BabyIDE user guide

Trygve Reenskaug

Table of contents

BabyIDE image user guide

BabyIDE is my program repository that includes all my Squeak programs. This user guide is made up of many snippets from a variety of sources without careful editing or review. Some of the texts have been hard to find. I hope you find this guide useful for navigating the image.

1 BabyIDE Image

BabyIDE1 Squeak image
http://dx.doi.org/10.17632/5xxgzv7fsp.1

Download the ZIP and unzip into an empty folder.
In Win 7/10, double-click Squeak.exe.
In other OS, open the image according to the rules of your system.

Note. Use BabyIDE to export a program as an HTML-file
BabyIDE>>window menu>>printHTML for this app
These reports are referenced in the *read the program* subsections of the different program descriptions.

2 Ellen's smart alarm

This is the smart alarm clock program that Ellen wrote in a demo. The clock only wakes her if the weather forecast promises a dry day. The *Personal Programming and the object computer article* describes this example with its code in detail:
(https://doi.org/10.1007/s10270-019-00768-3) in *section 2: Novice programming* and in the appendices.

The completed demo is open in the image when you open it.

2.1 Run the demo

World menu>>open…>>BBa11: PP

opens Ellen's IDE, a personal programming version of BabyIDE that is created for demonstration and exploration purposes. The IDE starts with Ellen's program on the screen.

There are 2 projections *Context* and *Data*. The *Data* projection specifies Data classes; their instances are Ellen's predefined objects. Objects that

reify their API and that provide the metadata supporting the illusion of RESTful servers.

The Context projection has the usual 2 views: The *Context class view* is automatically generated when Ellen moves an object into her Context. The *Context interaction view* is Ellen's programming tool, where she augments the roles with her role methods.

2.2 To do a demonstration:

1. open *Context>>EllenAlarmCtx>>interaction view*. In the role diagram (outside the roles), do *yellowmenu>>open data window*. This opens a new window with Ellen's resource objects. Here it's only 3, in a reality it should be hundreds.
2. For each role, do *yellowmenu>>remove role*
3. and the demo can start as described in the article

2.3 Read Ellen's program

View for reading
View for printing

3 DCI Examples

The examples were part of the first release of BabyIDE. They are first shots at the implementations. The hope was that each of them should form a starting point for discussing the solution to find better ones. This did not happen; there was no noticeable interest in BabyIDE at the time. (Caused by my feeble marketing)

3.1 BB2Shapes

An animation of a universe of interacting objects

3.1.1 Run the program:

World menu>>open…>>BB2: Shapes
yellow menu in the window:

- *animate shapes* illustrates a universe of objects where objects are added and removed. The shape of an object represents its class,
- *animate roles* illustrates repeated execution of the same operation: Different objects are allocated to roles and interact according to the same DCI algorithm,
- *animate chaos arrows* illustrates chaotic message flow, controlled by some unknown program.

(ALT-. stops the execution if BB2Shape hangs).

3.1.2 Edit the program:

World menu>>open…>>BB1: IDE>>BB2Shapes. BabyIDE is a browser that lets you edit any program written according to the DCI (Data, Context, Interaction) paradigm. More details about DCI and this browser is in the Personal Programming article
https://doi.org/10.1007/s10270-019-00768-3

3.1.3 Read the program

reader-friendly version
printer-friendly version

3.2 BB3Greed

This example illustrates a program with 8 Data classes and executions with loops within loops.

Greed is an exceedingly dull game, yet it is well suited as a programming exercise. Tom Love gave it as a programming exercise for an OOPSLA-89 workshop. There were many submissions of working programs. (I had one of them based on role modeling. Tom has later reported that my program didn't work. It actually did work, but I deemed its user interface to be too ugly for me to demo it at the time at our workshop. You can now run it and judge for yourself).

The use case is here:
http://fulloo.info/Examples/SqueakExamples/BB3Greed/index.html

The rules are:

Greed is a dice game played by two or more players. The game's object is to tally points from the rolls of the die and be the first player to score 5000 points. There are five dice in the game; they are rolled from a cup.

To enter the game, a player must score at least 300 points on the first roll of his turn; otherwise, the player is considered "bust." If he goes "bust," he must wait until his turn to roll again. If his first roll does produce 300 or more points, the player then has the option of stopping, thus keeping the initial score, or continuing. To continue, the player rolls only the die that has not yet scored in his turn. A player may continue rolling until all the dice have scored or until he is "bust." Except for the entry roll, a "bust" is when an individual roll produces no points. The player may stop and keep his score after any roll, as long as he is not "bust."

Each roll of the dice is tallied as follows:

Three of a kind score 100 x face value of one of the three dice. If the three of a kind are 1s, then it is scored as 1000. 22234 = 200 points; 43414 = 400 + 100 = 500 points. Single 1s and 5s score 100 and 50 points, respectively.

3.2.1 Run the program:

World menu>>open…>>BB3: Greed
The top row of the window has a comment pane, a *Play* button, and an array of 5 numbers representing the 5 dice. Dice that have scored are red. Below are 3 panes, one for each automatic player who has its own strategy for playing the game.
Press *Play* to start the game and observe the dice and the results in the comment pane.

3.2.2 Edit the program:

World menu>>open…>>BB1: IDE>>BB3Greed. There are 3 projections: *Context*, *Controller*, *Data*.

3.2.3 Read the program

reader-friendly version
printer-friendly version

3.3 BB4Plan

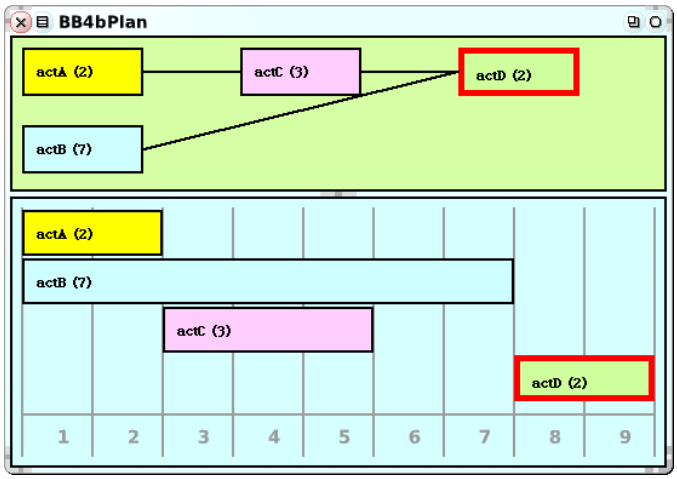
This example is a rudimentary activity planning application.

There is no user data input; the example network is hardcoded.

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

BB4aPlan: A conventional application with MVC, without DCI.
BB4bPlan: The application coded with MVC and DCI.

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



Note. BB9Planning: An Activity Planning Example (aka Prokon) is a more elaborate version that demonstrates the use of MVC and DCI in combination.

3.3.1 Run the program:

World menu>>open…>>BB4b: Plan
A window with 2 panes open. Three menu operations are in the upper pane:

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

3.3.2 Edit the program:

World menu>>open…>>BB1: IDE>>BB4bPlan. There are 4 projections: *Context*, *Controller*, *Data, View*. Notice that the object structure is traditional MVC.

3.3.3 Read the program

BB4aPlan (no DCI):

reader-friendly code
printer-friendly code

BB4bPlan (DCI version)

reader-friendly code
printer-friendly code

3.4 BB5Bank

This program illustrates a simple Context with two interacting Roles.

James Coplien has implemented a DCI infrastructure in C++. He has also developed a simple program that illustrates the DCI essentials. *BB6Bank* is the Squeak/DCI code for roughly the same problem.

The use case is that a human uses an Automatic Teller Machine (ATM) to transfer funds from her checking account to her savings account. She knows that her checking account is account number 1111 and that her savings account is account number 2222. The amount she wants to transfer on this occasion is $500. The use case operation is the transfer of money from one account to another. We follow the DCI paradigm and organize our code in the two essential projections; Data and Context. The *Interaction* is the dynamic part of the Context. We also have a Testing projection where the *BB5Testing* class drives the program.

In the core of banking, we find the *General Ledger (GL)*, a collection of transactions where each transaction documents the transfer of a value from one account to another (or "transfer a quantity from one resource to another" in modern parlance). Transactions are immutable by law and international conventions. Like any database record, a transaction has no behavior. The system behavior (use case) in our example is constructing a transaction from the input data. This behavior is not part of the GL (what the system IS) but is part of the bank transfer application (what the system DOES).

An essential attribute of an account is its balance, the funds available to its owner. It is a derived attribute that is computed by summing over selected GL transactions. We see our account objects as caches on the GL. Our program's task is to construct a new transaction while applying the appropriate checks, update the account caches, and add the new transaction to the GL.

There are two versions of the program. In *BB5aBank*, the GL is omitted to make a version suitable for use in short presentations. A companion version, *BB5bBank*, includes updating the GL at the end of a successful transfer.

3.4.1 Run the demo program:

World menu>>open…>>BB5a: Bank runs a test. The response is a window "*Test1 OK*". Press OK and get the error window: "*Insufficient funds*". Click OK and get: "*error: Test2 OK, it has failed as expected".*

3.4.2 Edit the demo program:

World menu>>open…>>BB1: IDE>>BB5aBank. There are 3 projections: *Context*, *Data, Testing.*

3.4.3 Read the demo program

BB5aBank (no General Ledger):

reader-friendly version
printer-friendly version

BB5bBank (with General Ledger)

reader-friendly version
printer-friendly version

3.5 BB6PayBills

Uses *BB5Bank* to pay a list of bills.

The example illustrates that a Context can play a role and be called as a subroutine in an outer context. (*BillPayer::payBills* calls the BankTransfer Context to pay one bill at a time). It also illustrates that a Collection may play a role.

3.5.1 Run the program:

World menu>>open…>>BB6: PayBills runs a test. The response is a window "Test OK".

3.5.2 Edit the program:

World menu>>open…>>BB1: IDE>>BB6PayBills. There are 3 projections: *Context*, *Data, Testing.*

3.5.3 Read the program:

BB6PayBills**:**

reader-friendly version
printer-friendly version

3.6 BB7 - the Dijkstra Algorithm

The Dijkstra Algorithm computes the shortest path to all nodes from a given origin node in a network of Nodes and Edges. This solution is based on James Coplien's solution written in Ruby, where the network has a modified and simplified Manhattan geometry. The solution illustrates the use of recursion rather than a loop.

This solution is particularly noteworthy because two Roles (*EastNeighbor* and *SouthNeighbor*) are played by instances of the same class and have different RoleMethods with the same name (*recomputeTentativeDistance*). There is no name conflict because Squeak/BabyIDE no longer uses RoleMethod injection.

3.6.1 Run the program:

World menu>>open…>>BB7: Dijkstra runs a test. The response is a window *"Test Dijkstra OK. More details in Transcript."*.

3.6.2 Edit the program:

World menu>>open…>>BB1: IDE>>BB7Dijkstra. There are 3 projections: *Context*, *Data, Testing.* Note that the Context as usual toggles between 2 views: *Class* and *Interaction*. The class view edits the *BB7CalculateShortestPathCTX class* that maps objects to roles, sets up the algorithm variables, and arranges for the recursive execution. The interaction view specifies the behavior of a node in the network.

Note that the code for traversal of the network with recursion is in the *Context>>class* view.

3.6.3 Read the program

BB7Dijkstra**:**

reader-friendly version
printer-friendly version

3.7 BB8 - MoveShape example

USE CASE.
Consider a Draw program such as a PowerPoint slide editor. A user can create and place different shapes such as ovals, polygons, lines, and other shapes. A Connector is a special line that runs from one shape to another. The line always connects the two shapes, even if the shapes move. This program implements the *shape:move:* operation.

There are two versions of this program, *BB8aMoveShape,* and *BB8bMoveShape.* The first version is for demo purposes, and its code is so simple that it can be explained in a regular talk by being limited to two shapes and a single connector.

The second version is general and illustrates that a role can be bound to a collection of shapes. The collection is an instance of *BB1OrderedCollection*, a collection that enumerates its members by

mapping them successively to a given role:
BB1OrderedCollection >>with: aRoleName do: aBlock
see
BB8bMoveShape>>Context>>Interaction
view>>LINECOLLECTION>>shapeHasMoved

3.7.1 Run the program:

World menu>>open…>>BB8a: MoveShape. The result is an inform-window: "Move shape test successful."

3.7.2 Edit the program:

BB8aMoveShape is limited to 2 shapes for demo purposes.
BB8bMoveShape has an unlimited number of connected shapes.

World menu>>open…>>BB1: IDE>>BB8aMoveShape. There are 3 projections: *Context*, *Data, Test.*

3.7.3 Read the program

BB8aMoveShape**:**

reader-friendly version
printer-friendly version

A version without the above restriction illustrates the use of Roles mapped to collection objects

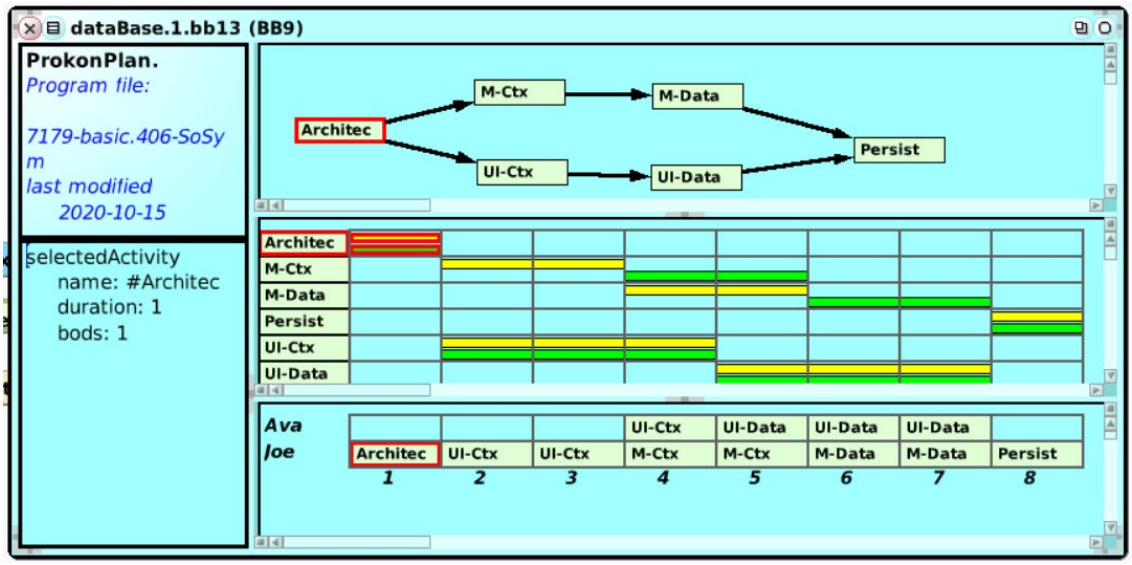
BB8bMoveShape**:**

reader-friendly version
printer-friendly version

3.8 BB9Planning: An Activity Planning Example (aka Prokon)

BB9Planning is an extended planning example that illustrates the combination of DCI and MVC.

Prokon is a simple planning tool; the screen dump below shows its user interface.



More details in the Personal Programming article (*https://doi.org/10.1007/s10270-019-00768-3*), *Appendix 1: ProkonPlan, an example*.

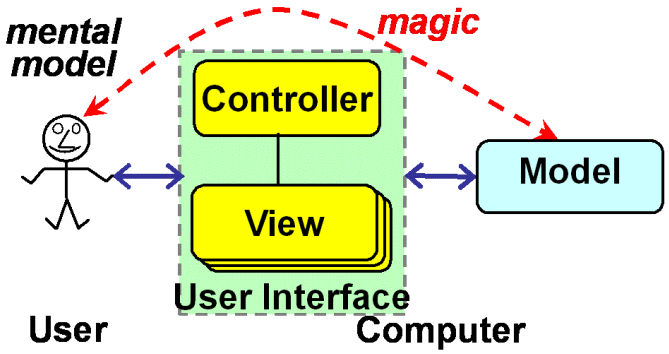
3.8.1 End-User Mental Model

An activity represents a task that needs to be done. It has a given duration; it cannot start before all its predecessor activities are completed, and it must end before its successor activities can start.

3.8.2 Program Architecture

The program architecture rests on two powerful principles for separation of concerns: MVC first and DCI second.

The Model-View-Controller (MVC) paradigm is illustrated below.



The Model is a representation of the user's information model. The *magic* in the above figure is that the user experiences the computer as an extension of their mind. The illusion is achieved by faithfully reflecting the user's mental model in the Model code. The user is in the driver's seat, and a View bridges the gap between their mind and the Model.

Different Views show different aspects of the Model in a way that can be readily intuited by the human. The human intuition makes it clear how to give input to the Model through a View. A Controller sets up one or more Views and coordinates them, e.g., making a selection show itself in all Views simultaneously.

The Model is again separated into two projections according to DCI; The *Data-Model* projection implements the static part of the user's mental model, while a *Context-Model* projection implements the operations on the Model. (Frontloading, backloading, etc.) The Model code is fairly generic and should be readable to a person with a rudimentary knowledge of the Squeak syntax.

The user interface (UI) bridges the gap between the human mind and the computerized Model. Usually, this is achieved by a Controller object and one object for each View. (Here 5). In Squeak, the classes of these objects build on its Morphic framework. The code is readable to a person who is familiar with this framework.

As an experiment, we have separated the static display classes from the algorithms for operating upon them, such as refreshing their contents. (*Data-UI* and *Context-UI,* respectively). This separation is an experiment, and it should be possible to find a simpler and more readable form.

3.8.3 Run the program:

World menu>>open…>>BB9: Planning

This command gives the choice of two plans, choose *database.1.bb13,* and get a plan similar to the one shown above.

3.8.4 Edit the program:

World menu>>open…>>BB1: IDE>>BB9Planning. 5 projections are reflecting the program's MVC/DCI architecture:
Controller-Data, Model-Context, Model-Data, View-Context, View-Data.

3.9 BB10: Frame

BB10Frame is an unsuccessful attempt at programming a version of the Pong game that Jim (Cope) Coplien has implemented in his **trygve** DCI programming language.

4 Squeak Reverse Engineering (SRE)

These tools let you study any object and object structure in the current image.

The program user manual is in

> *BabySRE, Squeak Reverse Engineering*
> *http://heim.ifi.uio.no/~trygver/themes/SRE/SRE-index.html*

4.1.1 Edit the Program

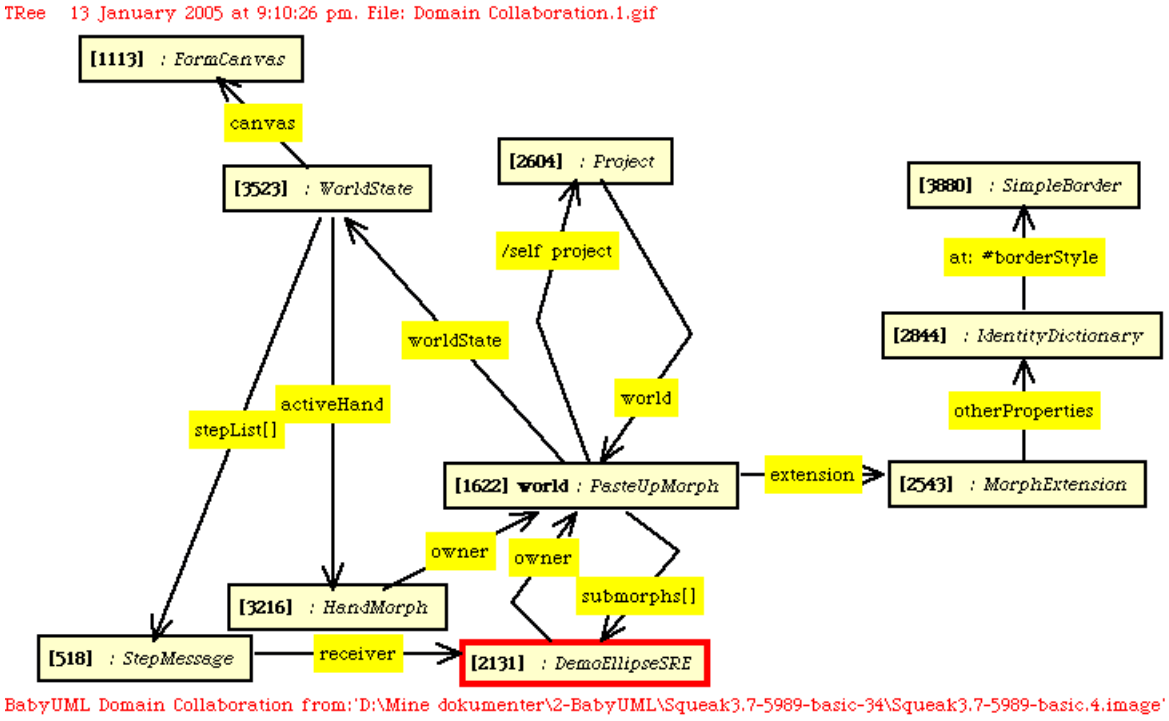
The SRE program code is in the following class categories:
'BabySRE-CClassDefiner', 'BabySRE-Connectors', 'BabySRE-Info', 'BabySRE-Inspector', 'BabySRE-Model', 'BabySRE-View'

4.2 The SRE Context browser

To do an example, execute
DemoEllipseSRE new openInWorld
and observe that an ellipse with cycling colors appear in the top left corner. Then do the following:

1. *ellipse halo>>Debug>>SRE Context browser*.>>new Context
   to open a context browser on the ellipse object
2. Place the object *[3625] : DemoEllipseSRE* in the diagram. (object identities will vary in different executions)
3. *[3625] : DemoEllipseSRE>>yellowmenu>>add link for variable... >>owner.*
   and place the owner object: *[999] World : PasteUpMorph*

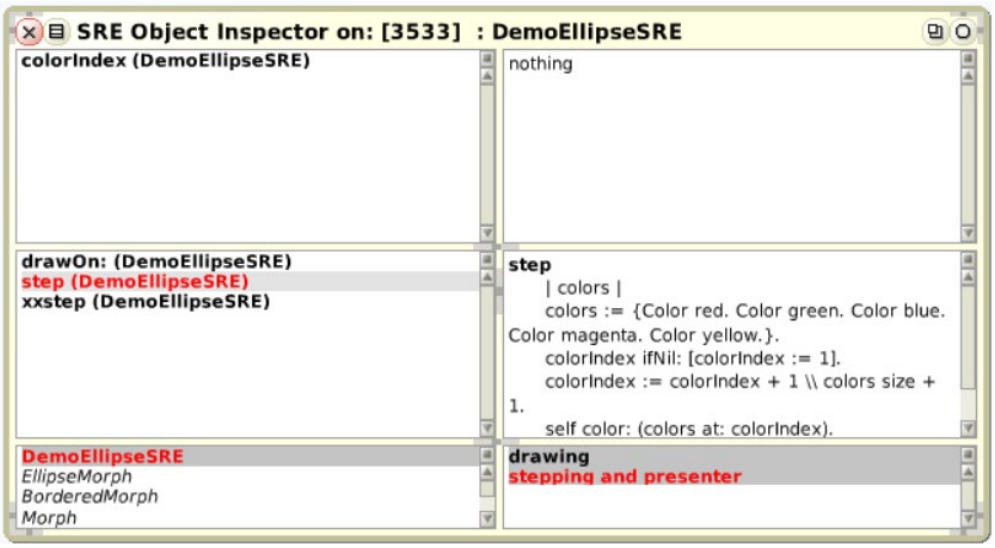
Then follow the user manual. As an example, the diagram below has *[2131] : DemoEllipseSRE* as its starting point. Note that the oops are different because the diagram was created in another execution.



4.3 SRE Object Inspector

The SRE Object inspector differs from the regular inspector in that inspects the whole object, including all the selectors (methods) it understands. Notice the multi-select lists for superclasses etc.

Open the halo around *demoEllipse>>debug>>SRE object inspector*.



5 Persistent workspace

This is a regular workspace that is represented as a *.ws*-file on disk in the directory of the image. Expand the third collapsed window from the

top-left on the desktop to see an example: "*Welcome (PersistentWorkspace)*"

5.1.1 Open a new persistent workspace

World menu>>open…>>Persistent workspace.

The name of the window is the name of the file with .ws added.