The DCI Paradigm Implemented in Squeak

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

The DCI paradigm 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.

This report is a commentary on the Squeak implementation of the DCI paradigm. It is written for the systems programmer who experiments with the DCI infrastructure. The systems programmer's task is to create an illusion; the illusion that contexts and roles are first class citizens in the programming universe. The application programmer can then work with networks of communicating roles without bothering about the magic that makes this possible.

We use the Squeak code for the *ArrowsAnimation* as an examples of DCI code and the underlying system code. A video of the animation can be seen here:

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

A Squeak image that runs the animation example as well as the BabyIDE1 programming environment that supports the DCI paradigm can be found here

http://heim.ifi.uio.no/~trygver/assets/BabyIDE-Squeak.ZIP

A report called "The Common Sense of Object Oriented Programming" can be found here:

http://heim.ifi.uio.no/~trygver/2008/commonsense.pdf

An e-mail list for discussions on different DCI implementations and other subjects is:

object-composition - at - googlegroups.com

We'll comment upon the Squeak implementation of the DCI paradigm in the following sections.

Table of Contents

1 The Data Perspective

The classes in the data perspective specify the static aspects of the system. System state is stored in Data class attributes and associations. This perspective also includes the internal behavior of the Data classes (System behavior is specified in the Context and Interaction perspectives.)

```
StarMorph subclass: #BB2Star
    instanceVariableNames: #(smallExtent bigExten)
    methods: #(displayLarge: displayNormal flash)
CircleMorph subclass: #BB2Circle
    instanceVariableNames: #(smallExtent bigExtent)
    methods: #(displayLarge: displayNormal flash)
LineMorph subclass: #BB2Arrow
    instanceVariableNames: #(startMorph endMorph')
    methods: #(grow)
```

We need a class for managing the data

```
Object subclass: #BB2Database
    instanceVariableNames: #(stars circles arrows)
    methods: #(addShape: removeShape: newArrow removeAllArrows)
```

There is also the state and behavior inherited from the superclasses. Notice that these classes make stars and circles what they are, but they do not say anything about how their instances are used.

2 The Context Perspective

Classes 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.

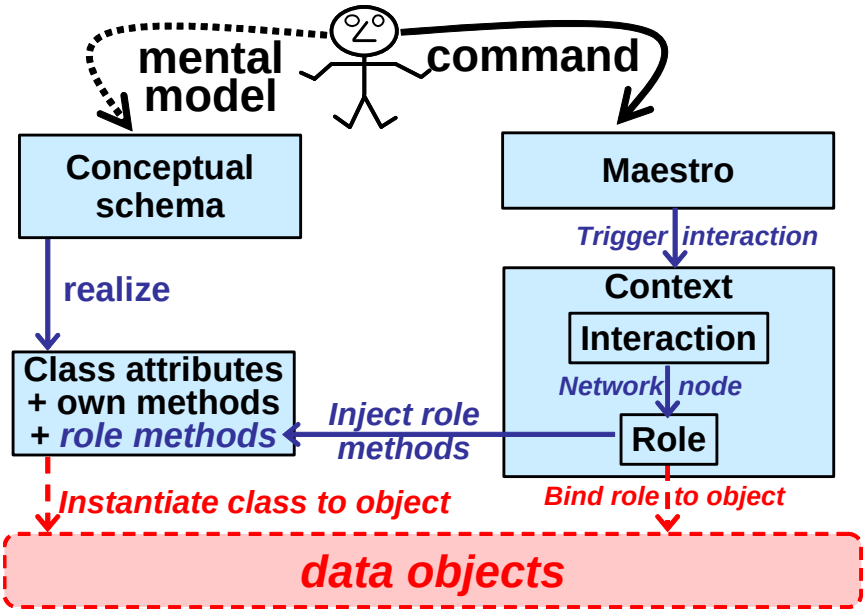
2. 1 The Context as an Extension of the VM

The context is a funny kind of animal because it spans the chasm between compile time and run time. I sometimes write the code first and then try to figure out what it means afterwards The root context class, *BB1Context*, is a good example. The code wasn't easy to write and I had many working versions before I was satisfied.

I struggled quite a bit with fitting the Context into the DCI architecture. Figure 1 is a copy of figure 20 in *The Common Sense of Object Oriented Programming*[1]. We here see the context as something that encapsulates roles and interactions.

1. http://heim.ifi.uio.no/~trygver/2008/commonsense.pdf. version of September 11, 2008.

Figure 1: The complete DCI paradigm (from *Common Sense...*)[1]



Serge Beaumont came up with much better model. He looked at it this way[2]: *"Context is shared state that is *encapsulated* by its roles! It's an "object" with its skin inside out and its state in the middle! I've enclosed a picture of what I mean"*. Serge added: *"A collaboration is a funny beast, isn't it? Its behavior has the interfaces pointing *inward*... I guess I'll get used to it"*. and: *"A role is more strongly bound to its collaboration than its object, and all the roles encapsulate their shared state, the context. "*.

Serge's enclosed picture is shown in figure 2.

Figure 2: Serge's Context as an object with the roles on its periphery.

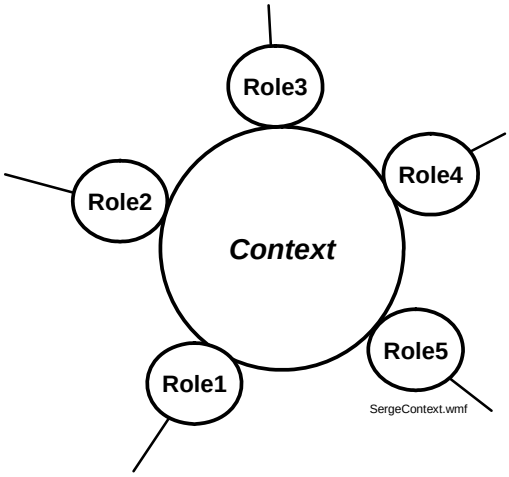


Figure 2 suggests to me that the application programmer sits within the context and works with networks of communicating roles. This gives him full control over the system's runtime behavior. The roles appear to have identity, state, and behavior because they are proxies for the real runtime objects. They also have interfaces; a common simplification is to say that roles *are* interfaces.

---

1. "Maestro' has been renamed to 'Env(ironment)'.
2. Quoted with Serge's kind permission.

A systems programmer (e.g., me) creates this illusion. This commentary is written for the systems programmer and contains details that must be outside the application programmer's mental model.

The role is a compile time concept and roles only exist at compile time. At runtime, the Context acts as a VM extension when it replaces the roles with real objects. .

2. 2 The Context classes

There is currently one Context class for each use case because we specify system behavior independently for each use case. All context classes must be subclasses of the *BB1Context* class:

```
Object subclass: #BB1Context
    instanceVariableNames: #(data roleMap)
```

The *data* attribute is needed because a context is an external view on the data. The *roleMap* is a dictionary that maps role names to objects. It is accessed at runtime to replace the role names used in the code with actual objects at runtime.

The class (static) side of a Context class specifies the roles and their connectors, the instance side specify the binding of roles to objects.

```
BB1Context subclass: #BB2ArrowsCtx
    instanceVariableNames: #().
```

The class and instance methods are discussed in the following subsections.

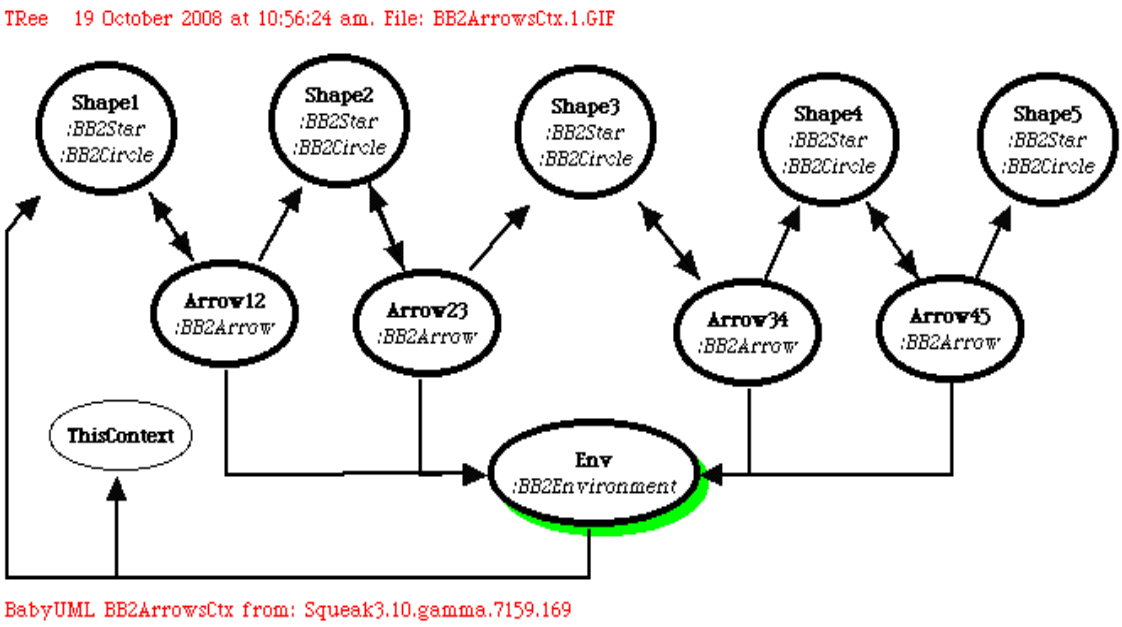
2. 3 The class (static) side of a context class specifies roles, role structure, and role diagram

A class (static) method specifies role names and role relations in a dictionary:

```
BB2ArrowsCtx class>>roleStructure
    ^super roleStructure
    at: #Arrow23 put: #(#Shape3 #Shape2 #Env );
    at: #Shape4 put: #(#Arrow45 );
    at: #Arrow12 put: #(#Shape1 #Shape2 #Env );
    at: #Shape5 put: #();
    at: #ThisContext put: #();
    at: #Shape3 put: #(#Arrow34 );
    at: #Arrow34 put: #(#Env #Shape4 #Shape3 );
    at: #Env put: #(#Shape1 #ThisContext );
    at: #Shape2 put: #(#Arrow23 );
    at: #Shape1 put: #(#Arrow12 );
    at: #Arrow45 put: #(#Env #Shape4 #Shape5 );
        yourself.
```

The corresponding diagram is shown in figure 3. All persistent context data are stored in class methods because methods are automatically persistent in Smalltalk. (The positions of role symbols is specified in a method *BB2ArrowsCtx class>>rolePositions* and the connector breakpoints in *BB2ArrowsCtx class>>linkBreakPoints*. All three methods return a dictionary. The methods are created automatically when the context diagram is edited.)

Figure 3: Role context diagram.



2. 4 The context instance specifies algorithms for binding roles to objects

The default role -> object binding done by a dictionary-like lookup in the context instance:

```
BB1Context>>at: roleName
    ^ self
        at: roleName
        ifAbsent:
            [self error: roleName , ' role not bound to data object.'].
```

and

```
BB1Context>>at: roleName ifAbsent: absentBlock
    | value |
    value := roleMap at: roleName ifAbsent: [absentBlock].
    value == self symbolForLazyBinding
    ifTrue:
        [value := self perform: roleName.
        roleMap at: roleName put: value].
    (value isBlock and: [value numArgs = 0])
    ifTrue:
        [^value value]
    ifFalse:
        [^value]
```

The roleMap is normally set when the context is instantiated. This lookup method also shows remnants of earlier experiments. First, the *roleMap* can be reset with symbolForLazyBinding so that the binding happens at the first access. Second, the value can be set to a block so that the object selection is done afresh at every access. (I found this last option far too lively for comfort when I tried it).

The *roleMap* is reset to default values in the *reselectObjectsForRoles* method:

```
1.  BB1Context>>reselectObjectsForRoles
2.      | messName |
3.      self class roleNames
4.          do: [:rNam |
5.          messName := rNam asString asSymbol.
6.          roleMap
7.              at: messName
8.              put: (self
9.                  perform: messName
10.                 ifNotUnderstood: [])].
```

Statement 2 says that it is the responsibility of the class to know the role names. Different subclasses will give different answers, of course.

Statements 5 ensures that the role name is a legal method name.

Statement 9 executes this method to find the object that is to play the role.

Statement 8 through 10 sets the entry in the binding dictionary (roleMap)

A subclass could override these default methods. This would probably be a very bad idea since it could change the concept of role as it is seen by the programmer.

The default role binding is done in a *Context* method with the same name as the role. These methods are defined in the appropriate context subclass. Examples:

```
BB2ArrowsCtx>>Shape1
    ^data anyShape

BB2ArrowsCtx>>Arrow34
    ^data newArrow

BB2ArrowsCtx>>Env
    ^ data environment

BB2ShapesCtx>>Context
    ^self
```

2. 5 The context instance supports dynamic variables

The context is instantiated and put on the stack with a statement

<context class> executeInContext: <aBlock> on: <data>

For example, the arrows animation is triggered from the system environment in this method:

```
1.  BB2Environment>>startArrowAnimation
2.      currentState = #ARROWS ifTrue: [^self].
3.      currentState := #ARROWS.
4.      processSemaphore wait.
5.      [   BB2ArrowsCtx executeInContext: [self animateArrows] on: data.
6.          processSemaphore signal.
7.      ] fork.
```

The animation is done in a separate process to prevent it from hogging the computer. This is essential for making the program as a whole behave properly and has nothing to do with DCI.

The context is instantiated and put on the stack in statement 5:

```
BB1Context class>>executeInContext: aBlock on: data
    (self new on: data)
        executeInContext: aBlock
```

and the following method calls the Squeak primitive that puts the context instance on the stack:

```
BB1Context>>executeInContext: aBlock
    ^aBlock on: self do: [:ex | ]
```

Any method that is called directly or indirectly from *aBlock* can access the objects by their role name with the statement

BB1Context playerForRole: <roleName >

for example

```
BB2ArrowsCtxShape3>>play3
    self displayLarge: '3'.
    self color: Color green.
    (BB1Context playerForRole: #Arrow34) play34
```

The *playerForRole*: method looks for the context by searching down the stack:

```
BB1Context class >> playerForRole: roleName
    self currentContexts
        do: [:contextb | (contextb includesKey: roleName)
            ifTrue: [^ contextb at: roleName]].
    self error: 'role named: #' , roleName , ' not found'.
    ^ nil
```

and

```
BB1Context class>>currentContexts
    "Search stack for all baby contexts (as opposed to other stack contexts),
        return most recent first."
    | 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: BB1Context])
        ifTrue: [babyContexts
            addLast: (ctx tempAt: 1)].
    ctx notNil]
        whileTrue: [ctx := ctx sender].
    ^ babyContexts
```

These relatively simple methods make a context available to all methods that are called directly or indirectly from the method that triggered an execution. *All methods* here means that any

method regardless of package or class or whatever can access an object by its role name as long as its activation record is above the context on the stack.

The main inefficiency in the current Squeak implementation of DCI is in this search of the stack that occurs on every access through a role. There are many obvious optimizations that could be implemented and some not so obvious ones. I do not fall for the temptation to optimize because I want the fundamental mechanism to be clear and reviewable.

Writing (BB1Context playerForRole: #Arrow34) play34 is very clumsy so I have extended the Squeak compiler to make it expand Arrow34 play34 inline into (BB1Context playerForRole: #Arrow34) play34.

There are two permissible syntaxes.The first declares the roles in a so called Pragma at the beginning of the method. This syntax can be used in any method whatsoever:

```
BB2ArrowsCtxShape3>>play3
    <Roles: #(Arrow34)>
    self displayLarge: '3'.
    self color: Color green.
    Arrow34 play34
```

The second syntax assumes that the method is a role method that is compiled in the context of a Context. All roles that are visible from the current methodful role can be accessed.

```
BB2ArrowsCtxShape3>>play3
    self displayLarge: '3'.
    self color: Color green.
    Arrow34 play34
```

The notion of role methods will be discussed further in section 3. 2.

There are several extensions to compiler, debugger, and inspector to make them handle roles. The details are outside the scope of this commentary. An extension of the *Encoder* that extracts roles from a context is shown here as an illustration:

```
1.  Encoder>>init: aClass context: aContext notifying: req
2.      | node n homeNode indexNode |
3.      requestor := req.
4.      class := aClass.
5.      nTemps := 0.
6.      supered := false.
7.      self initScopeAndLiteralTables.
8.      n := -1.
9.      (class isKindOf: BB1RoleTrait)
10.     ifTrue: [
11.         (class roleContextClass collaboratorsFor: (class roleContextClass
    roleNameFromTraitName: class name)) do:
12.         [:roleName |
13.         self scopeTableAt: roleName put: (BB1RoleNode new asVariable: roleName)]].
14.     class allInstVarNames do:
15.         [:variable |
16.         node := VariableNode new
17.                 name: variable
18.                 index: (n := n + 1)
19.                 type: LdInstType.
```

```
20.         scopeTable at: variable put: node].
21.     aContext == nil
22.         ifFalse:
23.         [homeNode := self bindTemp: 'homeContext'.
24.         "first temp = aContext passed as arg"
25.         n := 0.
26.         aContext tempNames do:
27.             [:variable |
28.             indexNode := self encodeLiteral: (n := n + 1).
29.             node := MessageAsTempNode new
30.                     receiver: homeNode
31.                     selector: #tempAt:
32.                     arguments: (Array with: indexNode)
33.                     precedence: 3
34.                     from: self.
35.             scopeTable at: variable put: node]].
36.     sourceRanges := Dictionary new: 32.
37.     globalSourceRanges := OrderedCollection new: 32.
```

This method sets up nodes for the variables available in the current scope. Statements 9 through 13 adds the visible roles to these variables. The new *BB1RoleNode* is a class that knows how to emit code for the context lookup at runtime.

3 The Interaction Perspective

The role context diagram shown in figure 3 specifies the named roles and the directed communication paths between them. In *BabyIDE1*, this diagram *is* code; the structure is edited by editing in the diagram. The code for role structure in the context class is generated automatically from this diagram.

3. 1 Methodless Roles

Role symbols with thin, light gray borders in figure 3 are *methodless roles*, messages to these roles trigger the execution of the own methods in the classes that implement the roles.

The run time binding of roles to objects was discussed in section 2. 4. The classes of objects that can be bound to a role can be deduced by studying the binding methods in the relevant context class. Any code controlling message sends from such roles is hidden within the corresponding data classes.

3. 2 Methodful Roles, Traits

Role symbols with heavy borders in figure 3 are *methodful roles*. These are ordinary roles with the addition of methods are automatically shared by all the role's *using classes*. The using classes are not permitted to override the role methods. This ensures that the role methods tell the whole story about the system behavior.

Methodful roles store their methods in Traits, a mechanism that has been introduced into Squeak by Nathanael Schärli, Stéphane Ducasse, Andrew Black, Adrian Lienhard and others. Traits lets several classes share the same methods as follows:

An executable method in Squeak is an instance of the class *CompiledMethod*. It holds the byte codes and other data that are needed by the VM so that it can interpret the code.

A class object in Squeak hold data that are common to all its instances. One such datum is the *methodDictionary* that binds method names (selectors) to *CompiledMethods*. The *CompiledMethod* itself does not need to know the class because the *CompiledMethod* is always accessed through the *methodDictionary* in the class.

An instance of class Trait is an object with a *methodDictionary*. It knows how to compile source code into *CompiledMethod*s and also how to inject its *CompiledMethod*s into one or more *using classes*.This is done by adding entries to the *methodDictionary* of the using class. The entry itself is unique to the using class, but its value (the *CompiledMethod*) is shared. There is, therefore, no cost in time and very little cost in space when methods are shared in this manner.

Like classes, Traits must have globally unique names in Squeak. Roles are local to the context, so their names need only be unique within their context. Traits are, therefore, named by the combination of context and role names:

```
<context class name><role name>
```

example

```
BB2ArrowsCtxShape5
```

which is the Trait with the role methods for the role named *Shape5* in the *BB2ArrowsCtx* context.

A DCI subclass of *Trait* has an additional instance variable:

```
Trait subclass: #BB1RoleTrait
    instanceVariableNames: 'roleContextClass'
```

This makes it possible for the Trait compiler to access the context and thus find the legal collaborators for any role.

Finally, the using classes are added to the role traits. The using class names are displayed together with the role names in the role context diagram as shown in figure 3.

RoleTrait instances must be created with a reference to the *roleContextClass*. The RoleTrait methods can be compiled in any Squeak browser, the role names are legal variable names:

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

4 The Env(ironment) perspective

A use case is realized by an avalanche of messages flowing through a network of communicating objects. The avalanche must somehow be triggered; a class that specifies such a trigger is called an *Environment* class, *Env* for short. The Environment is outside the DCI paradigm, but it is essential for triggering system behavior. I am not certain that the environment deserves its own perspective. The environment classes are domain classes that could easily fit into the Data perspective. I am working on an algorithm that can determine if a class is an environment class or not. The goal is to distinguish environment classes in the IDE. This has tentatively been done in figure 3 where the Env role has been distinguished by a green shadow.

The Env classes are distinguished by their sending a message of the form:

```
<context class> executeInContext: <aBlock> on: <data>
```