the class paradigm. Even *self* code describes one object at the time; there are no facilities for describing networks of collaborating objects). The result of the BabyUML project was like a new born baby. Its functionality is extremely limited and it may not be able to stand on its own two feet, but there is room for almost unlimited growth. My dream is that many people will adopt the Baby ideas and create their own vigorous variants.

12 Further work

BabyIDE1 is experimental and there are many things that still need to be done. These things range from the trivial to the profound:

Completion:	I have used *BabyIDE1* to write a few toy programs, most of them are documented in this report. I want to continue this work; the goal is to stabilize BabyIDE1 and the Squeak DCI infrastructure to a point where it makes sense to write valuable programs with DCI.
Semantic model:	What is a program? We need a precise definition of a DCI-conformant program. It could, for example, be in the form of a UML class diagram.
BabyIDE2:	BabyIDE1 should be re-implemented according to the DCI paradigm. This implementation should be based on the above semantic model.
Inheritance:	OOram has the notion of *role model synthesis* for combining roles models. UML has the somewhat fuzzy concept of *package merge* for flattening models. BabyIDE needs a similar function for merging DCI code. This looks like a good theme for a doctoral thesis. It could start from Egil Andersen's work on role model synthesis.[Andersen-97]
Enumeration:	The enumeration of collections pose an interesting problem. Current element in an enumeration should probably be visible as a Role in the Context Diagram. The collection itself should probably also be visible as a Role. We would then need notation for showing the element Role as being contained in the collection Role …
Dynamic binding:	The current implementation binds Roles to objects with the *Context>>reselectObjectsForRoles* method. Other schemes are possible, e.g. that a Role triggers the execution of the binding method whenever it is referenced. Early experiments with this solution led to code that was perfectly logical but far too lively for comfort. An in-depth discussion of the binding issue should prove very interesting. A Master thesis?
Multi-threading:	*BabyIDE* is for sequential programming. What about multi-threading?
Textbook:	Write a basic textbook on programming. This could be truly object oriented and cover the whole spectrum from system state to system behavior.
Platforms:	Create a BabyIDE for a mainstream platform such as C++.

My hope is that my *BabyIDE* implementation shall inspire programmers, developers, and researchers to pick up the baton and run with it. Personally, I will work hard at applying DCI to various programming tasks, modifying the Squeak *BabyIDE* as required.

13 Acknowledgements

The work that has led to the DCI paradigm and the BabyIDE has taken many years of lonesome ups and downs. I could not have stayed the distance if hadn't been for the encouragement I received from men I deeply respect, the foremost being Dave Thomas and Bran Selic.

Also, I haven't been as lonesome as all that. The group for *Cooperative and Trusted Systems* at SINTEF in Oslo and the group for *Object orientation, Modeling and Language* at the Department of Informatics, University of Oslo have both been supportive sparring partners.

The BabyIDE implementation rests heavily on Traits. My sincere thanks to Nathanael Schärli, Stéphane Ducasse, Andrew Black, and Adrian Lienhard for providing this very powerful extension of the Squeak class paradigm.

Last, but not least, I thank my friend Jim Coplien for innumerable discussions over the years. Our common ground has been our focus on people. The value of a system is its value for its users. Users can be the end users of an application or its developers using a programming environment. We have been following our separate paths when searching for a common truth we both believed must be out there somewhere. At long last we have joined forces to cooperate along a common track that we both believe is leading to something very important.

14 References.

[Andersen-97]	Andersen, E. P.: *Conceptual Modeling of Objects. A Role Modeling Approach.*; Dr.Scient thesis, November 1997, University of Oslo. [web page] http://heim.ifi.uio.no/~trygver/1997/EgilAndersen/ConceptualModelingOO.pdf
[AOP]	Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J; *Aspect-Oriented Programming*; In Proc. of ECOOP 1997.
[BabyUML-06]	Reenskaug, T.; *Expert' voice: The BabyUML discipline of programming*; Software & System Modeling (2006) 5(1): 3–12; DOI 10.1007/s10270-006-0005-0 (?); Springer Berlin / Heidelberg; ISSN 1619-1366 (Print) 1619-1374 (Online); pp. 1-107. [web page] http://www.springerlink.com/content/59v42qw781g5k075/fulltext.pdf. also at [web page] http://heim.ifi.uio.no/~trygver/2006/SoSyM/trygveDiscipline.pdf
[BabyUML-07]	Reenskaug, T;. *Programming with Roles and Classes: the BabyUML Approach;* In Klein, Ari D.; *Computer Software Engineering Research*; ISBN-13: 978-1-60021-774-6; Nova Publishers; Hauppauge NY, 2007; pp 45-88; [web page] http://folk.uio.no/trygver/2007/babyUML.pdf
[Coplien98]	Coplien, James: *Multi Paradigm Design for C++*, Addison-Wesley Professional, 1998, ISBN: 0-201-82467-1
[Dijkstra-68]	Edsger Dijkstra; *Go To Statement Considered Harmful*; CACM **11** (3) (March 1968); pp147–148.
[Engelbart-62]	Engelbart, D., C; AUGMENTING HUMAN INTELLECT: A Conceptual Framework; Stanford Research Institute report no. AFOSR-3233; Menlo Park, California, 1962; [web page] http://www.invisiblerevolution.net/engelbart/full_62_paper_augm_hum_int.html
[GOF-95]	Gamma, E; Helm, R; Johonson, R; Vlissides, J: *Design Patterns*; ISBN 0-201-63361-; Addison-Wesley, Reading, MA. 1995.
[Hoare-81]	Hoare, C. A. R.: *The Emperor's Old Clothes* 1980 Turing Award lecture; Comm.ACM vol24-81, 2 (Feb. 1981)
[ISO-66]	*IFIP-ICC Vocabulary of Information Processing*; North-Holland, Amsterdam, Holland. 1966; p. A1-A6.
[MVC]	Reenskaug, T.; *The original MVC reports*; [web page] http://www.duo.uio.no/sok/work.html?WORKID=52648 Reenskaug, T.; *The Model-View-Controller (MVC). Its Past and Present.* Dept. of Informatics, University of Oslo; August 2003; [web page] http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf
[NIAM]	Halpin, T., Morgan, T.; *Information Modeling and Relational Databases*; Elsevier 2001; ISBN 1558606726, 9781558606722
[ODMG]	The ODMG 3.0 standard is being revised by OMG. See [web page] http://www.odbms.org/odmg.html
[OOram]	Reenskaug et.al.: *Working with objects. The OOram Software Engineering Method.*Manning 1996; ISBN 1-884777-10-4. Draft version at http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects.pdf

[Readable]	Reenskaug, T;. *The Case for Readable Code;* In Klein, Ari D.; *Computer Software Engineering Research*; ISBN-13: 978-1-60021-774-6; Nova Publishers; Hauppauge NY, 2007; pp 3-8; [web page] http://heim.ifi.uio.no/~trygver/2007/readability.pdf
[Schärli]	See [web page] http://www.iam.unibe.ch/~scg/Research/Traits/ Schärli, N; Nierstrasz, O; Ducasse, S; Wuyts, R; Black, A; "*Traits: The Formal Model,*" Technical Report, no. IAM-02-006, Institut für Informatik, November 2002, Technical Report, Universität Bern, Switzerland, Also available as Technical Report CSE-02-013, OGI School of Science & Engineering, Beaverton, Oregon, USA. [WEB PAGE] http://www.iam.unibe.ch/~scg/Archive/Papers/Scha02cTraitsModel.pdf Schärli, N; Ducasse, S; Nierstrasz, O; Black, A;"*Traits: Composable Units of Behavior,*" Proc. ECOOP'03, LNCS, vol. 2743; Springer Verlag, July 2003, pp. 248—274. [DOI] 10.1007/b11832 [web page] http://www.iam.unibe.ch/~scg/Archive/Papers/Scha03aTraits.pdf
[Smalltalk]	Goldberg, A; Robson, D; *Smalltalk-80. The Language and its Implementation.* Addison-Wesley, Reading, Mass 1983; ISBN 0-201-11371-6
[Squeak]	Home page: http://www.squeak.org/
[UML]	*Unified Modeling Language: Superstructure.* Version 2.1.2. Object Management Group (OMG); formal/2007-11-02; November 2007; [web page] http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/
[Webster-08]	Merriam-Webster Open Dictionary. [web page] http://www.merriam-webster.com/dictionary/
[Wikipedia]	*WikipediA, the free enceclopedia*. [web page] http://en.wikipedia.org/wiki/Main_Page

Trygve Reenskaug is professor emeritus of informatics at the University of Oslo. He has 50 years experience in software engineering research and the development of industrial strength software products. He has extensive teaching and speaking experience including keynotes, talks and tutorials. His firsts include the Autokon system for computer aided design of ships with end user programming language, structured programming, and a data base oriented architecture from 1960; object oriented applications and role modeling from 1973; Model-View-Controller, the world's first reusable object oriented framework, from 1979; OOram role modeling method and tool from 1983. Trygve was a member of the UML Core Team. The goal of his current research is to create a new, high level discipline of programming that lets us reclaim the mastery of our software.

Appendix 1: Baby Terminology

Baby	Prefix for the names of artifacts produced in the BabyUML and BabyIDE projects.
BabyIDE	An Interactive Development Environment for developing programs that are structured according to the DCI paradigm. Also the name of a project for the evolution DCI and associated IDEs.
BB2Shapes	A Squeak animation program that visualizes an example of rapidly changing networks of communicating objects.
BB4aPlan	An activity network planning application coded without DCI.
BB4bPlan	An activity network planning application coded with DCI.
BB5Bank	A very simple application for transferring funds from one bank account to another.
BabyUML	A project that aimed at a new, system-oriented discipline of programming where code shall explicitly specify system behavior as well as system state. This project has reached it goal and is terminated. Work continues in the BabyIDE project.
Class	"*In object-oriented programming, a class is a programming language construct that is used as a blueprint to create objects. This blueprint includes attributes and methods that the created objects all share*".[Wikipedia]
Command	A user input that triggers a System Operation.
Conceptual Schema	A conceptual schema (or conceptual data model) is a map of concepts and their relationships. The term is used here to describes the semantics of a Mental Model.
Connector	A Connector is a directed relation between two Roles. It declares that there can be a link between the objects playing the Roles.
Context	Contexts implement System Operations and a System Operation is always executed within a Context. The static (class) side of the Context class specifies a network of communicating objects as a similar structure of interconnected Roles. The instance side of the Context class includes methods that specify how Roles are bound to objects at runtime. A Context instance is a dynamic namespace that binds Roles to objects; its scope is the execution of a System Operation.
Controller	An MVC element in the user interface that coordinates several related Views.
Data	The Data perspective exposes the classes that represent the Conceptual Schema for a system. *DATA. A representation of facts or ideas in a formalized manner capable of being communicated or manipulated by some process.* [ISO-66]
DCI	A paradigm defining a program architecture where a program is seen in different perspectives. Each perspective is a filter that exposes certain properties of the program and hides the rest. The essential perspectives are Data, Context, and Interaction

Environment	(Abbreviated *Env*). For a given system, the Environment is the set of all objects outside the system whose actions affect it, and also those objects outside the system whose attributes are changes by its actions.[OOram]. Baby systems are open Systems; i.e., systems that interact with their environment.
Information	*INFORMATION. In automatic data processing the meaning that a human assigns to data by means of the known conventions used in its representation.* [ISO-66]
Injection	in DCI: A mechanism that maintains the invariant that for any given Role, its Role Methods are shared among all its Role Player Classes.
Interaction	A specification of how a network of communicating objects realize a System Operation. The network nodes are Roles that are played by Data objects at runtime. An Interaction specifies all possible sequences of events (traces) in the execution of a System Operation. This specification is in the form of methods that are specified for each Role and injected into all its Role Player Classes. Polymorphism does not apply to these methods; methods specified for the Roles have priority over methods specified in the Role Player Classes.
Interface	An interface is a set of operations. Interfaces could be associated with Roles to specify messages that must be understood by all objects that play them. The concept is not used in BabyIDE. Partly because it did not seem relevant in our simple examples. Partly because we usually ended up with a large number of very small interfaces when modeling with OOram. We work with Role Methods directly in BabyIDE.
Link	A directed communication path between two objects that permits the transmission of messages to the object at its head from the object at its tail.
Mental Model	*A Mental Model is an explanation in someone's thought process for how something works in the real world* [Wikipedia].
Model	An MVC element that represents user domain information.
MVC	Model-View-Controller. A Paradigm that divides an application program into two distinct parts: The Model part implements the user's Mental Model. A View is a GUI that lets the user work with a particular aspect of the Model. A Controller is an element that manages a number of coordinated Views.
MVC-U	Model-View-Controller-User. The same as MVC, but stressing the importance of the user in the paradigm.
Object	An Object has identity and encapsulates state and behavior. An Object is an instance of a Class. An object can play many Roles. A Role can be played by many objects. "*a language mechanism for binding data with methods that operate on that data*" [Wikipedia]
OOram	A method and tool for modeling with roles.[OOram]

paradigm	Webster[Webster-08]: 3: *broadly*: a philosophical or theoretical framework of any kind Wikipedia [Wikipedia] cites a definition by Kuhn. Parts of it covers our use of the term: • *what is to be observed and scrutinized* • *the kind of questions that are supposed to be asked and probed for answers in relation to this subject* • *how these questions are to be structured*
Role	This concept forms a bridge between the compile time and the run-time properties of a system *node* — A Role identifies a node in a network of communicating objects. *responsibility* — A Role represents the responsibility of an object playing it. *interface* — A Role specifies an Interface that must be implemented in all its Role Player Classes. *methods* — A Role Method is a feature of a Role.
Role Method	A method that is a feature of a Role and is shared among all the Role's Role Player Classes. Polymorphism is suspended for such methods.
Role Player	An object that fills the position of a Role in a network of communicating objects during the execution of a System Operation.
Role Player Class	The class of a Role Player.
Sequence Diagram	A UML notation for an Interaction.[UML]
Smalltalk	A powerful information system, one in which the user can store, access and manipulate information so that the system can grow as the user's ideas grow. The programs constitute an important part of this information. [Smalltalk]
Squeak	A dialect of Smalltalk [Squeak]
System	"*A system is a part of the world which we* choose *to regard as a whole, separated from the rest of the world during some period of consideration; a whole that we* choose *to consider as containing a collection of objects, each object characterized by a selected set of associated attributes and by actions which may involve itself and other objects*".[OOram] A system is characterized by its *state* and *behavior.* The state of an object is composed from the values of its instance variables. The state of a System is composed from the states of its objects and the relations between them. The behavior of an object is composed from the way it handles its operations. The behavior of a System is composed from the way it handles its System Operations.
System Operation	A System Operation realizes certain functionality and triggers an Interaction.

Trigger	A Trigger is a message that starts the execution of a System Operation.
Use case	“*Use cases are a means for specifying required usages of a system. Typically, they are used to capture the requirements of a system, that is, what a system is supposed to do.*”[UML] In Baby, a use case describes a user task and specifies the Commands that must be available to the user when performing this task. Use case → Command → Trigger → Interaction
View	An MVC element that presents Model data in a form that simplifies its transformation to information in the user’s head.