BabyIDE, the first Loke implementation

Trygve Reenskaug
trygver@ifi.uio.no

BabyIDE is an implementation of Loke as a non-intrusive extension of Squeak, a variant of Smalltalk[1]. The implementation forms an executable, multidimensional, conceptual model of Loke that uses Squeak as its medium. A reader of the model can explore its static properties with its objects and their relationships (section A). The reader can also explore Loke's dynamic properties by studying program creation and execution (section C). This section is a linearized, commented, and simplified projection of the Loke model. Alan Kay pointed out the difference between the two media:[2]

> *The ability to 'read' a medium means you can access materials and tools generated by others. The ability to 'write' in a medium means you can generate materials and tools for others. You must have both to be literate. In print writing, the tools you generate are rhetorical; they demonstrate and convince. In computer writing, the tools you generate are processes; they simulate and decide.*

The first version of BabyIDE as a conceptual model is now completed:

BabyIDE is a conceptual model of Loke for inspection and exploration.
It embodies the Loke model in an executable form.

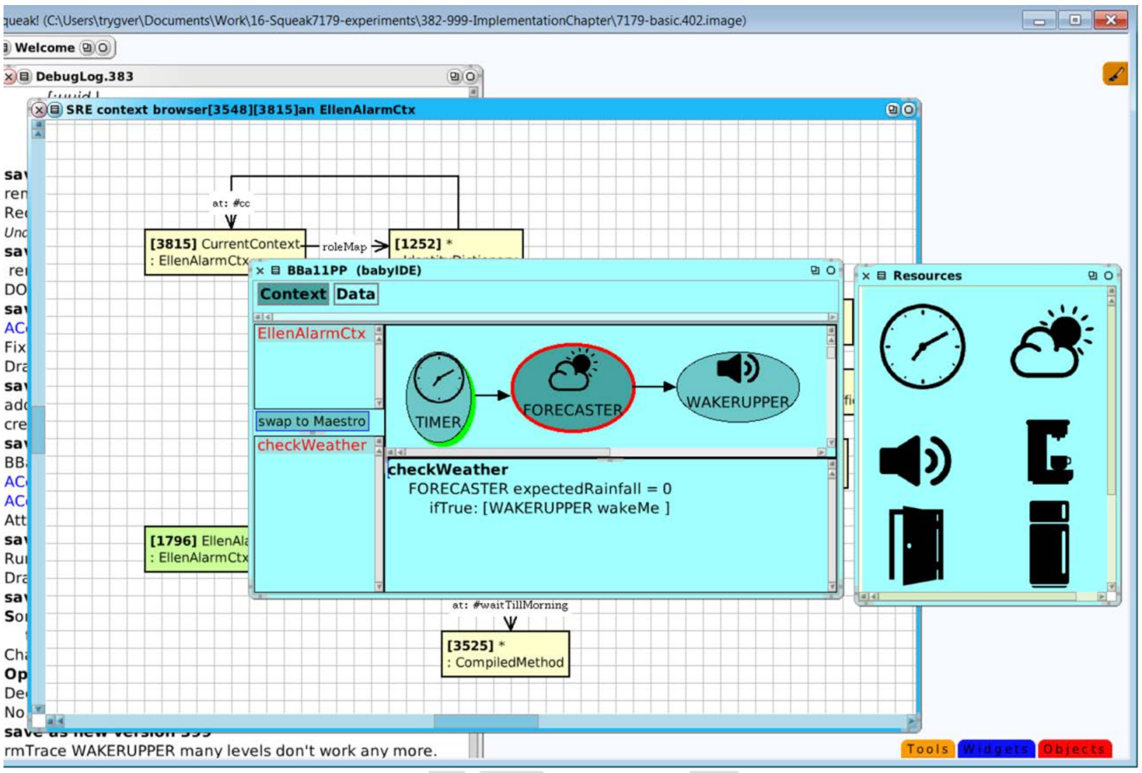
BabyIDE, as an interactive development environment, is still in its infancy. I use it for demonstrating Ellen's smart alarm clock and for exploring other applications. Ellen's programming interface consists of two Squeak windows, as shown in the screen dump of Figure 1. On the right is the Resources window. It is like the desktop of a smartphone with its icons for cached resources. On the left is the BabyIDE window where Ellen composes her program.

A side remark and confession:
This first version of BabyIDE has been programed "by inspiration."
It is neither bug-free, elegant, nor easy to read.
Being a one-man team,
I had to suspend working with the BabyIDE program (fun)
to write this article (a bore).

I'm a nonagenarian; my priorities are mandated by nature. The next step is to use the current BabyIDE to create an elegant, bug-free, and readable second BabyIDE that conforms to the DCI programming paradigm.

[1] BabyIDE works under Squeak version 3.10.2. It is not easily converted to later versions.
[2] http://www.vpri.org/pdf/hc_user_interface.pdf

Figure 1: A screen dump
that includes the BabyIDE personal programming interface.
~~LokeEmbeddedSqueak-4.png~~


queak! (C:\Users\trygver\Documents\Work\16-Squeak7179-experiments\382-999-ImplementationChapter\7179-basic.402.image)
Welcome
DebugLog.383
SRE context browser[3548][3815]an EllenAlarmCtx
at: #cc
[3815] CurrentContext : EllenAlarmCtx
roleMap
[1252] *
BBa11PP (babyIDE)
Context
Data
EllenAlarmCtx
swap to Maestro
checkWeather
TIMER
FORECASTER
WAKERUPPER
[1796] EllenAl : EllenAlarmCtx
at: #waitTillMorning
[3525] * : CompiledMethod
Resources
Tools
Widgets
Objects

```
checkWeather
    FORECASTER expectedRainfall = 0
        ifTrue: [WAKERUPPER wakeMe ]
```

```
rmTrace WAKERUPPER many levels don't work any more.
```

Loke, BabyIDE, Squeak, and Smalltalk are universes of objects and nothing but objects. For example, my Squeak universe of objects was a tangle of some 470.000 objects when I made the screen dump. They represent information of various kinds such as *message*s, *string*s, *collection*s, *stack*s, *compiler*s, and various services. Every Squeak object is an instance of a class (also represented by an object).

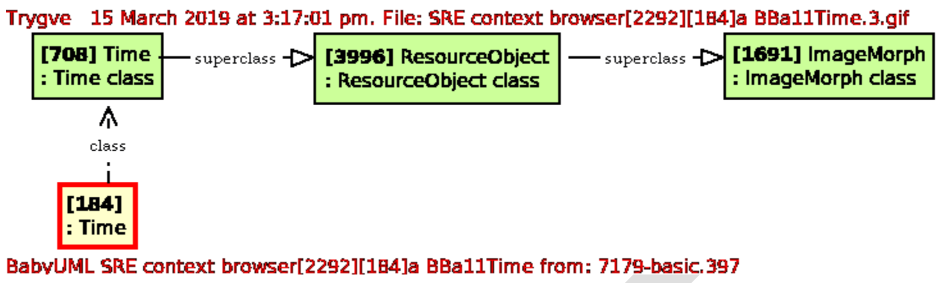
Squeak's many programming tools such as the Inspector and Class Browser give excellent support for thinking and programming in the class abstraction. The *Squeak Reverse Engineering (SRE)* tools support a programmer untangle the tangle by providing tools for creating snapshots of runtime object structures in the role abstraction:[3]

- *SRE Execution Tracer*. Object>>traceRM:levels: is like Transcript>>show: with the addition of the oop that identifies the receiver and prints a dump of the stack to the Transcript, Squeak's standardOut.
- I use it to describe the execution of Ellen's smart alarm in section C.
- *SRE Object Inspector*. A class with its superclasses and anonymous instances is only a partial description of its instances. The SRE Object Inspector shows the state and behavior of an object. The state is shown as the instantaneous value of its instance variables. The behavior is shown as the methods found in its flattened class hierarchy. Figure 4 is an example.
- *SRE Context Browser*. The essence of object orientation is that objects collaborate to achieve a goal. The Context Browser is used to plot a snapshot of a substructure of collaborating objects in an *object diagram*. As an example, Figure 2 shows an object that is an instance of a class that is the subclass of another class and so on. A rectangle represents an object. The first text line shows the object's Squeak identifier, [oop], followed by the object's name if any. The optional second line starts with a colon followed by the name of the object's class. An arrow in the

[3] SRE user manual: http://folk.uio.no/trygver/themes/SRE/BabySRE.pdf

diagram shows a message link that represents an instance or computed variable. Notice the difference between the concrete SRE *as is* reverse engineering documentation and the more usual abstract models like UML diagrams[4].

Figure 2: Object [184] is playing Ellen's time role, as will be shown later.
It is an instance of class Time, a subclass of ResourceObject, a subclass of ImageMorph
~~SRE Context browser[2292][184]a BBa11Time.3.gif~~


Trygve 15 March 2019 at 3:17:01 pm. File: SRE context browser[2292][184]a BBa11Time.3.gif
[708] Time
: Time class
superclass
[3996] ResourceObject
: ResourceObject class
superclass
[1691] ImageMorph
: ImageMorph class
class
[184]
: Time
BabyUML SRE context browser[2292][184]a BBa11Time from: 7179-basic.397

A. Loke objects

BabyIDE uses personal and shared resource objects as servers in client-server architectures where Loke is the client and where the servers are objects offering RESTful, self-explanatory interfaces[5]. I expect that a standard like the *Universally_unique_identifier (UUID)*[6] will provide a unique identifier for each and every object in the world. In BabyIDE, they are accessed through instances of a UUID subclass:

Object subclass: #UUID
instanceVariableNames: 'resource'
Class comment:
An instance of this class represents a personal or shared object.
Subclasses specialize the class for different access technologies;
they specify how to trigger the object's operations and how to access its properties.

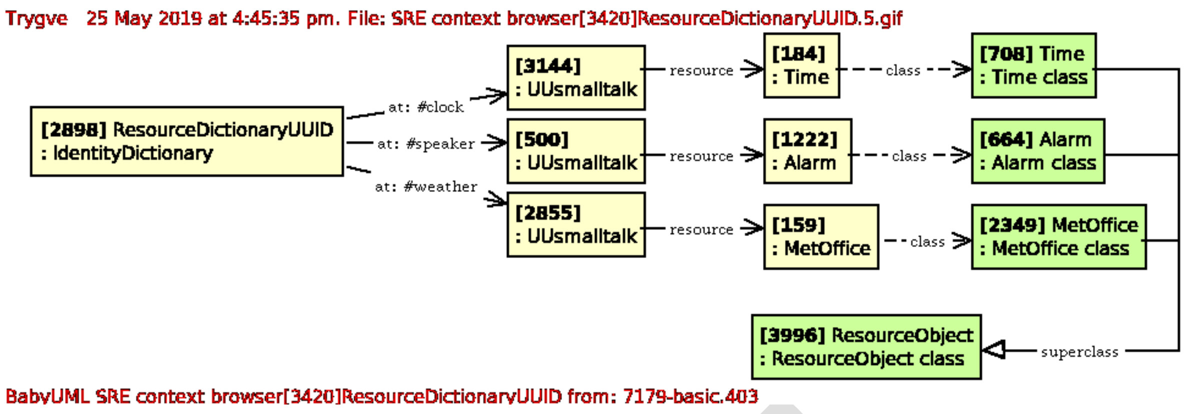
Subclasses of *UUID* implement RESTful message interfaces that include *apiMenuList* and *balloonText*:

Notice that instances of UUID are not wrappers but objects that know how to access the features of their resource whatever its access mechanism.

BabyIDE maintains a cache of objects in a global dictionary: [2898]ResourceDictionaryUUID. The objectDiagram in Figure 3 shows the structure of this Dictionary. The *Dictionary keys* are Universally_unique_identifiers and are visible as icons in the user's window[7]. The *Dictionary values* represent RESTful servers and are instances of a UUID subclass.

[4] https://www.omg.org/spec/UML/2.5.1/PDF
[5] https://en.wikipedia.org/wiki/Representational_state_transfer
[6] UUID: https://en.wikipedia.org/wiki/Universally_unique_identifier
[7]. For the purposes of this document, they are simple Squeak Symbols; #clock, #speaker, and #weather).

Figure 3: Ellen's ResourceDictionarfyUUID
~~SRE Context browser[3420]ResourceDictionaryUUID.5.gif~~

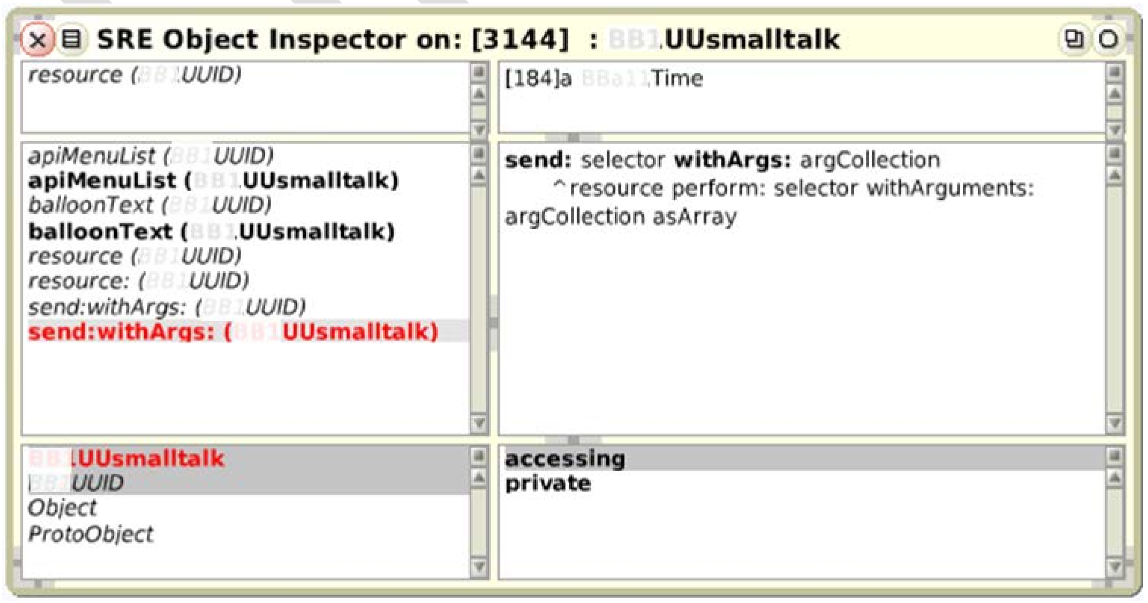

Trygve 25 May 2019 at 4:45:35 pm. File: SRE context browser[3420]ResourceDictionaryUUID.5.gif
[2898] ResourceDictionaryUUID : IdentityDictionary
at: #clock
at: #speaker
at: #weather
[3144] : UUsmalltalk
[500] : UUsmalltalk
[2855] : UUsmalltalk
resource
resource
resource
[184] : Time
[1222] : Alarm
[159] : MetOffice
class
class
class
[708] Time : Time class
[664] Alarm : Alarm class
[2349] MetOffice : MetOffice class
[3996] ResourceObject : ResourceObject class
superclass
BabyUML SRE context browser[3420]ResourceDictionaryUUID from: 7179-basic.403

Ellen accesses her personal objects through instances of UUSmalltalk, a subclass of UUID. The resource instance variable is the sole instance of a ResourceObject subclass[8]Instances of other subclasses will then identify shared objects, and their substructures may be different. In the future, vendors of shared objects will probably supply meta-information with their IoT products in a form that can be automatically converted to a subclass of UUID.

B. Personal objects

Figure 4 shows an *SRE Object Inspector* on Ellen's #clock resource. The bottom-left pane shows its class with superclasses in a multi-select list. Classes UUID and UUsmalltalk are selected and coalesced for the rest of the inspection. The bottom-right pane shows the method categories of the coalesced classes, accessing is selected. The middle-left pane shows its accessing methods. The essential send:withArgs: method is selected; its code is shown in the middle-right pane. The coalesced instance variables are in the upper-left pane. The upper-right pane shows the value, the [184]: a Time object.

Figure 4: Ellen's Time identifier object[9]
~~SRE ObjectInspector on# [3144]-6.png~~



SRE Object Inspector on: [3144] : .UUsmalltalk

resource (.UUID)

[184]a .Time

apiMenuList (UUID)
apiMenuList (.UUsmalltalk)
balloonText (UUID)
balloonText (.UUsmalltalk)
resource (.UUID)
resource: (.UUID)
send:withArgs: (.UUID)
send:withArgs: (UUsmalltalk)

```
send: selector withArgs: argCollection
    ^resource perform: selector withArguments:
argCollection asArray
```

.UUsmalltalk
UUID
Object
ProtoObject

accessing
private

[8] BabyIDE has to be slightly modified if a ResourceObject class shall have more than one instance.
[9] The screen dump pictures has been edited to cover class name prefixes not used in this article

We see from Figure 3 that like all personal objects, class Time is ultimately a subclass of ResourceObject:

```
ImageMorph subclass: #ResourceObject
 instanceVariableNames: ''    Class comment
      There is one instance of each subclass that is uniquely identified
 by its resource ID, which is hardcoded in its resourceID method.
```

Ellen or her mentor program her personal classes as subclasses of ResourceObject. For example, object [184] is an instance of class TIME:

```
ResourceObject subclass: #Time
    instanceVariableNames: ''

"The API method category: "
Time>>delayFor: seconds
    "Wait for the given number of seconds. "
    (Delay forSeconds: seconds) wait.
Time>>waitUntil: timeString
    " Wait until the clock is 'hh:mm'. e.g.: TIMER waitUntil: '06:00'."
    | secondsDelay |
    secondsDelay := ((Time readFrom: (ReadStream on: timeString))
                               subtractTime: Time now) asSeconds.
    secondsDelay < 0 ifTrue:
        [secondsDelay := secondsDelay + (24*60*60)].
    (Delay forSeconds: secondsDelay) wait.

" The accessing method category: "
Time>>apiMenuList
    | list source |
    list := OrderedCollection new.
    (self class organization listAtCategoryNamed: #API) do:
        [:methSel |
        source := self class sourceCodeAt: methSel.
        list add: {methSel asString. (source copyUpTo: Character cr) asString.}].
    ^list
Time>>balloonText
    | strm |
    strm := TextStream on: Text new.
    strm nextPutAll: 'Time' asText allBold.
    ^strm contents
Time>>defaultRoleName
    ^#TIMER ''
Time>>resourceID
    ^#clock " Instead of a universal ID "
```

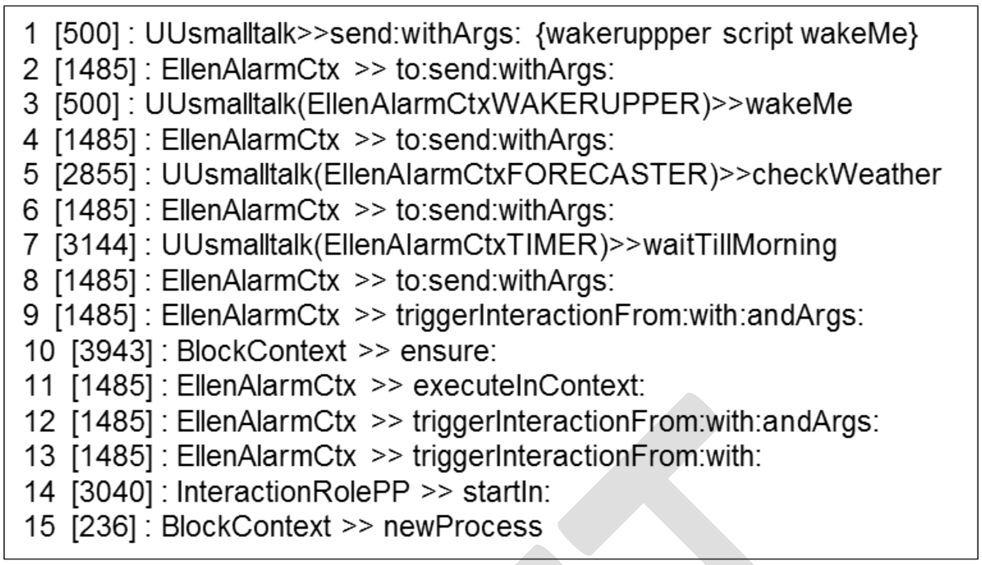
C. A BabyIDE execution

As is the case for most variants of object oriented languages, a BabyIDE execution takes the form of a stream of messages flowing through participating objects. I used the SRE traceRM:levels: tool to capture a trace of the stack at the point in the execution where the WAKERUPPER sounds the alarm:

```
WAKERUPPER >>wakeMe
    WAKERUPPER traceRM: 'traceRM' levels: 50.
    WAKERUPPER soundAlarm.
```

When the execution of Ellen's demo reaches this stage, it prints the stack15 deep in the Transcript, Squeak's standard output window (Figure 5).

Figure 5: The stack
~~SRE-traceRM-stack.png~~



```
1  [500] : UUsmalltalk>>send:withArgs:  {wakeruppper script wakeMe}
2  [1485] : EllenAlarmCtx  >> to:send:withArgs:
3  [500] : UUsmalltalk(EllenAlarmCtxWAKERUPPER)>>wakeMe
4  [1485] : EllenAlarmCtx  >> to:send:withArgs:
5  [2855] : UUsmalltalk(EllenAlarmCtxFORECASTER)>>checkWeather
6  [1485] : EllenAlarmCtx  >> to:send:withArgs:
7  [3144] : UUsmalltalk(EllenAlarmCtxTIMER)>>waitTillMorning
8  [1485] : EllenAlarmCtx  >> to:send:withArgs:
9  [1485] : EllenAlarmCtx  >> triggerInteractionFrom:with:andArgs:
10  [3943] : BlockContext >> ensure:
11  [1485] : EllenAlarmCtx  >> executeInContext:
12  [1485] : EllenAlarmCtx  >> triggerInteractionFrom:with:andArgs:
13  [1485] : EllenAlarmCtx  >> triggerInteractionFrom:with:
14  [3040] : InteractionRolePP >> startIn:
15  [236] : BlockContext >> newProcess
```

Smalltalk, and thus Squeak, has many advanced features such as its inherent reflection and the concrete realization of its own conceptual model. For example, I could print the above list because the stack is an accessible linked list of context objects. BabyIDE leans heavily on this and other advanced features as will be seen in the following.

C.1. level 15 [236] : BlockContext >> newProcess

```
BlockContext>>newProcess
    "Answer a Process running the code in the receiver. The process is not
    scheduled."
    <primitive: 19>
```

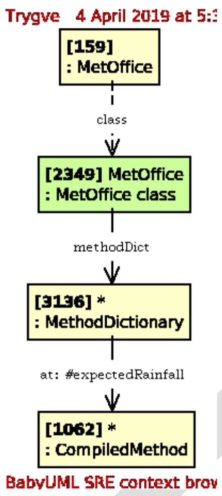
I opened Ellen's personal BabyIDE process with a World menu command. This stack frame has receiver = [236], a new BlockContext object.

C.2. level 14 [3040] : InteractionRolePP >> startIn:

```
InteractionRolePP >>startIn: startRole
    | w context |
    [
        color := diagram color.
        diagram color: Color green.
        (w := self world) ifNotNil: [w doOneCycle].
        context := self diagram model contextBrowser selectedClass new.
        context triggerInteractionFrom: self name with: startRole.
        diagram color: color.
        (w := self world) ifNotNil: [w doOneCycle].
    ]
        forkAt: Processor userBackgroundPriority.
```

After programming it, I started Ellen's smart alarm clock with the *Cue 'waitTillMorning'* menu command in the Context diagram's TIMER symbol: [3040] :IntercationRolePP. The cue is a regular Squeak object and a regular Squeak message. The Ellens and Antons of this world have the illusion that the object encapsulates its methods and invokes them in response to received messages, an illusion that is sustained in the SRE ObjectInspector (Figure 4). In reality, objects delegate to their class to compile, store, and execute methods (Figure 6). Like all Squeak objects, object [159] has a link to its class: [2349]MetOffice. This class object has an instance variable named methodDict. This Dictionary binds selectors (messages) to CompiledMethods, Squeak methods in executable form. Here is another example of that in Squeak, everything is represented by an object and that an object structure can go across different abstractions.

Figure 6: An object's methods are stored in its class.
~~SRE context browser[359][159]a BBa11MetOffice.3.gif~~



A significant side effect of the startIn - method was that it created an instance of Ellen's Context class, [1485] Maestro : EllenAlarmCtx:.that performs many tasks during the execution (Figure 5).

```
Object subclass: #Context
    instanceVariableNames: 'roleMap'

Context subclass: #EllenAlarmCtx
 instanceVariableNames: ''
```

The class side declares the roles and their structure as a method that returns a Dictionary:

```
EllenAlarmCtx class>>roleStructure[10]
    ^super roleStructure
        at: #TIMER put: #(#FORECASTER );
        at: #FORECASTER put: #(#WAKERUPPER );
        at: #WAKERUPPER put: #();
        yourself.
```

BabyIDE generates and compiles the code for this method automatically when Ellen edits her Context diagram. The declaration is later used by the compiler to find role names and permissible message links. For example, a TIMER roleScript can send messages to FORECASTER but not to WAKERUPPER.

[1485] Maestro : EllenAlarmCtx forms the environment for the execution of roleScripts. It has one essential instance variable, roleMap, a Dictionary that maps role names to resource objects at runtime. All roles are mapped together in an ensemble of methods to ensure consistency:

```
Context>>remap
    "Map all roles to a Data object."
    self resetRoleMap.
    self class roleNames do:
        [:roleName | "All roles are mapped together."
            roleMap at: roleName put:
                (self perform: roleName ifNotUnderstood: [nil]) "Execute the mapping method."
        ].
    self checkRoleMap
```

[10] The method represents the BabyIDE way of obtaining persistent objects without depending on a database.

A method named after each role maps it to its roleplaying object, e.g.:

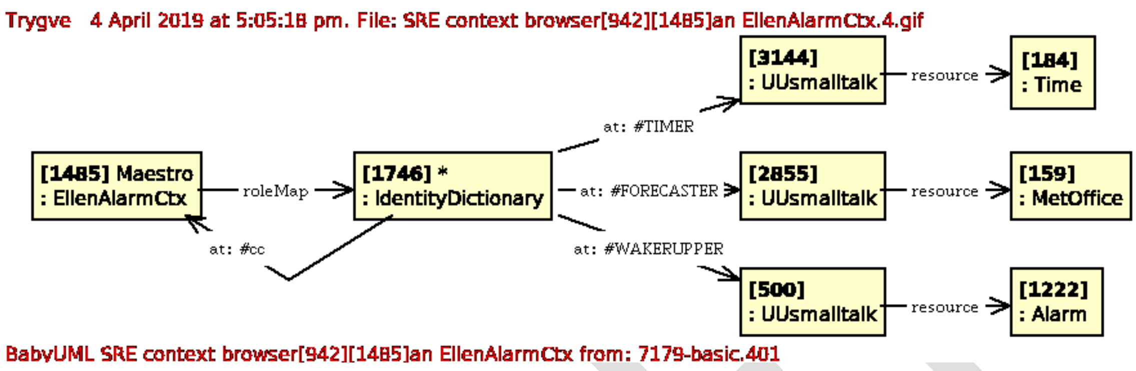
```
    EllenAlarmCtx>>WAKERUPPER
        ^ResourceDictionaryUUID at: #speaker ifAbsent: [nil]
```

When Ellen moved resource icons into her Context, BabyIDE created the roles with their default names automatically.

As part of its initialization, the Maestro executed Context>>remap to bind roles to objects. The result was the ephemeral object structure in Figure 7.

Figure 7: Maestro runtime object structure

~~SRE context browser[942][1485]an EllenAlarmCtx.4.gif~~


Trygve 4 April 2019 at 5:05:18 pm. File: SRE context browser[942][1485]an EllenAlarmCtx.4.gif
[1485] Maestro : EllenAlarmCtx
roleMap
[1746] * : IdentityDictionary
at: #cc
at: #TIMER
at: #FORECASTER
at: #WAKERUPPER
[3144] : UUsmalltalk
[2855] : UUsmalltalk
[500] : UUsmalltalk
resource
resource
resource
[184] : Time
[159] : MetOffice
[1222] : Alarm
BabyUML SRE context browser[942][1485]an EllenAlarmCtx from: 7179-basic.401

Note the "secret" role #cc. Every Context is initialized with this role; BabyIDE uses it on level 8 to find the current player of a given role.

C.3. level 13 [1485] : EllenAlarmCtx >> triggerInteractionFrom:with:

```
Context>>triggerInteractionFrom: triggerRoleName with: selector
    ^self triggerInteractionFrom: triggerRoleName with: selector andArgs: {}
```

C.4. level 12 [1485] : EllenAlarmCtx >> triggerInteractionFrom:with:andArgs:

```
Context>>triggerInteractionFrom: triggerRoleName with: selector andArgs: args
    selector numArgs = args size ifFalse: [^self error: 'Number of arguments mismatch'].
    " The Context lives on the stack during the execution of a application: "
    self executeInContext: " Enter the world of roles and scripts. "
        [self remap.
        ^(roleMap includesKey: triggerRoleName)
            ifTrue:
                [^self to: triggerRoleName send: selector withArgs: args]
            ifFalse:
                [self inform: 'Data object for role named ' , triggerRoleName , ' is undefined. Interaction not started.'.
                ^nil]]
```

The framed block will be executed in level 9.

The Context instance, [1485] Maestro: EllenAlarmCtx, was created on level 14. Now is the time to put it to work.

C.5. level 11 [1485] : EllenAlarmCtx >> executeInContext:

```
Context>>executeInContext: aBlock
    ContextStack pushContextStack: self.
    aBlock ensure: [ContextStack popContextStack].
```

Class ContextStack is a global stack of Contexts: A new Context instance is put on the stack when the execution of a Context starts and is popped when it ends. The

ContextStack also forwards some messages to the CurrentContext, which is the Context on the top of the stack.

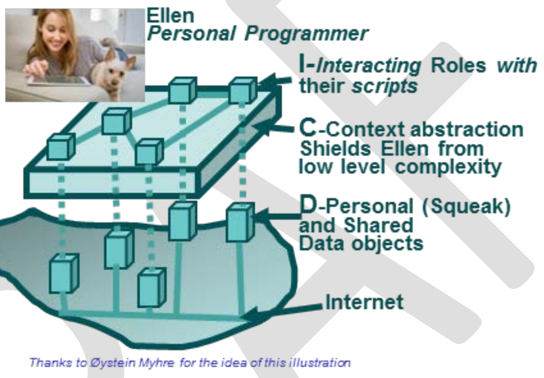
C.6. level 10 [3943] : BlockContext >> ensure:

```
BlockContext>>ensure: aBlock
    "Evaluate a termination block after evaluating the receiver, regardless of whether the receiver's
evaluation completes."
    <primitive: 198>
```

C.7. level 9 [1485] : EllenAlarmCtx >> triggerInteractionFrom:with:andArgs:

We have digressed in levels 11 and 10 to initialize the Maestro before we execute the inner (framed) block in this method from level 12. We leave the level of Squeak messages and methods and enter the higher abstraction level of roles and roleScripts, as illustrated in Figure 8. This higher level lets Ellen work in the role abstraction and protects her from the intricacies of Squeak with its classes.

~~Figure 8: The communicating roles in Loke are on a new abstraction level.~~
~~DCI-abstractionLayer-4.png~~


Ellen
Personal Programmer
I-Interacting Roles with their scripts
C-Context abstraction Shields Ellen from low level complexity
D-Personal (Squeak) and Shared Data objects
Internet
Thanks to Øystein Myhre for the idea of this illustration

C.8. level 8 [1485] : EllenAlarmCtx >> to:send:withArgs:

```
Context>>to: roleName send: selector withArgs: argCollection
    | receiver roleClass compiledMethod |
    receiver := roleMap at: roleName.
    roleClass := self class roleClassForRoleName: roleName. "script repository class"
    (roleClass notNil and: [(compiledMethod := roleClass compiledMethodAt: selector) notNil])
    ifTrue:
        [" roleScript exists, execute it. "
        ^receiver withArgs: argCollection asArray executeMethod: compiledMethod]
    ifFalse:
        [(receiver isKindOf: UUID)
        ifTrue: "remote receiver, the UUID object will do the right thing"
            [^receiver send: selector withArgs: argCollection]
        ifFalse: "send regular Sqeak message"
            [^receiver perform: selector withArguments: argCollection asArray]]
```

All message sends in the Role abstraction are handled by this method. The method transforms a message send into specialized messages to roles and objects, shared or personal. The code first tries to find a CompiledMethod for a roleScript and executes it if it exists. Else, the method forwards the message to a UUID or a regular Squeak object.

Context>>to: roleName send: selector withArgs: argCollection *is the key to the illusion that all messages are handled the same way independent of the protocol used for message transmission and the nature of the receiver*

C.9. level 7 [3144] : UUsmalltalk(EllenAlarmCtxTIMER)>>waitTillMorning

```
TIMER>>waitTillMorning
    TIMER waitUntil: '06:00'.
    FORECASTER checkWeather
```

There is something strange here. The class of the message receiver is EllenAlarmCtxTIMER. Where does this class come from, and what does it do? The question needs a long answer. We saw in Figure 6 and on level 14 that a Squeak method is compiled in the context of a class, is stored in the methodDict of that class, and is executed in the context of an instance of that class. This mechanism breaks down for roleScripts. A role is a name and not an object: There is no class, and the role is late-bound to an object at runtime. Finally, if the object is accessed through the Net, its implementation is inaccessible. Any implementation of Loke must deal with this dilemma. First, roleScripts can't be compiled as regular Squeak methods, so a new way has to be found. Second, a regular method is stored in the object's class. Here, there is no known class, and a new home for roleScripts has to be found. Third, regular methods are executed in the context of the object. Here, there is no object, and a new home for executing roleScripts has to be found.

The compilation of roleScripts

The roleScript compiler namespace includes the names of the current Role and the Roles that are visible from it. The compiler is a modified Squeak compiler that compiles roleScripts in two steps. First, it transforms Role names to regular Squeak code. Second, it compiles the resulting Squeak code in the usual way.

Ellen's code:

```
TIMER>>waitTillMorning
    TIMER waitUntil: '06:00'.
    FORECASTER checkWeather
```

is first transformed to:

```
TIMER>>waitTillMorning
    (ContextStack playerForRole: #cc)
        to: #TIMER
        send: #waitUntil:
        withArgs: {'06:00'}.
    (ContextStack playerForRole: #cc)
        to: #FORECASTER
        send: #checkWeather
        withArgs: {}
```

We know the ContextStack and the hidden #cc-role from level 11. (ContextStack playerForRole: #cc) is an inefficient way of finding the current Context, the Maestro. After that, it's the Context>>to:send:withArgs: method known from level 8 that sends the messages called for in Ellen's code.

The output from the compiler is a CompiledMethod that is independent of the class that compiled it. A CompiledMethod can, therefore, be stored in any class as long as it has a unique name that makes it possible to retrieve it when needed.

A Squeak CompiledMethod without reference to instance variables is pure behavior.
It can be stored in any class and be executed in the context of any object.

The storing of compiled roleScripts

Context class names are unique within Squeak, and role names are unique within a Context. BabyIDE stores roleScripts (CompiledMethods) in hidden classes named by the unique concatenation <Context name><role name>, e.g. EllenAlarmCtxTIMER. These classes are artifacts of the BabyIDE implementation and are exceptional classes with no instances, no own methods, and the programmer can't see them.

The execution of roleScripts

The execution of the roleScript was triggered in the key method Context>>to: roleName send: selector withArgs: argCollection in level 8.

C.9.1. level 6 [1485] : EllenAlarmCtx >> to:send:withArgs:

see level 8

C.9.2. level 5 [2855] : UUsmalltalk(EllenAlarmCtxFORECASTER)>>checkWeather

```
FORECASTER>>checkWeather
    FORECASTER expectedRainfall = 0
        ifTrue: [WAKERUPPER wakeMe ]
```

see level 7

C.9.3. level 4 [1485] : EllenAlarmCtx >> to:send:withArgs:

see level 8

C.9.4. level 3 [500] : UUsmalltalk(EllenAlarmCtxWAKERUPPER)>>wakeMe

```
WAKERUPPER>>wakeMe
    WAKERUPPER traceRM: 'wakeruppper script wakeMe' levels: 50.
    WAKERUPPER soundAlarm.
```

see level 7

C.9.5. level 2 [1485] : EllenAlarmCtx >> to:send:withArgs:

see level 8

C.9.6. level 1 [500] : UUsmalltalk>>send:withArgs: {wakeruppper script wakeMe}

```
WAKERUPPER >>wakeMe
    WAKERUPPER traceRM: 'traceRM' levels: 50.
    WAKERUPPER soundAlarm.
```

Finally, BabyIDE dumped the stack on the Transcript, and the alarm sounded.