The Common Sense of Object Orientated 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 communicating objects work together to achieve a common goal. Surely, the common sense of object orientated programming is that we should write code that describes how this works. We have, 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 say how these instances interact to achieve the system's behavior nor does it say how they represent the system's state in the objects and the relations between them.

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* is such a paradigm. It separates a program into different perspectives where each perspective focus on a certain system property. Code in the *Data perspective* specifies the representation of system state. Code in the *Context perspective* specifies runtime networks of communicating objects. Code in the *Interaction perspective* specifies how the networked objects collaborate to achieve the system behavior.

The *BabyIDE1* is an experimental 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.

The *BabyIDE1* experiment marks a new departure for object oriented programming technology. It opens up a vista of new and interesting research challenges and product opportunities. 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 – Baby Browser

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

Table of Contents

1 Introduction and summary

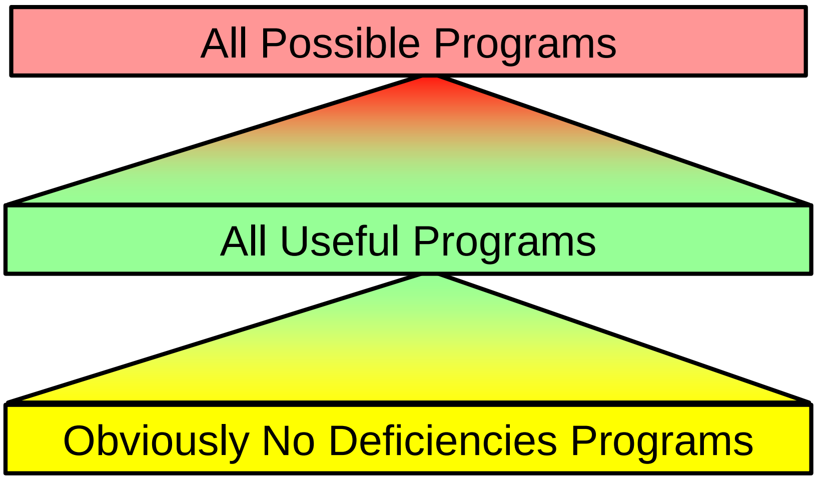
In his 1991 Turing Award Lecture, Tony Hoare succinctly stated the value of readable code [Hoare-81].

> *"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 set of programs that can be executed by a given computer is enormous, this is symbolized by the upper rectangle in the figure. Most of these programs will either come to an early stop, some will run forever. A very small subset are all useful programs that are symbolized by the middle rectangle in the figure. 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, obfuscated programs. They also include the very small subset of programs that are so simple that there are obviously no deficiencies that is symbolized by the bottom rectangle in the figure. These programs must clearly not only be readable, but they must also have been read, checked, and accepted. I call them comfortable programs because I like to have them around.

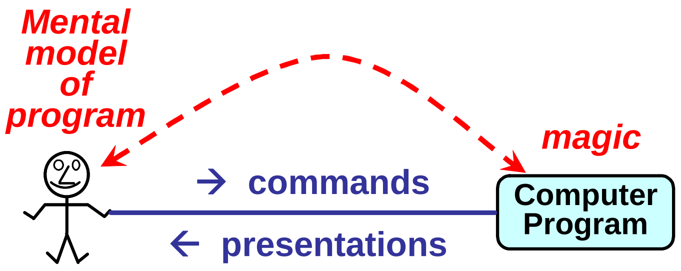
Figure 1: All readable programs are but a small subset of all useful programs.



In this connection, a readable program is much more than a program where I understand the syntax and semantics of its language. I require that the code shall let me build a mental model of the program that tells me both what it does and how it does it. Figure 2 illustrates that the essential property of a readable program is that my mental model and the computer program operate within the same paradigm.

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

Figure 2: What it is all about.



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. As an example of unreadable code 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 [GOF-95]:

> *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 reliable programs if their code doesn't reveal how they work? Are we doomed to live with programs that are so complicated that there are no obvious deficiencies? Some people believe that our problems are so complex that they cannot be solved with readable code. I refuse to accept that. I claim that

> ***No problem is so complex that its hasn't got a simple solution.***
> ***We must merely be prepared to spend the necessary time and effort to find it.***

There is a catch. "*If you only have a hammer, you tend to see every problem as a nail.*" [1]. I believe that since the only tool we have to think with is the class, every problem looks like an object since a class can only describe objects. A *runtime structure that consists of rapidly changing networks of communicating objects* is quite simply not visible if our universe of discourse is a class structure.

We all know that an object encapsulates state and behavior. We tend to forget that this also applies to systems of interconnected objects. The state of a system is the combined state of its objects and the relationships between them The behavior of a system is the set of operations that the system can perform. We are here only dealing with sequential programming where the execution of a system operation is triggered by a message from outside the system. The message is received by one of the system objects and leads to an avalanche of messages that flow through the system. The code that control this flow constitutes the system behavior. This code is fragmented and distributed among the system classes in traditional code. In the BabyIDE, this code is collected and made visible. The critical questions that need to be answered for every system operation are:

1.Attributed to Abraham Harold Maslow.

1. What is the network of communicating objects?

2. How are the objects interlinked?

3. How do the objects interact?

I have called my retirement project *BabyUML*. I have no deadlines, I have no project plan, and I do not need anybody's approval for what I am doing. My goal is to create a new kind of hammer that extends my universe of discourse to explicitly include system state and system behavior. I pursue the mirage of readable, object oriented code.

The BabyUML project has gone through several phases. I have learned something from each of them. Two articles describe the background for the current solution. *The BabyUML discipline of programming*[BabyUML-06] explores new programming elements that have lead up to my current solution. *Programming with Roles and Classes: the BabyUML Approach*[BabyUML-07] describes the DCI paradigm for describing a system state and behavior:

- The ***D*** for ***Data*** perspective is a "micro database" that represents the system state.
- The ***C*** for ***Context*** perspective specifies the configuration of a network of communicating objects that implements a given system operation. The configuration is expressed as a conforming network of objects and the links between them. The context also binds roles to objects by selecting from the set of Data objects at run time. (This will be a *select* operation in database terms, the context as a whole would be a kind of external view on a database)
- The ***I*** for ***Interaction*** perspective specifies how objects work together to realize a system operation. The interaction is coded as *role methods* attached to the roles. For each object that will play a role during its lifetime, we inject these methods into the object's class.

The BabyUML project has reached its first goal. The main deliverable is a Squeak image with two demos:

- ***BabyShapes4***. An animation that visualizes a runtime structure consisting of rapidly changing networks of communicating objects. There are two use cases:
  - The first is an animation where objects are visualized as shapes that are added and removed at random from a rectangular window.
  - The second is an animation that visualizes message passing in a network of communicating objects. The process appears as repeating sequences of arrows connecting randomly selected shapes.
- ***BabyIDE1***. An experimental interactive development environment that supports working with a program seen in the DCA perspectives.

This report is a supplement to the demo programs. It has three main sections that are written so that a reader can meaningfully quit reading at the end of each section.

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

Section 2: *System behavior: the BabyShapes4 visualization*. The story told by the visualization demo.

The first use case is the *ShapesAnimation*. It illustrates that objects are created and garbage collected at runtime by visualizing the objects as shapes that appear and disappear on the screen. Some shapes are stars, some are circles. The difference illustrates that objects may be instances of different classes.

See the ShapesAnimation here[1].

The second use case is the *ArrowsAnimation*. It illustrates that objects communicate by showing messages 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 each repetition, the actual objects are different in each repetition.

See the ArrowsAnimation here.[2]

The section ends with a discussion about how we can make a static description of this unchanging 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 configuration of the network of communicating objects. Its nodes are called *roles*, the edges are called *Connectors*. The roles are bound to objects at runtime.

Section 3: *The BabyShapes4 program as seen in BabyIDE1*. The *BabyIDE* is an Interactive Development Environment that is built on the concepts of classes and roles. The section explains how *BabyIDE1* is used to read and modify the program that drives the *BabyShapes4* animation. The *BabyIDE* idea is that a programmer need to see a program in different perspectives. The perspectives are organized in overlays, and the programmer switches between perspectives by the press of a button.

The BabyShapes4 program is shown in four overlays, each overlay is a browser that is specialized browser for working in one of the perspectives:

Environment: A *BB1ClassBrowser* for browsing the classes of the objects that span the boundary between the system and its environment. An environment is an object that triggers the execution of a system operation.

Data: A *BB1ClassBrowser* for browsing the set of data classes that define the state of the system.

Context: A *BB1ClassBrowser* for browsing the Context classes. There is one Context class for each system operation, this class specifies the Roles, their Connectors, and the methods that bind Roles to objects during the execution of the operation.

Interaction: A *BB1InteractionBrowser* for working with the roles and the methods that are common to all classes of the objects that can play the roles.

The *BB1ClassBrowser* is a browser for working on a set of classes that are relevant for a given perspective[3]. The browser has 5 panes:

1. A list of the classes that are seen in the browser's perspective.

2. A list of the superclasses of the selected class. (A multi-select list).

3. A list of message categories for the selected class and the selected superclasses.

4. A list of methods, including overridden methods, that are visible in the selected classes and categories.

1. http://heim.ifi.uio.no/~trygver/assets/animate-shapes.mov
http://heim.ifi.uio.no/~trygver/assets/animate-shapes.mpeg
2. http://heim.ifi.uio.no/~trygver/assets/animate-arrows.mov
http://heim.ifi.uio.no/~trygver/assets/animate-arrows.mpeg
3.In BabyIDE1, the classes belonging to a perspective are found in a similarly named sub-package in the SystemOrganizer. Example: *BabyShapes4-Context*.

5. The code for the selected method.

The *BB1InteractionBrowser* has 4 panes:

1. A list of system operations
2. A graphical, role based presentation of the configuration of communicating objects for the selected system operation.
3. A list of the methods that are properties of the selected role.
4. The code for the selected method.

Both browsers are designed to tell a complete story in order to minimize the need for moving between several windows.

Section 4: *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 information in the system and the parts that are responsible for interaction with the user.

DCI is the paradigm underlying *BabyIDE1*. The goal of the paradigm is to minimize any gap that might exist between the programmer's mental model of her program and the program that is actually stored and executed in the computer. 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 5: *The BabyIDE1 implementation*. The less said about it, the better.

Section 6: *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.

Section 7: *Further work*. What remains to be done is the formidable task to evolve from the current simple experiment to an alpha version of a usable tools.

This section also suggests some of the jobs that need to be done and some research issues that it can be profitable to study.

2 System behavior: the *BabyShapes4* visualization

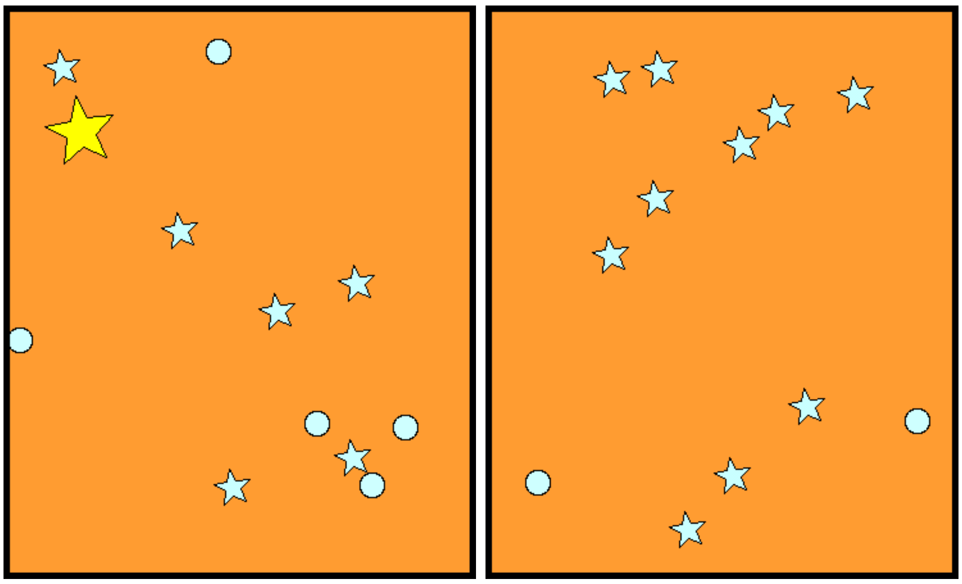
The quote from the Design Patterns book on page 4 says that the code *doesn't tell us how a system will work*. This is clearly a problem that begs for a solution. As a first step towards getting to grips with it, I have written two animations that visualize "*rapidly changing networks of communicating objects*".

The animation program, *BabyShapes4*, shows a system on 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.

In this section, we build intuitive descriptions of what we see in the animations. We find that the basic object paradigm is sufficient for the *ShapesAnimation* but that is inadequate for the *ArrowsAnimation*. We then extend it to describe executions that involve several objects such as the execution visualized in the *ArrowsAnimation.* We call this extended paradigm *DCI*, 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 recommend 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 (See appendix 2.)

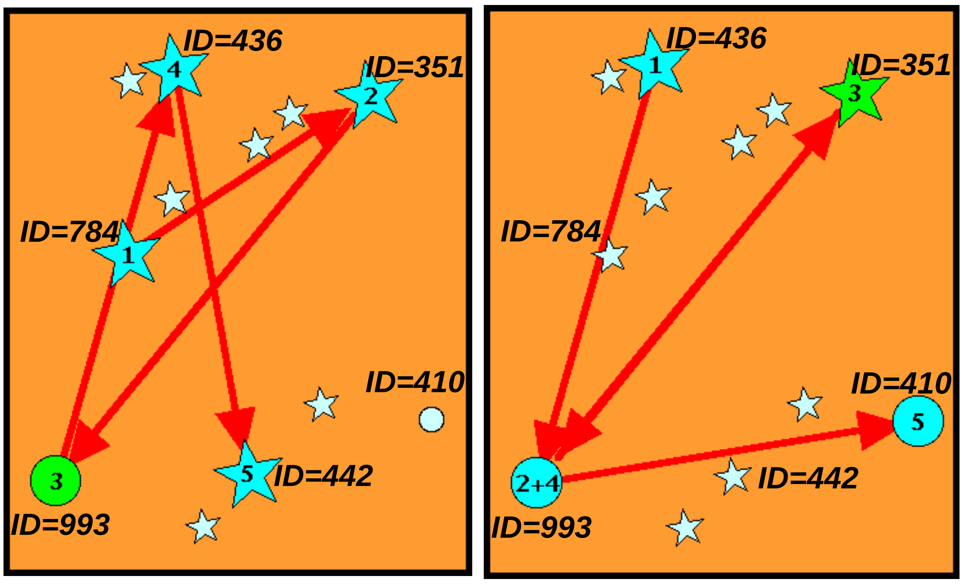
Figure 3: .Two snapshots from the *ShapesAnimation*.



We find that the *ShapesAnimation* visualizes a process that fits well within the basic object paradigm because it only involves one object at the time.

1. http://heim.ifi.uio.no/~trygver/assets/animate-shapes.mov
http://heim.ifi.uio.no/~trygver/assets/animate-shapes.mpeg

Figure 4: Two snapshots from the *ArrowsAnimation*.



The *ArrowsAnimation* is different because it visualizes a runtime structure that "*consists of rapidly changing networks of communicating objects*".Click animate-arrows.mov [1] to see the *ArrowsAnimation*. Again, static snapshots do not communicate the dynamic nature of a program execution and we strongly suggest that you take the trouble to actually see the video or run the program (appendix 2).

Figure 4 shows two snapshots taken from the *ArrowsAnimation*. They visualize that messages are sent from object to object by showing arrows that grow from sender objects to receiver objects. Both snapshots show the system after a train of arrows has connected five objects:

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 and an arrows has been drawn from object 1 to object 2,
3. and so on up to object 5.

The stars and circles are instances of different classes. Note that the classes of the objects do not appear to matter in this animation. For example, object 5 is a star in the left snapshot and a circle in the right one. If this were the visualization of a real and complex process, the classes would clearly 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; 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.

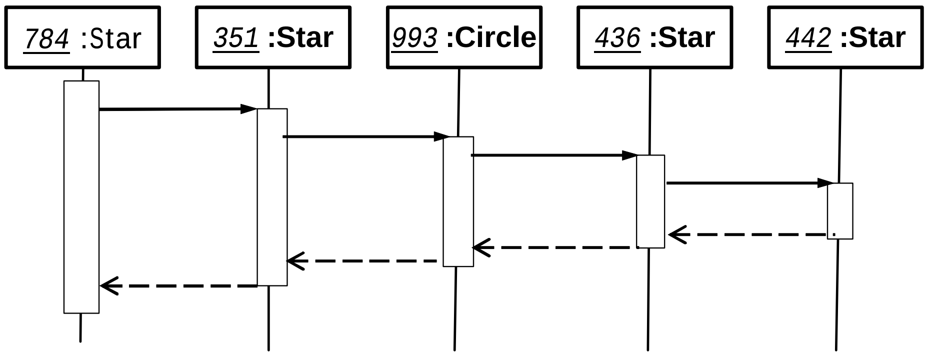
We are keeping things simple, and we are only considering sequential execution. The execution of a system operation starts with a message to some object. This activates a method that sends a message to the same or to another object. We can make a *trace* of the execution by observing every message; its sender, its receiver, and its name (message selector). Every visited object is a participating object. Every sender/receiver pair is a link between those objects. Every message

1. http://heim.ifi.uio.no/~trygver/assets/animate-arrows.mov
http://heim.ifi.uio.no/~trygver/assets/animate-arrows.mpeg

is a part of the sequence of object interactions that realize the system operation.The trace describes a dynamic network of communicating objects; a network that is likely to be unique to that particular execution and that probably does not exist in its entirety at any time. The network disappears at the termination of the execution.

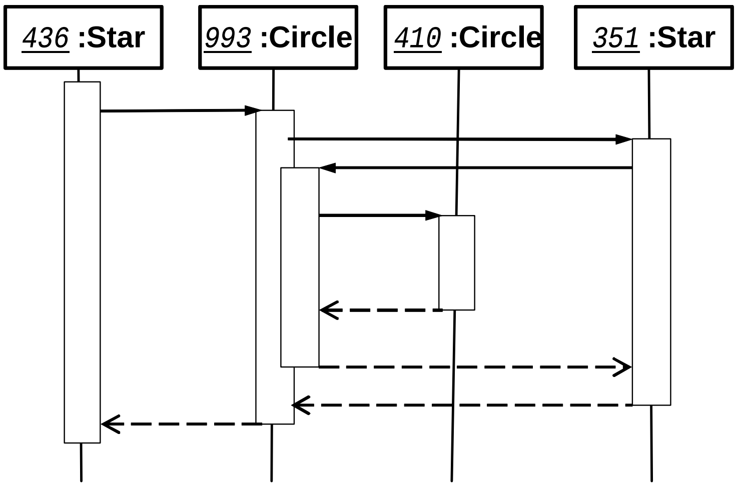
The *sequence diagram* in figure 5 models the process that resulted in the left snapshot of figure 4. A vertical line with a box on top is called a *lifeline*; it represents the history of one of the communicating objects. The box contains the name of the participating object with the format <name :class>.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

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



The sequence diagram of 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 modeling the process visualized in the right snapshot in figure 4.[1]



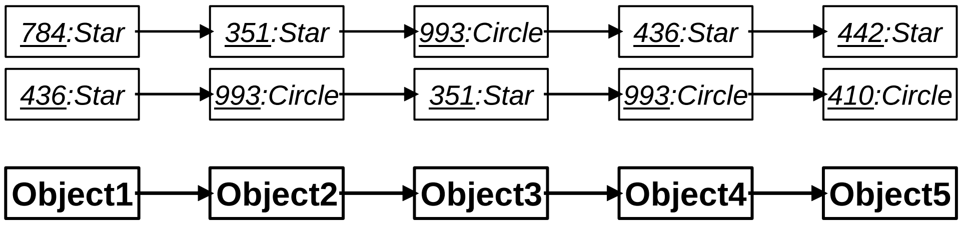
The two sequence diagrams are very different, yet we see the same pattern over and over again when we observe the actual animation. We need to capture this commonality in code so that the code specifies the process explicitly.

We revisit these two execution traces, this time recording the objects visited and the links between them. The result is given as the two top diagrams in figure 7. We then generalize to cover all possible traces for the *ArrowsAnimation* and draw the bottom diagram so that it rep-

1.For the purposes of this document, the animation is visualizing a system with few objects. There is, therefore, a great deal of object reuse between the two snapshots.

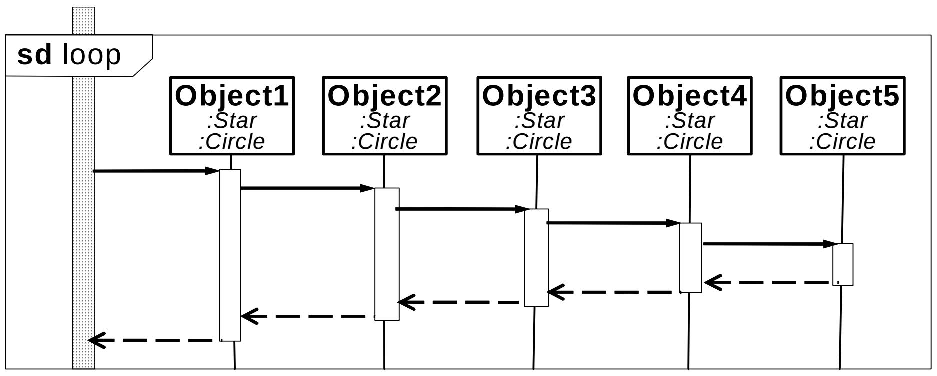
resents their common configuration. The nodes in the new diagram are called *roles* and the edges *Connectors*. The diagram represents the *context* of the interaction, where a role is shown as a rectangle and a Connector as an arrow.[1]

Figure 7: The context of the interactions shown in figure 5 and figure 6.



A use case is realized in the context of a network of communicating objects where the communicating objects are represented by the *Roles* they play in the interaction.We replace the objects in the sequence diagrams of figure 5 and figure 6 with the corresponding roles and get Figure 8. This sequence diagram captures the commonality between all the different arrow trains in the *ArrowsAnimation*. There is one lifeline for each role. It describes part of the life of any object that plays the role in an instance of the interaction. The methods executed by these objects are the same for all of them, they are shown as narrow, vertical rectangles in the diagram.

Figure 8: Sequence diagram for the ArrowsAnimation.



A role can be played by instances of different classes, here symbolized by stars and circles. We call these classes the *Role Player Classes* and have added their names in the diagram to conform to the UML notation. These classes include the properties needed to enable their instances to play the given role.

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

We solve the problem by suspending polymorphism for the methods that are essential for the integrity of the process. We define the methods shown as narrow, vertical rectangles in figure 8 as properties of the roles and force the Role Playing Classes to share these methods and prohibit any private versions in the classes.

We discussed the problem statement, *foo delete*, on page 3. The DCI paradigm lets us make the statement more explicit. We write something like *Object2 delete* where *Object2* is a reference to the object that currently happens to play the *Object2* role. The method selector *delete* now causes the execution of a *delete* method that is a property of the *Object2* role. A reader of the 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.

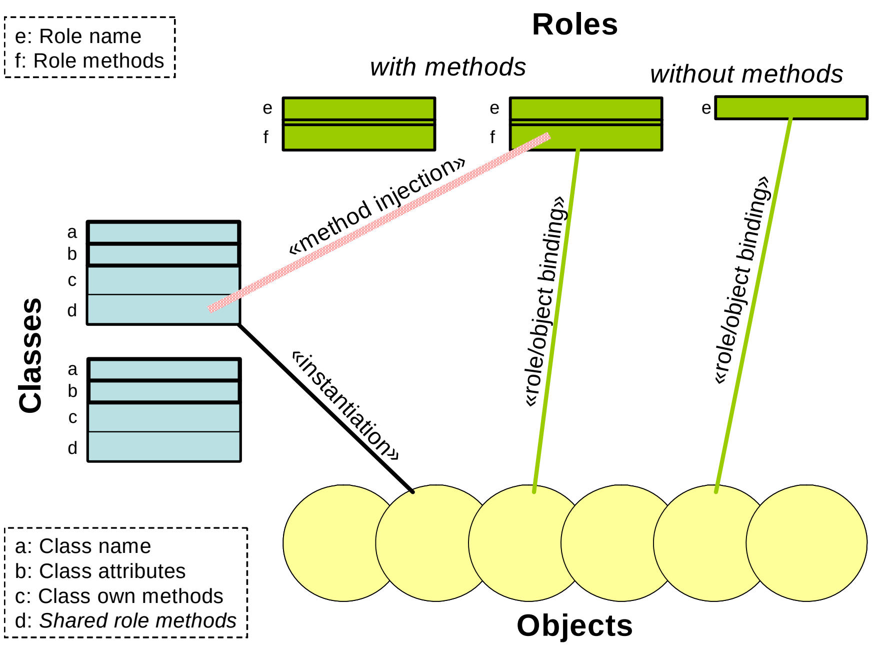
1.Such networks are called role models in [OOram] and Collaborations in [UML].

The concept of a role is central in 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 capabilities needed to play the role.
- *function*: A role represents a functionality specified by the role methods.
- *job, task, work*: Work has to be done to perform a use case. 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*: A role represents a node in the network of communicating objects.
- *place, position*: A role represents a position in the network of communicating objects.

James Coplien has contributed the illustration of the essential concepts of the DCI paradigm that is shown in figure 9. The classes are drawn according to the UML notation with one additional compartment for the methods that are injected into the class from the roles. There are two versions of the roles. Some roles are defined with method properties; these methods are injected into the appropriate classes. All roles are bound to objects at runtime; these binding are valid during the execution of a system operation.

Figure 9: The relationships between Roles, Classes and Objects according to James Coplien.



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 *BabyIDE1*; it is described in section 3. It is based on the DCI paradigm that is described in detail in the next section.

3 The *BabyShapes4* program as seen in *BabyIDE1*

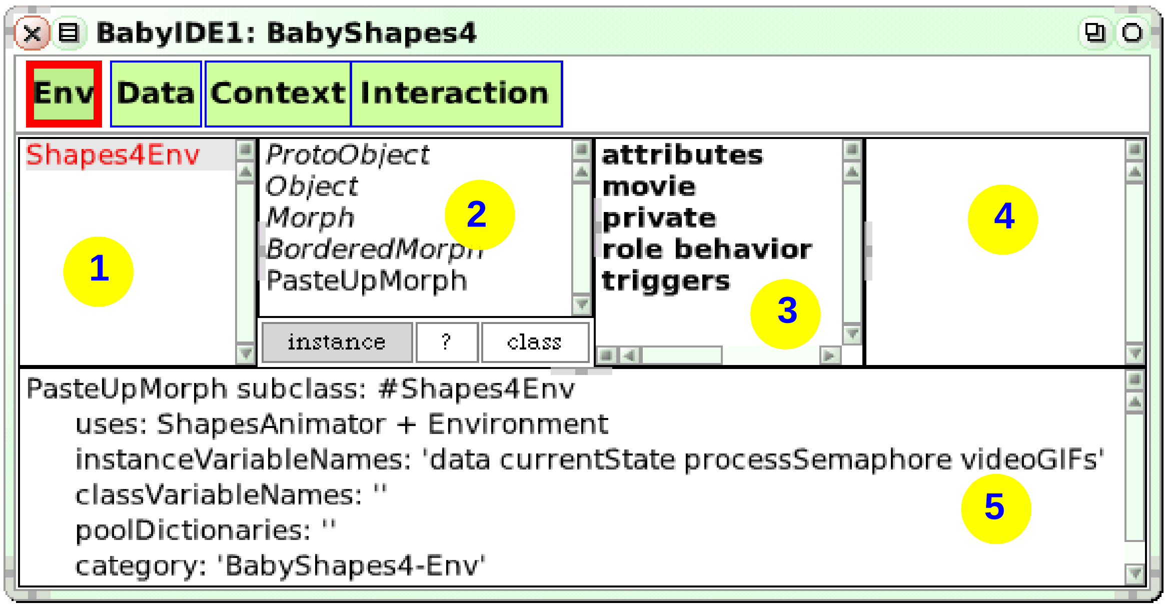
The *BabyIDE* is a tool designed for working with a program as seen in different perspectives. Each perspective tells part of the story; all perspectives taken together tell the whole story. We introduced the DCI paradigm on page 5 and have discussed it extensively in the preceding pages. The BabyIDE supports this paradigm by providing specialized browsers for four perspectives:

Data	The data classes. These classes specify system state and can be relatively simple since they do not specify runtime structure and behavior.
Context	These classes specify the runtime configuration of roles, their communication paths, and the methods that bind roles to data objects at runtime.
Interaction	The context diagram is a graphical presentation of the roles together with their communication paths. This perspective also shows the role methods, i.e., methods that define the behavior of the communicating objects irrespective of their class.
Env	The system *Environment* consists of all objects outside the system. The *Env* classes specify the objects in the environment that trigger system operations.

A first, experimental BabyIDE has been implemented in Squeak[Squeak], a dialect of Smalltalk. Its first implementation is called *BabyIDE1*.

We discussed the *BabyShapes4* program as it is seen by its users in section 2. We now shift our attention to the code for the *BabyShapes4* as it is seen by a programmer. figure 10 is a snapshot of *BabyIDE1*. It is opened on the *BabyShapes4* program and is organized in the four perspectives *Env*, *Data*, *Context*, and *Interaction* as shown in the strip of flaps in the figure. 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 flap above the browsers. The Env perspective has been chosen in the figure.

Figure 10: BabyIDE1 showing the *Env* class definition.



```
PasteUpMorph subclass: #Shapes4Env
    uses: ShapesAnimator + Environment
    instanceVariableNames: 'data currentState processSemaphore videoGIFs'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BabyShapes4-Env'
```

We will discuss each of the program as seen in the four perspectives in the following subsections. We will find that the classes we see in the *Env* and *Data* perspectives 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 system structure and behavior. The *Env* and *Data* classes are, therefore, much simplified and we can trust that they do not hide ugly surprises from a reader of the code.

3. 1 The Env perspective

The *BabyShapes4* environment object is an instance of the *Shapes4Env* class. It is presented and edited in the *Env* perspective as shown in figure 10. This perspective is a *BB1ClassBrowser, a browser* designed for browsing a small set of classes. This browser has four panes as follows:

1. Class list showing the *Env* classes.
2. The superclasses of the selected class shown in a multiple-select list. This is a presentation filter, only features of selected superclasses are shown.
3. 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 selected superclasses and method categories are shown.
4. A list of methods in the selected classes, superclasses, and method categories. The implementing classes are shown in parenthesis.
5. A code pane showing the selected method. The class definition is shown in figure 10 because there is no category selection.

The *Shapes4Env* class definition is shown in figure 10. Comments:

line 1:	The *PasteUpMorph* superclass gives *Shapes4Env* the capability to show the animation on the screen.
line 2:	this line makes the methods of the named roles become part of the class behavior. This is where the code for the runtime behavior is injected into the class from the Interaction perspective see section 3. 3 below.
line 3:	*data* is a link to the Data objects. Their classes are coded in the Data perspective, see section 3. 2 below.
line 6:	The category name is used by *BabyIDE1* to find the *Env* classes.

The *Env* classes form the bridge between a system and its environment. The code for the *Shapes4Env>>yellowButtonActivity* is shown as selected in figure . We see that the *animateShapes* command triggers the *startShapesAnimation* method. Likewise, the *animate arrows* command triggers the *startArrowAnimation* method. Each of these methods trigger an avalanche of messages that achieves the use case through a flow of messages through a network of communicating objects:

```
1. Shapes4Env>>startArrowAnimation
2.      currentState = #ARROWS ifTrue: [^self].
3.      currentState := #ARROWS.
4.      processSemaphore wait.
5.      [   Shapes4ArrowsCtx executeInContext: [self animateArrows].
6.          processSemaphore signal.
7.      ] fork.
```

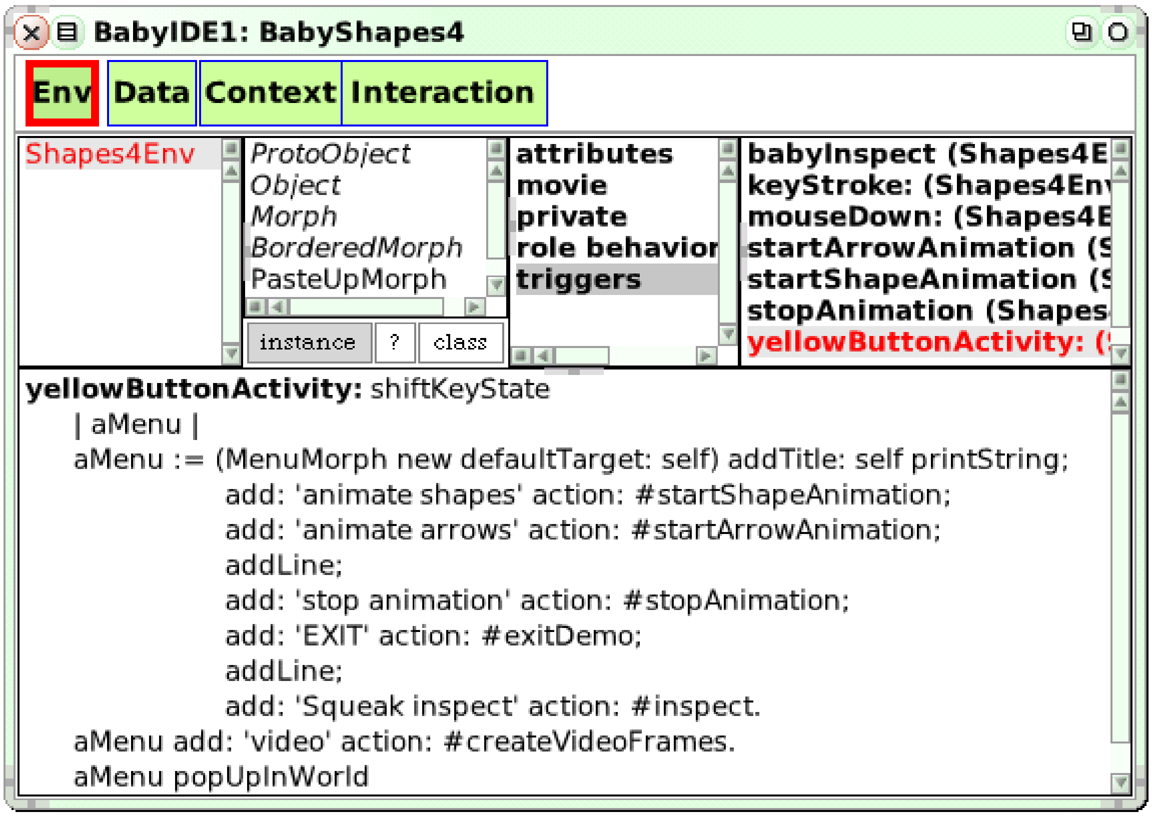
Comments:

Line 2:	We don’t start the animation if it is already running.
Lines 4-7:	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.

Line 5: A crucial statement. It says that we execute *animateArrows* in the context of an instance of *Shapes4ArrowsCtx*. This ensures that the role names will be bound to actual objects during the execution of the method and all its submethods regardless of class.

The participating objects will be instances of the *Data* classes, but the networks will be different for the different use cases and the selection of objects that fill the nodes of the networks will also be different.

BabyIDE1 showing the *Env* class definition.



```
yellowButtonActivity: shiftKeyState
    | aMenu |
    aMenu := (MenuMorph new defaultTarget: self) addTitle: self printString;
                add: 'animate shapes' action: #startShapeAnimation;
                add: 'animate arrows' action: #startArrowAnimation;
                addLine;
                add: 'stop animation' action: #stopAnimation;
                add: 'EXIT' action: #exitDemo;
                addLine;
                add: 'Squeak inspect' action: #inspect.
    aMenu add: 'video' action: #createVideoFrames.
    aMenu popUpInWorld
```

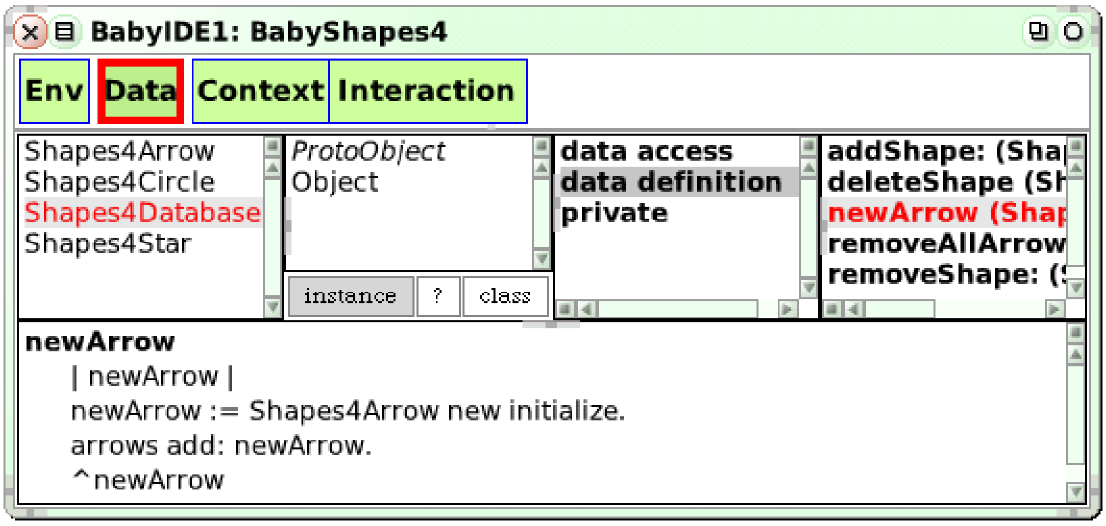
This leads to the *Interaction* perspective as described in section 3. 3 on page 16.

The mental transition from regular Squeak to code that is valid within a context may the hardest part to get used to. Once it is mastered, the code is eminently readable.

3. 2 The Data perspective

The browser for the *Data* perspective is a *BB1ClassBrowser* as shown in figure 11. We see the shapes and arrows classes, *Shapes4Circle*, *Shapes4Star*, and *Shapes4Arrow*. The growing arrows visualize message passing in the animation, but in the *BabyShapes4* program they are instances of regular classes. In addition, there is the *Shapes4Database* class for data definition and selection.

Figure 11: The BabyIDE1 showing the *Data* perspective.


BabyIDE1: BabyShapes4

Env | Data | Context | Interaction

```
Shapes4Arrow        ProtoObject     data access        addShape: (Sha
Shapes4Circle       Object          data definition    deleteShape (Sh
Shapes4Database                     private            newArrow (Shap
Shapes4Star                                            removeAllArrow
                                                       removeShape: (
                    instance ? class
```

```
newArrow
    | newArrow |
    newArrow := Shapes4Arrow new initialize.
    arrows add: newArrow.
    ^newArrow
```

An important feature of the DCI style of programming is that the data classes do not define the runtime object structure. An example is the Shapes4Star class definition:

```
1. StarMorph subclass: #Shapes4Star
2.     uses: Shape1 + Shape2 + Shape3 + Shape4 + Shape5
3.     instanceVariableNames: 'smallExtent bigExtent'
4.     classVariableNames: ''
5.     poolDictionaries: ''
6.     category: 'BabyShapes4-Data'
```

Comments:

Line1:	*Shapes4Star* is subclass of the library class *StarMorph*.
Line 2:	This line shows the *role methods* from the named roles are injected into this class definition. The methods are defined in the *Interaction* perspective, they are read-only in the *Data* perspective classes. This protects us against surprises; the story told in the *Interaction* perspective shall be the whole story.
Line 3:	The instance variables are only *smallExtent* and *bigExtent*, there is no reference to the arrows that bind shapes together.

All methods defined in this perspective shall all be “local” in the sense that they shall not have system-wide side effects.

The runtime shape and arrow objects are held in a micro database that is an instance of *Shapes4Database*.

3. 3 *The Interaction perspective*

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 of the *BabyIDE1* is the *Interaction* perspective that forms our main answer to this request. This perspective show us the *“rapidly changing networks of communicating objects”* captured as a stable network of interconnected roles. (And the Context perspective binds these roles to runtime objects).

The Interaction perspective is where we specify how a DCI program realizes the system operations (use cases). There are three critical questions that the code must answer for each *system operation*:

1. *What are the objects?* This question is answered by the role binding methods in the Context.
2. *How are they interlinked?* We answer this question by displaying a network of connected roles.
3. *How do they work?* We answer this question by fitting roles with a method property. These methods are called *Role Methods*[1]. The Role Player lasses are constrained to give preference to these methods and are not allowed to override them. The only variables visible in the role methods are:
   a. *self*, i.e. the object that happens to play this role.
   b. *connected roles.* A role can be connected to other roles as seen in the Context Diagram.

A DCI program is edited in the *Interaction perspective* with a *BB1InteractionBrowser*. This a new graphical browser that answers the above critical questions and that lets us work with the implementation of a *system operation* in the context of a network of interacting objects. This in contrast to our class browsers where we only see one kind of object at the time.

We will illustrate the power of the *BB1InteractionBrowser* with two examples, one for the *ArrowsAnimation* and one for the *ShapesAnimation*.

Figure 12 shows the code for the *ArrowsAnimation* in the Interaction perspective.The panes are as follows:

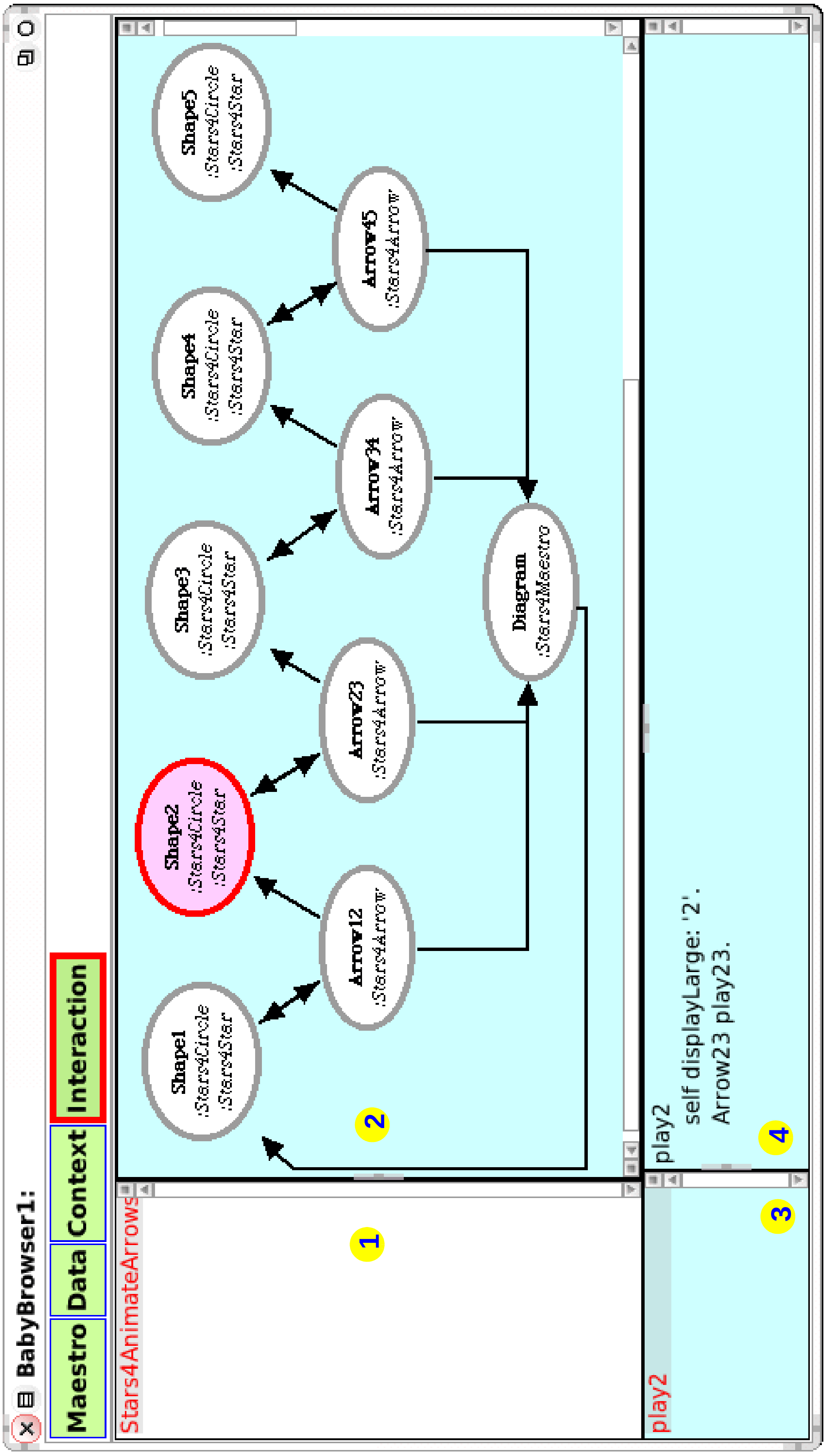
1. *use cases:* A single-selection list of *system operations* (use cases). The *ArrowsAnimationCtx* is selected here. (There is a one-to-one correspondence between use case and Context class in *BabyIDE1*).
2. *context diagram:* The context diagram for the selected use case. It shows the roles of the interacting objects and their Connectors.
3. *role methods:* A single selection list of role methods for the role selected in the context diagram.
4. *code:* The code for the selected method.

We see a context diagram with the 9 shape and arrow roles and their Connectors. In addition, we see the *Diagram* role that represents the system in figure 4 on page 9 and is bound to the colored background at runtime.

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 make all Role Playing Classes be users of this trait.

3. 3. 1 The ArrowsAnimation use case

Figure 12: The *BB1InteractionBrowser*.



```
play2
    self displayLarge: '2'.
    Arrow23 play23.
```

The *context diagram* is an example of graphical code. Roles and Connectors can be added and removed graphically as a convenient way of coding the context.

Roles are played by objects that are specified by classes. *Shape2* is selected in figure 12. Classes are bound to the roles by the '*add using class*' yellow-menu command. The instances of a using class are objects that can play the selected role. We follow the UML style and show the role names extended with using class names preceded with a colon. We see from the diagram that *Shape2* has two player classes: *Shapes4Circle* and *Shapes4Star*. These classes implement the *play2* role method and will behave as specified in pane #4.

The first role methods are as follows:

```
1.  Environment>>animateArrows
2.      [self currentState == #ARROWS]
3.      whileTrue:
4.          [self removeAllArrows.
5.          ThisContext reset.
6.          Shape1 play1.
7.          (Delay forMilliseconds: 1500) wait].
```

Comments:

Line 1: This method is a property of the Environment role.

Line 5 *ThisContext* is the role name for the current Context instance. We reset it to compute a fresh set of role-to-object bindings.

Line 6: The context diagram (figure 12) shows the roles that are visible from a given role. Here: The *Environment* role has a link to *Shape1*. Visible roles are legal variables in a role method. We start the ball rolling, beginning with the *play1* method in the *Shape1* role.

```
1.  Shape1>>play1
2.      self displayLarge: '1'.
3.      Arrow12 play12.
```

Comment:

line 2: *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 can of course vary over time.)

```
1.  Arrow12>>play12
2.      Environment addMorphBack: self.
3.      self from: Shape1 to: Shape2.
4.      self grow.
5.      Shape2 play2.
1.  Shape2>>play2
2.      self displayLarge: '2'.
3.      Arrow23 play23.
1.  Arrow23>>play23
2.      Environment addMorphBack: self.
3.      self from: Shape2 to: Shape3.
4.      self grow.
5.      Shape3 play3.
```

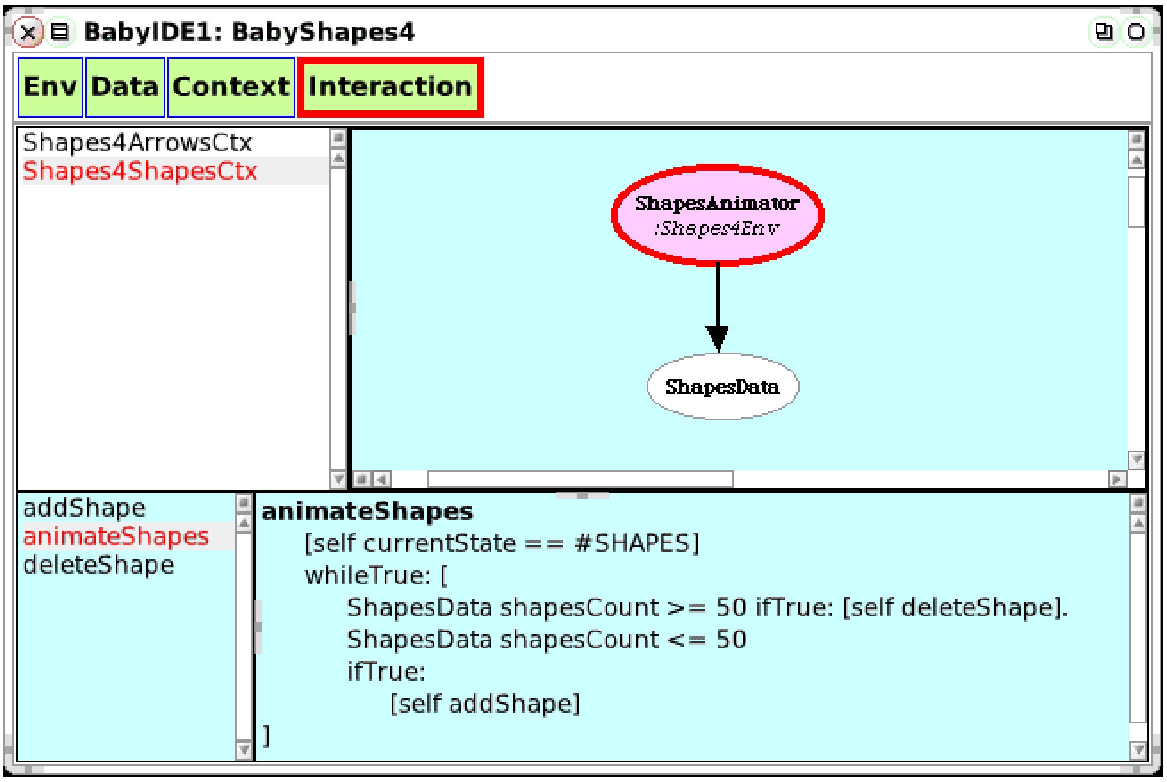
And so on until *Shape5*.

The Interaction perspective tells us the essentials about how the *ShapesAnimation* works: What are the roles? How are they connected? How do they interact? And even: What do they do? There are no surprises; the Role Player Classes do not know "*the network of communicating objects*" so no overworked maintainer can upset our great scheme.

The code is readable.

3. 3. 2 The ShapesAnimation use case

Figure 13: The ShapesAnimation program in the Interaction perspective.



```
animateShapes
    [self currentState == #SHAPES]
    whileTrue: [
        ShapesData shapesCount >= 50 ifTrue: [self deleteShape].
        ShapesData shapesCount <= 50
        ifTrue:
            [self addShape]
]
```

The *ShapesAnimation* involves a single shape at the time. This shape is either added or removed from the pool of shapes. The context diagram is extremely simple as shown in figure 13. The essential role is called *ShapesAnimator*. This is a role with methods; the methods that drive the animation. *ShapesAnimator* is connected to *ShapesData*; a role without methods.

If we look in the *Shapes4ShapesCtx* class, we find that the *ShapesAnimastor* is always bound to the Environment object and the *ShapesData* is always bound to the *Shapes4Database* object.

The *ShapesAnimation* methods as defined here are injected into the *Shapes4Env* class. There is no polymorphism, and the methods could just as easily have been written as regular methods in the *Shapes4Env* class.

Using DCI here is a matter of taste. The advantage is that all the code for the ShapesAnimation is pulled out of the class so that the whole story is collected in the appropriate perspectives. The disadvantage is that the code is distributed into several perspectives, and most people would probably just add the code into the *Shapes4Env* class as follows:

```
Shapes4Env>>startShapeAnimation
    currentState = #SHAPES
        ifTrue: [^ self].
    currentState := #SHAPES.
    processSemaphore wait.
```

```
        [    self removeAllArrows.
             self animateShapes.
             processSemaphore signal
        ] fork
```

The shapes are added and removed until the loop is terminated manually:

```
    Shapes4Env>>animateShapes
        | newShape suggestedShapeCenter |
        [self currentState == #SHAPES]
            whileTrue: [data shapesCount >= 50
                ifTrue: [data deleteShape].
                data shapesCount <= 50
                ifTrue: [newShape := (Collection randomForPicking next * 10) rounded odd
                            ifTrue: [Shapes4Star new initialize]
                            ifFalse: [Shapes4Circle new initialize].
                    data addShape: newShape.
                    [suggestedShapeCenter :=
                        (self bounds left to: self bounds right) atRandom
                            @ (self bounds top to: self bounds bottom) atRandom.
                    data allShapes
                        noneSatisfy: [:someShape | (someShape fullBounds extendBy: someShape extent)
                            containsPoint: suggestedShapeCenter]] whileFalse.
                    newShape center: suggestedShapeCenter.
                    self addMorphBack: newShape.
                    newShape flash]]
```

This method sends some messages to the shape objects, but they are all local to the objects and cannot interfere with the animation as a whole. There can be no surprises; the three methods above are self-contained. The important parts are all together and we conclude that the code for this use case is readable.

The disadvantage of this monolithic solution is that the code gets mixed into the other code of the already pretty large *Shapes4Env* class. It also feels wrong architecturally because the *ShapesAnimation* code gets mixed up with the environment. I have tried moving the code into a separate class, but this doesn't clarify anything.

In conclusion, my preference is to follow DCI so that this use case is implemented the same way as the other one.

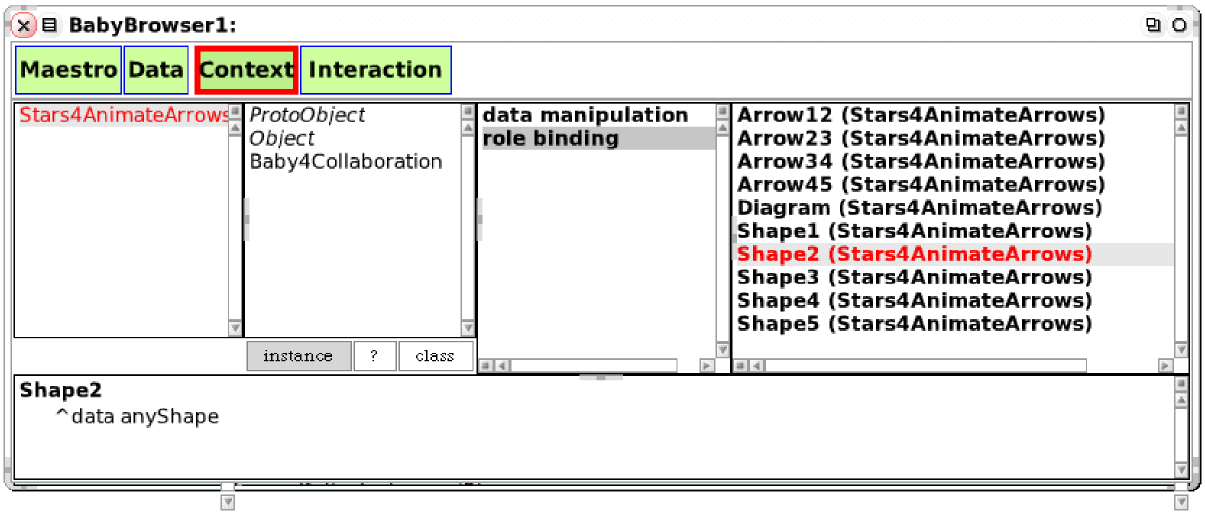
3. 3. 3 The Context perspective

Classes seen in the Context perspective form the bridge between the static data classes as seen in the Data perspective and the runtime system behavior as seen in the Interaction perspective. The instance methods in the context classes are equally important as the class methods.

An instance of a context class is responsible for a runtime dictionary that maps the role names to the object playing these roles. There is one entry in the dictionary for each role shown in the context diagram.figure 12. The mapping between role and object is defined by a method, one method for each role. figure 14 shows that the methods are in the role binding message category. For example, the mapping of the *Shape2* role:

```
    Shapes4AnimateArrows>>Shape2
        ^data anyShape
```

Figure 14: *BabyIDE1* in the Context perspective.



Stars4AnimateArrows

ProtoObject
Object
Baby4Collaboration

data manipulation
role binding

Arrow12 (Stars4AnimateArrows)
Arrow23 (Stars4AnimateArrows)
Arrow34 (Stars4AnimateArrows)
Arrow45 (Stars4AnimateArrows)
Diagram (Stars4AnimateArrows)
Shape1 (Stars4AnimateArrows)
Shape2 (Stars4AnimateArrows)
Shape3 (Stars4AnimateArrows)
Shape4 (Stars4AnimateArrows)
Shape5 (Stars4AnimateArrows)

```
Shape2
    ^data anyShape
```

The mapping between a role and its object is defined in one method. There are no surprises; the code is readable. The selection of objects is random in this simple example. In a *real* program, the *Shape2* method would use business knowledge to indicate the appropriate object in context at run time. The method is like a *select* statement in a database-centered application.

There is no magic involved here. The details of the contexts and the role/object binding is given in appendix 3 for the specially interested.

4 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. This model protects me against surprises and helps me interpret what I see and predict what will happen if I do things with my hands and feet.

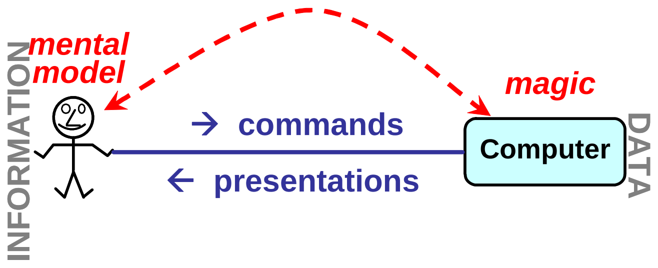
A computer system is part of my environment. A computer system that serves me well will let me build a mental model of its internals that helps me understand its presentations, remember its operations, and predict the effect of those operations.

The ISO vocabulary[ISO-66] distinguish 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.*[ISO-66]

Figure 15 illustrates the relationship between user and system. 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. The mental model itself 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 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

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. A typical example are the so called wizzards where I often do not know the underlying semantic model and, therefore, do not know what I should answer.

precisely, my mental model and the computer representation of my program must follow the same paradigm.

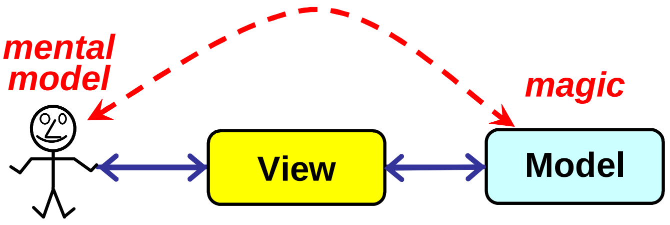
MVC and *DCI* are two paradigms that are designed to be shared by me as a programmer and my program as represented in my code. MVC is old, I implemented the first program designed according to this paradigm in 1978. DCI is new, the first program designed according to this paradigm was the *BabyShapes4* visualization program that is described in section 2 on page 8. The first experimental IDE supporting DCI was the *BabyIDE1* described in section 3 on page 13.

4. 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.

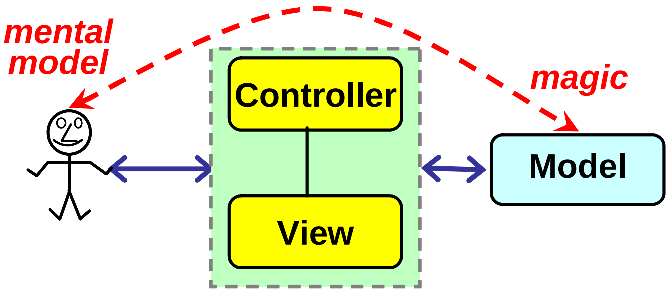
Figure 16: GUI bridges gap between brain and model.



The separation between Model and View leads to greatly simplified code due to the separation 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 tasks may require tools that include several, coordinated views. This coordination is done by a Controller as illustrated in figure 17.

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



The MVC paradigm has proven its value over its 30 years existence. It is usually applied as a pattern without any specific tool support.

Some models may be complex. The DCI paradigm was originally intended for the detailing of the Model part, but we now believe that DCI has many applications outside this scope.

4. 2 DCI: the Data-Context-Interaction paradigm

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

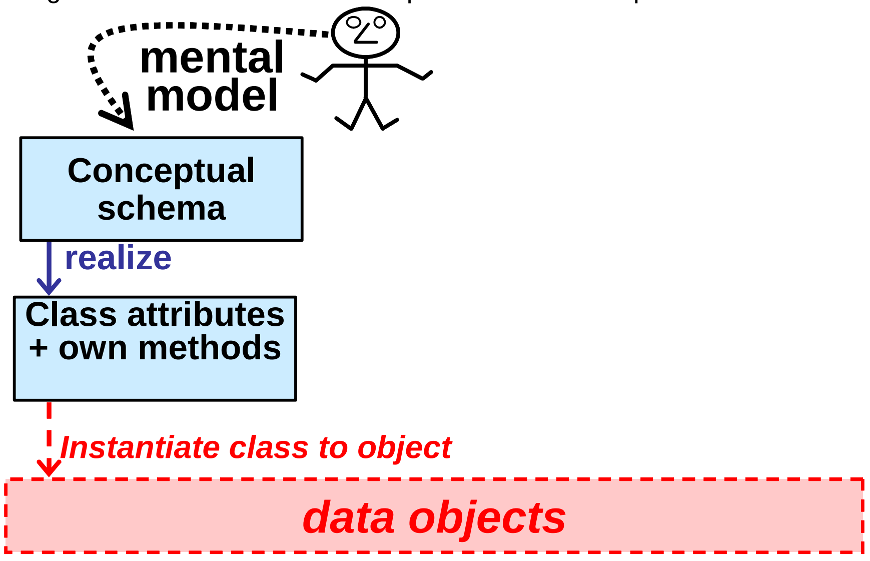
Data:	The computer representation of the user's mental model.
Context:	Specification of a network of communicating objects that realizes a user command.
Interaction:	Specification of how objects communicate when executing a user command.

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

4. 2. 1 System state: the Data perspective

The 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 shows this part of the paradigm. The user's mental model is described in an optional *conceptual schema*. A suitable language for the schema could be something like NIAM[NIAM], ODMG[ODMG], or even UML class diagrams without behavior[UML].

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



The conceptual schema is coded as a set of class definitions in the *Data perspective*. The only methods that belong in these classes are data access methods and methods for derived attributes. An example of a Data perspective is shown in figure 11 on page 16. It keeps the data classes together so that it is easy for a reader to see the conceptual schema.

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

Note that the actual data objects and their relations are runtime phenomena. These runtime elements are shown in red with dashed lines in figure 18.

4. 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 *BabyIDE1* 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]

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

Env: A class that specifies how the execution of a *system operation* is triggered by a message to one of the Context roles.

Context: A class that specifies a network of communicating objects as a similar structure of roles and Connectors.[3] The context class also specifies methods that bind roles to objects at runtime.

Interaction: A specification of how objects interact to realize a system operation expressed in terms of their roles.

1.We use the terms use case, user command, and system operation as synonyms in our discussion of the DCI paradigm because they all describe phenomena that trigger system behavior.
2.We only consider single thread execution in this first version of the DCI paradigm.
3.The notion of communicating roles is taken from OOram and is described intuitively on page 6 ff.

Figure 19: System behavior modeled in a BabyContext (role model).

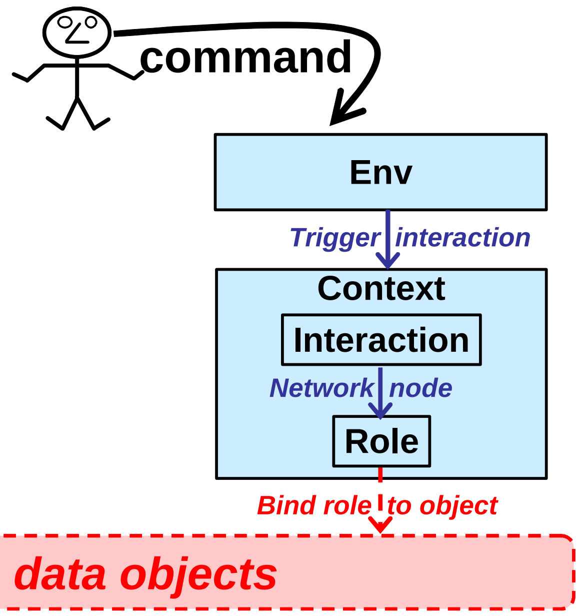


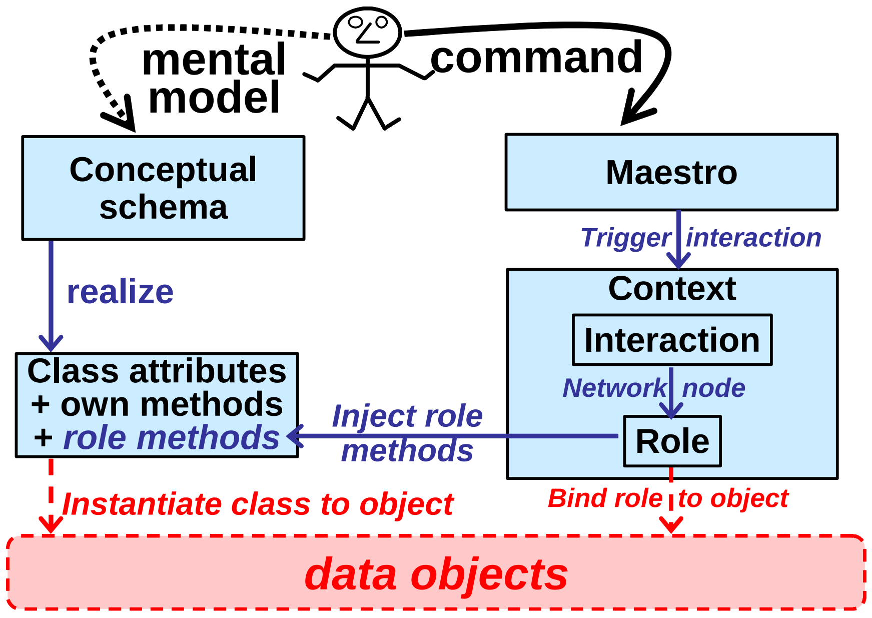
Figure 19 models system behavior up to interaction messages. Its critical weakness is that the methods triggered by these messages will normally be properties of the data classes. This means that different objects that play the same role may handle the same message with different methods.

The DCI paradigm resolves 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 properties of the role. The role methods are *injected* into the data classes, and the classes may not override them. A code reader can, therefore, trust that there are no surprises hidden in the code.

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], [2]

1.Traits are used to implement role methods in BabyIDE1. This usage differs from regular Traits by blocking Trait overrides in the class.
2.Aspect oriented programming [AOP] may possibly be used to implement a BabyIDE in languages that are not as expressive as Squeak. This has not been investigated.

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.

5 The BabyIDE1 implementation

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

In spite of all its flaws, *BabyIDE1* has two important characteristics to recommend it. It exists, and it illustrates what programming according to the DCI paradigm is all about.

6 Conclusion

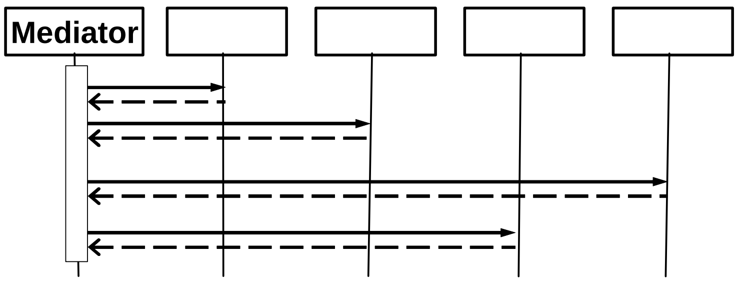
I started with a dream; *let me make my programs so simple that there are obviously no deficiencies*. I found that my current object oriented programs *won't reveal everything about how a system will work*. Obviously, the present wasn't good enough and I had to do something about it. I have achieved what I set out to do on page 5; I have extended my universe of discourse to explicitly include system state and system behavior. I have achieved readable, object oriented code.

The *ArrowsAnimation* is a challenge to conventional OO programming because it involves a network of communicating objects where the class and identity of object 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 21.

Figure 21: Extreme centralization. The Mediator pattern.

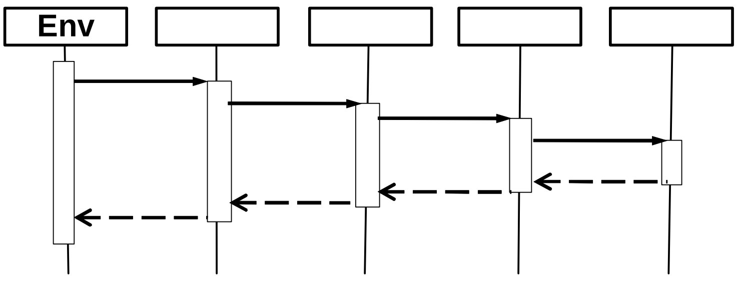


The Mediator class specify how the communicating objects communicate. The class gets very complex if the communication pattern is complex.

The DCI paradigm is a middle solution between the extreme centralization of the Mediator and the extreme distribution of 'pure' object orientation.

Extreme fragmentation is too chaotic, extreme centralization is too rigid. DCI leads to distributed logic under full control. The sequence diagram in figure 22 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 22: Sequence diagram illustrating DCI distributed logic.



The *BabyShapes4* program is the only program that so far has been written according to the DCI paradigm. Extrapolating from a single point is somewhat uncertain, but I claim that the results are promising. I have observed that the DCI implementation of the *BabyShapes* animation program is radically more readable than the many versions preceding it.

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.

Is DCI a procedure oriented paradigm in disguise? My answer is no. I see DCI as more object oriented than regular OO. In DCI, I work with objects represented by the roles they play in an execution. In regular OO, I work with classes and can only see one object at the time.

DCI has drawn upon concepts from many sources. Semantic models from database technology and the idea of calling trees from procedure orientation are two of them.

The DCI paradigm outlaws many working programs as was illustrated in figure 1 on page 3. I will miss many cherished programming constructs that I can no longer use if they lead to unreadable programs.

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.
- When polymorphism gets in the way of readability, I suspend it for the methods that are essential for the integrity of an interaction.
- All objects that play a given role shall process the same interaction messages with the same methods.
- An important feature of the DCI style of programming is that the data classes do not define the runtime object structure.

I take comfort from history. Some time in the sixties, Dijkstra considered the GOTO statement harmful. I 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 slowly 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 hard to extend their attention with runtime behavior, I am confident they will find such a shift very profitable and well worth the effort. The resulting code can be audited before testing and understood by maintainers.

I simply refuse to continue writing unreadable programs. I started the BabyUML project with the conviction that object oriented program code could and should be readable. The background for the project name was somewhat whimsically as follows:

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.

The world's first digital stored program computer was the *Manchester Small Scale Experimental Machine—"The Baby"*. This Baby was small, it was designed for testing the Williams-Kilburn cathode ray tube high speed storage device. It was a truly minimal computer with an operations repertoire of just 7 instructions. It executed its first program on 21st June 1948. The machine was insignificant in itself, but it marked the beginning of a new era.

The BabyUML project has 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 descibes one object at the time; there are no facilities for describing networks of collaborating objects). The result of the BabyUML project is like a new born baby. Its functionality is extremely limited. It cannot 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.

7 Further work

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

Completion: A first task is to develop *BabyIDE* to a state where I and others can use it to write different kinds of programs.

Dynamic binding: The current implementation binds roles to objects with the Context>>reset method. Other schemes are possible, e.g. that a role name triggers the execution of the binding function when it is referenced. Early experiments with this solution led to a code with many surprises. An in-depth discussion of the binding issue should prove very interesting.

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 or a NIAM like model

BabyIDE2: BabyIDE should be re-implemented based on 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 programs. This looks like a good theme for a doctoral thesis. It could start from Egil Andersen's work on role model synthesis.[Andersen-97]

Multi-threading: BabyIDE1 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 of system state and system behavior.

Platforms: Port a BabyIDE to Java, Eclipse, AspectJ, Ruby (on Rails), or some other programming environment. Jim Coplien says that there is an obvious implementation of Traits in C++ using templates. Somebody should take him up on this.

It should be useful to create and market a professional variant of BabyIDE some time in the not too distant future.

My hope is that this first *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 BabyIDE1 as required.

8 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 BabyIDE1 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 object 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 both been following our separate paths when searching for a common truth we both have felt must be out there somewhere. At long last we both feel that we are on a common track of something important.

9 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 40 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 and was a contributor to UML 1.4. The goal of his current research is to create a new, high level discipline of programming that lets us reclaim the mastery of software.

Appendix 1: Baby Terminology

Baby	Prefix for the names of artifacts produced in the BabyUML project.
BabyIDE	An Interactive Development Environment for developing programs that are structured according to the DCI paradigm.
BabyShapes4	A Squeak animation program that visualizes an example of rapidly changing networks of communicating objects.
BabyUML	A project that aims at a new, system-oriented discipline of programming where code shall explicitly specify system behavior as well as system state.
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]
Cognitive model	*"A cognitive model is an approximation to animal cognitive processes (predominantly human) for the purposes of comprehension and prediction."*[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. This 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	(Abbreviated ctx). There is one Context class for each System Operation. The class (static) side of the Context class defines 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 to be bound to objects at runtime. A Context instance is a dynamic namespace that binds roles to objects; its scope is the execution of the operation.
Data	The Data perspective exposes the classes that represent the state of the 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	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.
Env	Abbreviation of 'Environment'. Used to indicate the classes of the Environment objects.
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. A Baby interaction specify 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. Polymorphism does not apply to these methods; methods specified for the roles have priority over methods specified in the Data classes.
Interface	An interface can be used to specify a set of operations that must be understood by all objects that play a given role. The concept is not used in BabyIDE. Partly because it did not seem relevant in this simple example. Partly because we usually ended up with a large number of very small interfaces when modeling with OOram. For example, we often needed an interface for read and write operations for a set of variables and another interface for read-only operations. We try to 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].
MVC	Model-View-Controller. A Paradigm that divides an application program into two distinct parts: The Model part that implements the user's Mental Model and the View/Controller part that implement facilities for user interaction with the Model.
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. *"a language mechanism for binding data with methods that operate on that data"* [Wikipedia]
OOram	A tool for modeling with roles.[OOram]
paradigm	Historian of science Thomas Kuhn gave this word its contemporary meaning when he adopted it to refer to the set of practices that define a scientific discipline during a particular period of time. Kuhn himself came to prefer the terms exemplar and normal science, which have more exact philosophical meanings. However, in his book The Structure of Scientific Revolutions Kuhn defines a scientific paradigm as: • *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* • *how the results of scientific investigations should be interpreted*[Wikipedia]

Role	This concept forms a bridge between the compile time and the run-time properties of a system *node* — A role names a node in a network of communicating objects. *responsibility* — A role represents the responsibility that the objects playing it have in a network of communicating objects. *interface* — A role specifies an interface that must be implemented in the classes of all objects that play the role. *methods* — The Role method specified for a role are shared among the classes of all objects that can play it so that polymorphism is suspended for these methods.
Role method	A method that is specified for a role. For each object that will play a role during its lifetime, we put a link to this method into the object’s class.
Role Player	An object that fills the responsibilities of a Role during the execution of a System Operation.
Role Player Class	The instances of a Role Player Class have the capabilities needed to play a given Role.
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 are 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 relationships 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.
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 -> System Operation

Appendix 2: Running the Squeak image with The Shapes animation and the BabyIDE1

Download
http://heim.ifi.uio.no/~trygver/assets/BabyIDE-Squeak.ZIP
into an empty directory. You should find 7 files:

README.txt
this file

commonsense.pdf (~538 796 byte)
the first BabyIDE report. Open with Adobe Acrobat reader.

Squeak.exe (985 600 byte)
Squeak virtual machine.

SqueakV39.sources (17 584 742 byte)
Source file 1 for Squeak

Squeak3.10.gamma.7159.138.changes (7 824 767 byte)
Source file 2 for Squeak.
Your own work in squeak operations are appened to this file.

Squeak3.10.gamma.7159.138.image (18 112 528 byte)
All the objects

Squeak.ini (211 byte)
Configuration file

If you are in Windows, you can open (double click) Squeak.exe to start Squeak with BabyIDE preloaded, then follow instructions there.

If you are not in Windows, go to the Squeak website at
http://www.squeak.org/
and follow instructions there to install Squeak on your computer. Then open squeak on
Squeak3.10.gamma.7159.138.image
note that the **Squeak3.10.gamma.7159.138.changes** file must be in the same directory as the .image-file

Enjoy
-- Trygve

Appendix 3: The mapping between role and object at runtime

The mechanism for the role/object mapping shall transparent for the programmer. There is no magic; the mechanism is fairly simple and we have included the details here for the specially interested:

The compiler has been operated upon to expand the role name inline to a dictionary lookup in the appropriate context at runtime. The *Shape1>>play1*-method looks like this to the programmer:

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

This is expanded inline to:

```
Shape1>>play1
    self displayLarge: '1'.
    (Baby4Context playerForRole: #Arrow12) play12
```

The *Baby4Context*>> playerForRole: class (=static) method searches down the stack until it finds a context that defines the requested role mapping:

```
Baby4Context class>>playerForRole: roleName
    self currentContexts
        do: [:contextb | (contextb includesKey: roleName)
            ifTrue: [^ contextb at: roleName]].
    self error: 'role named: #' , roleName , ' not found'.
    ^ nil
```

and

```
Baby4Context class>>currentContexts
    | babyContexts ctx squeakContexts |
    babyContexts := OrderedCollection new.
    ctx := thisContext.
    squeakContexts := OrderedCollection with: ctx copy.
    [   ctx := ctx findNextHandlerContextStarting.
        squeakContexts add: ctx copy.
        (ctx notNil and: [(ctx tempAt: 1) isKindOf: Baby4Context])
            ifTrue: [babyContexts addLast: (ctx tempAt: 1)].
    ctx notNil]
        whileTrue: [ctx := ctx sender].
    ^babyContexts
```

It remains to note that the context was put on the stack in an Env kick-off statement such as *Shapes4ArrowsCtx executeInContext: [self animateArrows]*. on page 14. We note hat all context classes are subclasses of *Baby4Context* and find:

```
Baby4Context class>>executeInContext: aBlock
        | env |
    env := thisContext sender receiver.
    (self new on: env data env: env) executeInContext: aBlock.
```

The sneaky part is that the sender of the above message must be the instance of the *Env* class and must understand the *data* message. We then do a initial role/object binding:

```
Baby4Context>>on: dataBase env: env
    self initialize.
    data := dataBase.
    environment := env.
    roleMap := IdentityDictionary new.
    self class roleNames do:
        [:key |
        roleMap at: key put: nil].
    self bindRolesToObjects.
    ^self
```

and:

```
Baby4Context>>executeInContext: aBlock
    ^aBlock on: self do: [:ex | ]
```