Object-Oriented Design: A BreakOut

Draft

Stéphane Ducasse
stephane.ducasse@inria.fr

©

February 14, 2014

Contents

Part I

BreakOut

To be fleshed, the code and some discussions are there. Basic story is missing.

1

A Ball and a Field

```
afterChapterOneSectionTwo

    BreakOutField newStandAlone openInWorld
```

This chapter is the first one of the breakout we will be designing together. The idea is that we will define step by step this breakout application. However, we will not do the best design first but simply build first what is possible then evaluate what we made and refactor the code if necessary. Indeed object-oriented design is about pros and cons and refinement. Moreover, an application usually has changing requirements that naturally forces the code to change. We also believe that building a prototype, i.e., a first version of a system is a good way to understand the problem and understand the trade-off that the design solution should address. You will see that during the chapter the responsibilities may change and the interactions between objects too.

First we start by building the ball and the game field.

1 Ball

@@

Class 1.1

```
EllipseMorph subclass: #BallMorph
   instanceVariableNames: 'speed direction '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

initialization category

Method 1.1

```
BallMorph>>initializeToStandAlone

   super initializeToStandAlone.
   self extent: 9@9.
   speed := 3.
   direction := 1@1.
   self startStepping.
```

The speed is the number of pixels that are added to the position of the receiver. The direction is a point representing how the speed is taken into account for each dimension. Here 1@1 says that the receiver will move each step 3 pixels in x and 3 pixels in y as the speed is 3.

In stepping

Method 1.2

```
BallMorph>>stepTime

   ^ 50
```

Method 1.3

```
BallMorph>>step
   "Change the ball position following its direction and speed"

   self position: self position + (speed * direction)
```

In a category named movement

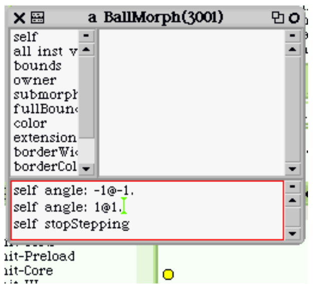
Method 1.4

```
BallMorph>>direction: aPoint

   direction := aPoint
```

Script 1.1 (*Take a grip on the ball.)*

```
BallMorph newStandAlone openInWorld inspect
```

With the inspector change the speed and the direction of the ball. Use stopStepping and startStepping if needed to stop the ball movement. Experiment to find how we can express that the ball is bouncing on a vertical or horizontal surface (see method**??** for the answer).



```
self angle: -1@-1.
self angle: 1@1.
self stopStepping
```

Figure 1.1:

2 BreakOutField

Class 1.2

```
BorderedMorph subclass: #BreakoutField
   instanceVariableNames: 'ball '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

In initialization

Method 1.5

```
BreakOutField >>initializeToStandAlone

    super initializeToStandAlone.
    self bounds: (0@0 corner: 600@350).
    self color: (Color red alpha: 0.5).
```

In

Method 1.6

```
BreakOutField>>ball: aBallMorph

    ball := aBallMorph.
    self addMorph: aBallMorph.
```

Now the ball belongs to the field and is detsroyed when the field is destroyed. When the field is moved (brown halo) the ball moves with it. When the ball is stop (ball stopStepping) and the field moved (black halo) the startStepping message is propagated to it. This is the field open in world that will open in wolrd its constituents, hence we do not have to openInworld explicitly the ball as shown in the method 1.7. Once the addMorph: method is executed the ball can access the morph it belongs to via the owner message.

Method 1.7

```
BreakOutField>>initializeToStandAlone

    super initializeToStandAlone.
    self bounds: (0@0 corner: 600@350).
    self color: (Color red alpha: 0.5).
    self ball: BallMorph newStandAlone.
```

Script 1.2 (*Opening the field.*)

```
BreakOutField newStandAlone openInWorld
```

3 Ball In The Field

In initialization

Method 1.8

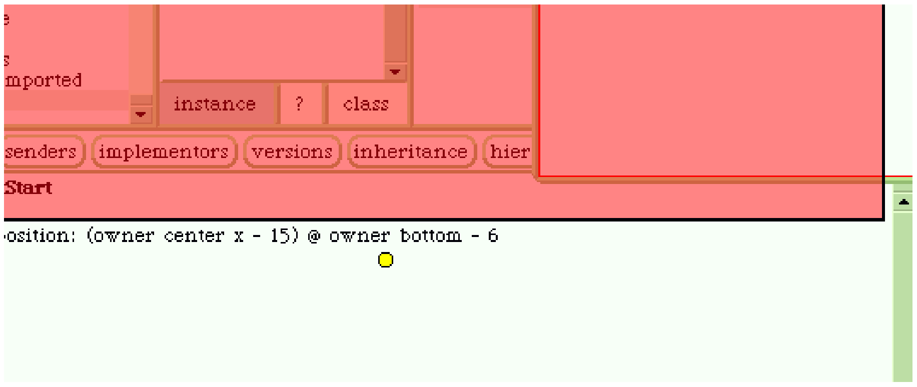
```
BallMorph>>positionAtStart

   self position: (owner center x - 15) @ owner bottom - 6
```

However we cannot call this method from within the BallMorph»initializeMethod because the addMorph: method is executed after the ball instance creation and in such a case the owner would be nil. So we let the responsibility to specify where the ball should be to the

Method 1.9

```
BreakOutField>>initializeToStandAlone

    super initializeToStandAlone.
    self bounds: (0@0 corner: 600@350).
    self color: (Color red alpha: 0.5).
    self ball: BallMorph newStandAlone.
    self positionAtStart.
```

Keeping the Ball Inside. To be able to force the ball to stay inside the field we have to determine its next position according to its speed and direction. This way we will be able to write some testing methods and ask the ball to bump so briefly presented by the method sketch 1.10.



```
osition: (owner center x - 15) @ owner bottom - 6
```

Figure 1.2:

Method 1.10

```
BallMorph>>step
   ...
   ownerBounds := owner bounds.
   (self isNextPositionLeft: ownerBounds)
      ifTrue:
         [self bounceLeftRight.
         self moveToNextPosition .
         ^self]
      ifFalse:
         [(self isNextPositionAbove: ownerBounds)
            ifTrue:
               [self bouncingTopBottom.
               self moveToNextPosition .
               ^self]
         ....
```

`nextPosition .`

From this sketch we see that we will have to compute the next position that the ball will occupy.

Method 1.11

```
BallMorph>>nextPosition
   "Return the next position  that the receiver would occupy when he does
   not bump into something"

   ^ self center + (speed * direction)
```

Then we need some methods to check whether the ball will exist the field.

Method 1.12

```
BallMorph>>isNextPositionAbove: aRectangle
   "Return true whether the next position is above the rectangle"

   ^ self nextPosition  y - 4 < aRectangle top
```

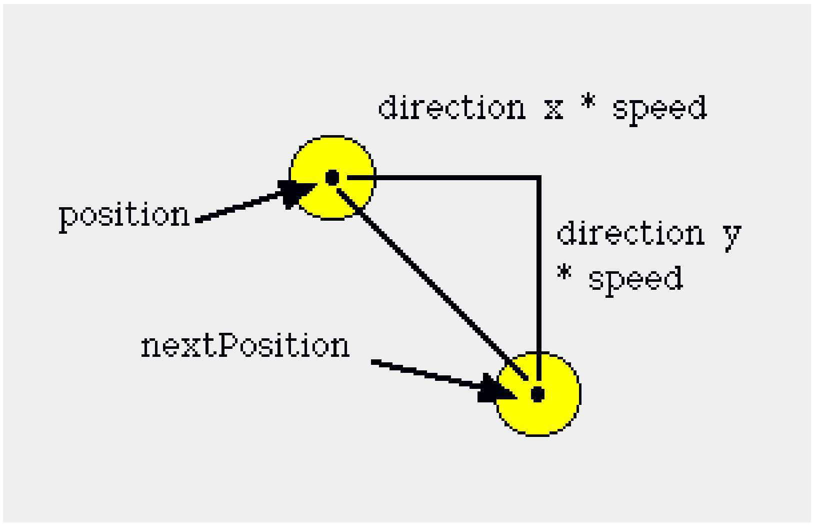


Figure 1.3: Computing the next position of a ball.

Method 1.13

```
BallMorph>>isNextPositionBelow: aRectangle
   "Return true whether the next position is below the rectangle"

   ^ self nextPosition  y + 4 >= aRectangle bottom
```

Method 1.14

```
BallMorph>>isNextPositionLeft: aRectangle
   "Return true whether the next position is to the left of the rectangle"

   ^ self nextPosition  x - 4 < aRectangle left
```

Method 1.15

```
BallMorph>>isNextPositionRight: aRectangle
   "Return true whether the next position is to the right of the rectangle"

   ^ self nextPosition  x + 4 > aRectangle right
```

Method 1.16

```
BallMorph>>bounceLeftRight
   "Make the ball bouncing on an vertical surface by changing its direction"

   direction :=  direction  * (-1@1).
   self moveToNextPosition.
```

Method 1.17

```
BallMorph>>bounceTopBottom
   "Make the ball bouncing on an horizontal surface by changing its direction"

   direction := direction * (1@-1).
   self moveToNextPosition.
```

Method 1.18

```
BallMorph>>nextPosition
   "Move the receiver to the next position according to its direction and speed"

   self center: self nextPosition
```

Method 1.19

```
BallMorph>>step

   | ownerBounds|
   ownerBounds := owner bounds.
   (self isNextPositionLeft: ownerBounds)
      ifTrue:
         [self bounceLeftRight.
         ^self]
      ifFalse:
         [(self isNextPositionRight:  ownerBounds)
            ifTrue:
               [self bounceLeftRight.
               ^self]
            ifFalse:
               [(self isNextPositionBelow: ownerBounds)
                  ifTrue:
                     [self bounceTopBottom.
                     ^self]
                  ifFalse:
                     [(self isNextPositionAbove: ownerBounds)
                        ifTrue:
                           [self bounceTopBottom.
                           ^self]
                        ifFalse:
                           [self moveToNextPosition .
                           ^ self]]]]
```

This method can be shortened a bit as the same action has to be done for the left or right side. For the bottom side we will introduce the fact that the game is over in a following chapter.

Method 1.20

```
BallMorph>>isNextPositionVerticallyInside: aRectangle
"Return true whether the next position is to the right of the rectangle"

| nextPosition |
nextPosition := self nextPosition x.
  ^  (nextPosition + 4 > aRectangle right) or: [nextPosition - 4 < aRectangle left]
```

Method 1.21

```
step

   | ownerBounds|
   ownerBounds := owner bounds.
   (self isNextPositionVerticallyInside: ownerBounds)
      ifTrue:
         [self bounceLeftRight.
         ^self]
      ifFalse:
         [(self isNextPositionBelow: ownerBounds)
            ifTrue:
               [self bounceTopBottom.
               ^self]
            ifFalse:
                [(self isNextPositionAbove: ownerBounds)
                     ifTrue:
                        [self bounceTopBottom.
                        ^self]
                     ifFalse:
                        [self moveToNextPosition.
                        ^ self]]]
```

Notice that this solution let the ball decide its own ,...and its coherent with oo however we will see soon that we want to minimize the number of collaborator so we will let the field controlling the ball.

4 Optimizing a Bit and Design Discussions

In a first reading you can skip this section that discusses several points of the implementation we proposed. The solution we propose is one solution and others are possible which can be better. For example, the methods `isNextPosition...` always call `nextPosition` which is then multiply times computed. To avoid such a situation we can call on in the method `step` the method `linearPosition` then pass an extra argument to the methods `isNextPosition` as in the method 1.23.

Method 1.22

```
BallMorph>>step
...
ownerBounds := owner bounds.
  nextPosition := self nextPosition .
(self isNextPosition: nextPosition left: ownerBounds)
ifTrue:
[self bounceLeftRight.
^self]
ifFalse:
[ (self isNextPosition: nextPosition above: ownerBounds)
ifTrue:
[self bounceTopBottom.
^self]
...
```

Method 1.23

```
BallMorph>>isNextPosition: aPoint right: aRectangle
    "Return true whether the next position is to the right of the rectangle"

    ^ aPoint x + 4 > aRectangle right
```

Caching the position. Another solution is to change the method `nextPosition` so that it caches the computation. For this we would have to add an instance variable to class called for example, `cachednextPosition` , and changed the method `nextPosition` and the method `nextPosition` to invalidate the cache. This example illustrates that for a cache two questions are important when to store the information and when to invalidate it. Note also that the fact we use the method `nextPosition` we can add the cache without changing all the methods that use it.

Method 1.24

```
BallMorph>>nextPosition

    cachedNextPosition  isNil
        ifTrue: [ cachedNextPosition  := self center + (speed * direction)].
   ^ cachedNextPosition
```

Method 1.25

```
BallMorph>>moveToNextPosition

   self center: self nextPosition .
   cachedNextPosition  := nil
```

Note that passing an extra parameter is a good solution when the method calling them is defined on the class `BallMorph` because the class can control the consistency between the current position and the next one. When the caller is another class as we will introduce in a future chapter, the class `BallMorph` loses a bit this control. Hence it is possible to write silly code as follow where a wrong point is passed as argument representing the next linear position.

Method 1.26

```
BreakOutField>>moveBall

   | nextPosition  b |
   nextPosition  := ball nextPosition  + 100@100.
   (self isNextPosition: nextPosition  right: aRectangle: owner bounds)
ifTrue: ...
```

2

A Batter In the Field

1 Batter

Class 2.1

```
RectangleMorph subclass: #BatterMorph
   instanceVariableNames: 'speed '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

Method 2.1

```
BatterMorph>>initializeToStandAlone

   super initializeToStandAlone.
   self color: (Color blue alpha: 0.6).
   self borderColor: (Color blue alpha: 0.8).
   self useRoundedCorners.
   speed := 15.
```

2 Batter in Field

Redefine

Class 2.2

```
BorderedMorph subclass: #BreakoutField
   instanceVariableNames: 'batter ball '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

Method 2.2

```
BreakOutField>>batter: aBatterMorph
   "Add the batter as contained by the receiver"

   batter := aBatterMorph.
   self addMorph: aBatterMorph
```

Method 2.3

```
BatterMorph>>positionAtStart

self center: (owner center x @ owner bottomRight y - self height - 5)
```

Method 2.4

```
BreakOutField>>initializeToStandAlone

    super initializeToStandAlone.
    self bounds: (0@0 corner: 600@350).
    self color: (Color red alpha: 0.5).
    self ball: BallMorph newStandAlone.
     ball positionAtStart.
    self batter: BatterMorph newStandAlone.
    batter positionAtStart
```

Controlling the Batter.

Method 2.5

```
BreakOutField>>handlesKeyboard: evt

   ^ true

BreakOutField>>handlesMouseOver: evt

   ^ true

BreakOutField>>mouseEnter: evt

   evt hand keyboardFocus: self

BreakOutField>>mouseLeave: evt

   evt hand releaseKeyboardFocus: self

BreakOutField>>keyDown: evt

   | char |
   char := evt keyCharacter.
   char = $n
      ifTrue: [Transcript show: 'Left'].
   char = $m
      ifTrue: [Transcript show: 'Right'].
```

Now when you press the key n and m, you should get in the Transcript the string 'Left' or 'Right' printed.

Moving the Batter. Now the batter define methods to move left and right. We named them `moveLeft` and `moveRight`. Redefine the method `keyDown:` to call these methods (2.6) and propose a definition for the methods `moveLeft` and `moveRight`.

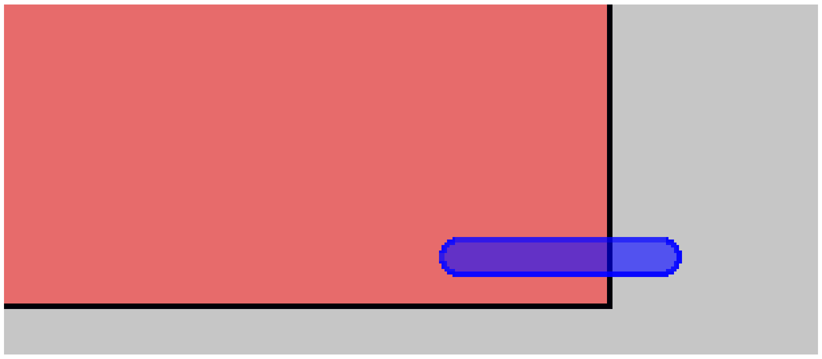


Figure 2.1: The method `BatterMorph»moveRight` method 2.7 effectively moves the batter but does not check whether it stays in the field

Method 2.6

```
BreakOutField>>keyDown: evt

   | char |
   char := evt keyCharacter.
   char = $n
      ifTrue: [batter moveLeft].
   char = $m
      ifTrue: [batter moveRight].
```

The method 2.7 proposes a first solution. It just changes the position of the batter by adding the speed converted as a point to the current position.

Method 2.7

```
BatterMorph>>moveRight

    self position: self position + (speed @ 0)
```

The method `BatterMorph»moveRight` method 2.7 effectively moves the batter but does not check whether it stays in the field as shown by the figure2. Experiment and propose a solution.

Towards a Good `moveRight` method. A solution is to check before moving the batter if it will stay in the field bounds as shown in the method the method 2.8. Implement it to evaluate if you like it.

Method 2.8

```
BatterMorph>>moveRight

   (owner bounds containsPoint: self bounds topRight + speed)
      ifTrue: [self position: self position + (speed @ 0) ]
```

In fact the solution proposed in the method 2.8 has still a problem. When the batter has a significant speed such as 10 or 15 pixels, it may happen that we forbid it to move else it would exist the field but doing so we let some space where the ball could pass between the wall and batter which is really bad for the gameplay. This situation is shown in figure 2.2. Propose a solution to fix this problem.

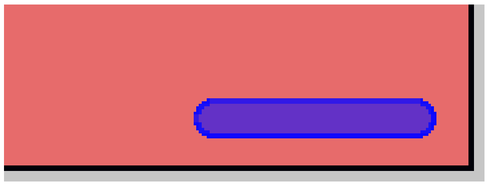


Figure 2.2: The second version of the method `BatterMorph»moveRight` method 2.8 effectively keeps the batter in the field but sometimes leading to gameplay problem. Here the ball can still pass between the batter and the wall.

A Good `moveRight` method. Solving the problem enonced before is based on the following idea. When the batter may end up outside of the field, it should not move by its speed but by the difference between the wall and the batter extremity.

Method 2.9

```
BatterMorph>>moveRight

   | topRight owBounds xToMoveRight |
   topRight := self bounds topRight.
   owBounds := self owner bounds.
   xToMoveRight := (owBounds containsPoint: topRight +  speed)
                                    ifTrue: [speed ].
                                    ifFalse: [owBounds right - topRight x].
     self position: self position + (xToMoveRight @ 0)
```

We can reexpress the method 2.9 by seeing that we want to move the batter by the miminum between the speed and the difference between the batter extremity and the wall. The methods method 2.10 shows the final version where we split the method to have better readibility. Apply the same idea for `moveLeft` before looking at the solution we propose method 2.11).

Method 2.10

```
BatterMorph>>moveRight

self position: self position + (self xToMoveRight @ 0)

BatterMorph>>xToMoveRight
"Return the x so that the receiver can move to the right without existing the field"

^ speed  min: (self owner bounds right - self bounds topRight x)
```

Method 2.11

```
moveLeft

self position: self position - (self xToMoveLeft @0)

xToMoveLeft

^  speed  min: (self bounds topLeft x - owner bounds left)
```

3 Batter and Ball Interaction

Now we want the ball to bounce on the batter. This brings an interesting issue: where do we describe the game logic? Right now the ball is checking that it is not going outside the field. But now we want to check whether the ball bumps into the batter and in the future we will have brick collision. We could continue this way but the ball would have to be in collaboration with the field, the batter and the bricks. This is not per se a problem but the field will also be in a such a situation due to the fact that it contains the other objects.

What you see is that there is a tradeoff between having objects with too much collaborators and objects having too much control of the others. There is no definite answer about the distribution of the logic game even if object-oriented programming favors distribution of the responsibilities.

We decide to move the behavior of the method `BallMorph»step` in the class `BreakOutField`. To do so we define the method `BreakOutField»moveBall`

Method 2.12

```
BreakOutField>>moveBall
   | ownerBounds|
   ownerBounds := owner bounds.
   (ball isNextPositionVerticallyInside: ownerBounds)
      ifTrue:
         [ball bounceLeftRight.
         ^self]
      ifFalse:
         [(ball isNextPositionBelow: ownerBounds)
               ifTrue:
                  [ball bounceTopBottom.
                  ^self]
               ifFalse:
                  [(ball isNextPositionAbove: ownerBounds)
                     ifTrue:
                        [ball bounceTopBottom.
                        ^self]
                     ifFalse:
                        [ball moveToNextPosition.
                        ^ self]]]
```

And change the method step to call it.

Method 2.13

```
BallMorph>>step

   owner moveBall
```

The code of the method `moveBall` is not really good. So we decide to cut it into pieces and to change the methods calculating if the ball will exit the field such as `BallMorph»isNextPositionVerticallyInside: aRectangle` to to take the next position as extra argument as shown by the method method 2.14. We let you transform the other related methods in the class BallMorph.

Method 2.14

```
BallMorph>>isNextPosition: aPoint verticallyInside: aRectangle
   "Return true whether the next position is to the right of the rectangle"

   | xOfNextPosition |
   xOfNextPosition := aPoint x.
   ^  (xOfNextPosition + 4 > aRectangle right) or: [xOfNextPosition - 4 < aRectangle left]
```

The method `moveBall` now looks as the definition method 2.15. The idea is that when the ball will bounce on the wall the method `ifBallNotBouncedOnWall:` takes care of making the ball bounces and forward. To avoid to move the ball twice, it returns false indicating that the ball does not have to be further controlled for this step. When the ball will not touch a wall or lost, the method `moveBall` checks whether the ball will bump into the batter. In such a case the ball bounces on it. Note that the collision between the batter and the ball is rudimentary and may lead to strange situations. We will you imagine other collision detection approaches taking into account the sides of the batter and the direction of the ball.

Method 2.15

```
BreakOutField>>moveBall

   | nextPosition |
   nextPosition := ball nextPosition.
   (self ifBallNotBouncedOnWall: nextPosition)
      ifTrue:
         [(batter containsPoint: nextPosition)
            ifTrue:
               [ball bounceTopBottom].
         ball moveToNextPosition]
```

Method 2.16

```
BreakOutField>>ifBallNotBouncedOnWall: aPoint
   "Make the ball bouncing on walls and dying if possible. Return true if the ball next position
   not  bouncing on a wall false otherwise"

   (ball isNextPosition: aPoint verticallyInside: bounds)
      ifTrue:
         [ball bounceLeftRight.
         ^false]
      ifFalse:
         [(aPoint y  > (batter bottom))
            ifTrue: [self lostABall.
                     ^false.]
            ifFalse:
               [(ball isNextPosition: aPoint above: bounds)
                  ifTrue: [ball bounceTopBottom.
                           ^false]
                  ifFalse: [^ true]]]
```

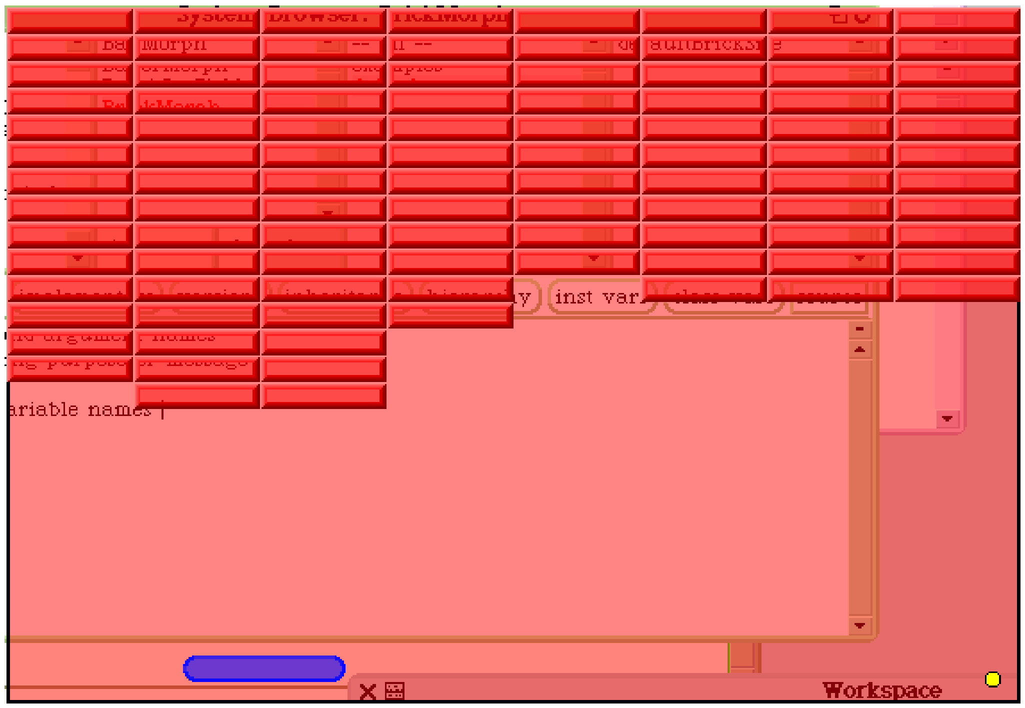
Method 2.17

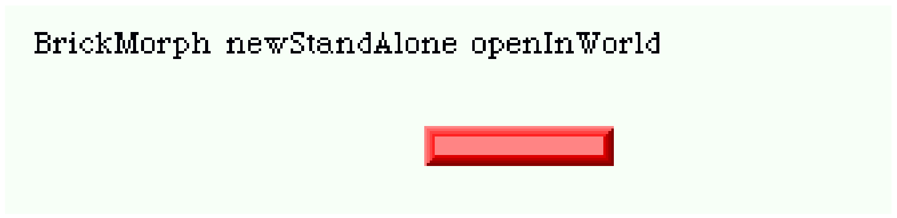
```
BreakOutField>>lostABall

Transcript show:  'You die'
```

By the way you will see that this is boring to stop the game when you lose the ball. You could for example say to the ball to bounce.

3

Bricks





```
BrickMorph newStandAlone openInWorld
```

Figure 3.1:

1 Adding Bricks

Class 3.1

```
BorderedMorph subclass: #BrickMorph
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

Method 3.1

```
BrickMorph>>initializeToStandAlone
   "self newStandAlone openInWorld"

   super initializeToStandAlone.
   self color: (Color red alpha: 0.5).
   self borderColor: Color orange.
   self borderStyle: (BorderStyle complexRaised width: 4).
   self extent: 71@15
```

Script 3.1 (*Creating a brick)*

```
BrickMorph newStandAlone openInWorld
```

2 Bricks in the BreakOutField

Class 3.2

```
BorderedMorph subclass: #BreakOutField
   instanceVariableNames: 'batter ball bricks '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

Method 3.2

```
BreakOutField>>initializeToStandAlone

   super initializeToStandAlone.
   self bounds: (0@0 corner: 600@350).
   self color: (Color red alpha: 0.5).
   self ball: BallMorph newStandAlone.
   ball positionAtStart.
   self batter: BatterMorph newStandAlone.
   batter positionAtStart.
   self initializeBricks
```

The easiest way to have brick is to lay them down in line. In chapter **??** we will show you how to place bricks in a more flexible way. Right now we just want to have the possibility to create lines of bricks. The method `initializeBricks` call a certain number of times the method `brickLine: anInteger` where anInteger represents the number of the line (0 starting on the top of the field).

Method 3.3

```
BreakOutField>>initializeBricks

   bricks := OrderedCollection new.
   0 to: 9 do: [:y | self brickLine: y].
```

If we want to have 8 columns of bricks we create a brick, then computes its position which is its size multiplied by the row and column where it should be, and finally add the brick to the field using the `addBrick:` method that we have still to define.

Method 3.4

```
BreakOutField>>brickLine: y
   "Add a line of brick at the y row"
   | b |
   0 to: 7 do:
      [:i | b := BrickMorph newStandAlone.
         b position: (i @ y) * (71@15).
         self addBrick: b].
```

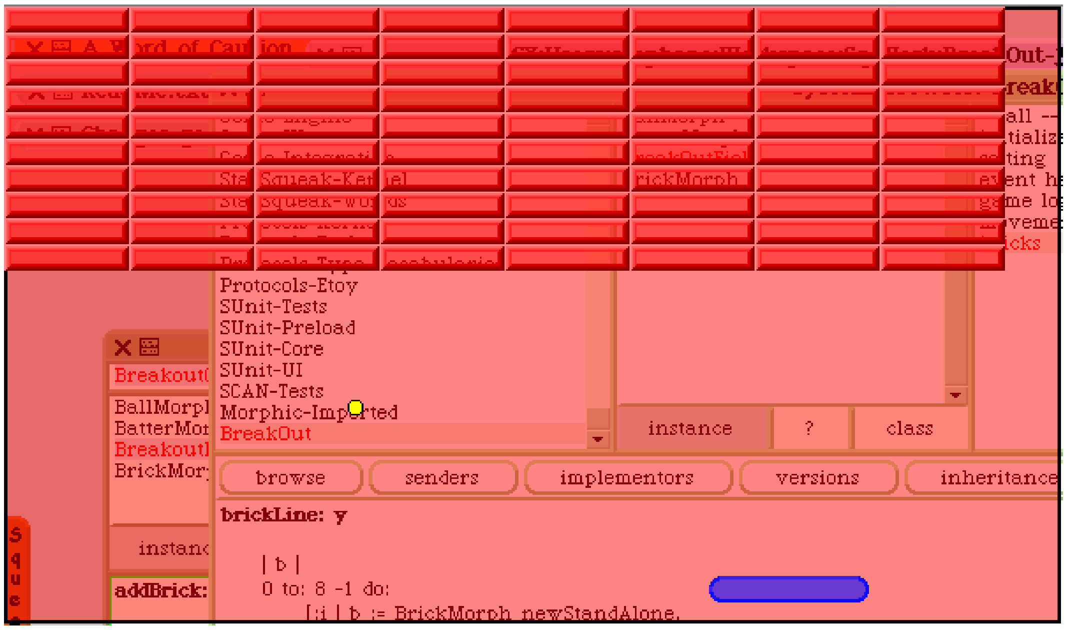
Finally we have to define what means to add a brick in the field. The method `addBrick:` adds the brick as owned by the field using the addMorph message. After this message is executed the brick is contains in the field. Then it adds the bricks to the collection of brick the field have. Note that we could have only used the fact that the field is a morph and as such knows all the morph it contains, but this would have forced us to filter among the morphs to only get the bricks and not the ball or the batter.

Method 3.5

```
BreakOutField>>addBrick: aBrickMorph

   self addMorph: aBrickMorph.
   bricks add: aBrickMorph
```

3 Avoiding Hardcoded Information

As you see the BreakOutField size is not adjusted to the number of bricks we draw per line. to fix this situation we will just change the size of the field to be related to the number of bricks. A possible solution would be to change the initialize to look as the method 3.6.



```
brickLine: y

    | b |
    0 to: 8 -1 do:
        [:i | b := BrickMorph newStandAlone.
```

Figure 3.2:

Method 3.6

```
BreakOutField>>initializeToStandAlone
   "self newStandAlone openInWorld"

   super initializeToStandAlone.
   self bounds: (0@0 corner: (71@15) * (8@9)).
   self color: (Color red alpha: 0.5).
   self initializeBall.
   self initializeBatter.
   self setBallAndBatterPosition.
   self initializeBricks
```

Look at this method and the code we wrote until now and try to find what could be a problem. The problem is that we are hardcoding everywhere in the method body information such as the number of rwo and lines that the fields have, the size of the bricks...This is definitively not a good practice because if we decide to change one of these parameters, we will have to look everywhere and remember why we have a certain number in a given place and we may have hard time to fix possible bugs that will be popping up. On this simple problem this may not look as a problem but on a bigger system this will.

Making Explicit the Field Size

We propose you to fix these problems by defining a couple of methods that will return the different value we use for building the game. Once you will get done we suggest you to change the value to see how easy it is to change these parameters. First let us make explicit the size of the field in terms of brick numbers. The method playFieldSize will return a point representing the number of columns and lines.

Method 3.7

```
BreakOutField>>playFieldSize
   "Return the size of the complete game area expressed in terms of brick rows and columns"
   ^ 8@26
```

Method 3.8

```
BreakOutField>>initializeToStandAlone

   super initializeToStandAlone.
   self bounds: (0@0 corner: (71@15) * self playFieldSize).
   self color: (Color red alpha: 0.5).
   self ball: BallMorph newStandAlone.
   ball positionAtStart.
   self batter: BatterMorph newStandAlone.
   batter positionAtStart.
   self initializeBricks
```

Method 3.9

```
BreakOutField>>brickLine: y

   | b |
   0 to: self playFieldSize x -1 do:
      [:i |
      b := BrickMorph newStandAlone.
      b position: (i @ y) * (71@15).
      self addBrick: b].
```

Note that in this method we do not go over the complete field area but let a space of 12 lines of bricks between the last row and the bottom of the game.

Method 3.10

```
BreakOutField>>initializeBricks

   bricks := OrderedCollection new.
   0 to: self playFieldSize y - 12 do: [:y | self brickLine: y].
```

Hardcoding 12 is not really so change the code.

Making Explicit the Brick Size

The second hardcoded value that we want to make explicit is the size of a brick. In such a case changing it would have more impact because two classes are using it: the class `BrickMorph` and the class `BreakOutField`.

For the brick size, the story is a bit more difficult because the `BreakOut` uses it before any brick is created so we cannot send message to a brick and ask it size. One solution is to put the expressions `self bounds: (0@0 corner: aBrick size * self playFieldSize).` of the method `BreakOutField»initializeToStandAlone` after the brick initialization, to take any bricks contained in the brick collection, and to ask its size. We favor another solution because it will give us the opportunity to show how class themselves can be used. In fact in a similar manner that we send message to objects, we can send messages to classes.

So the first step is to change the method `BrickMorph»initializeToStandAlone` to invoke a new method called `brickSize`.

Method 3.11

```
BrickMorph>>initializeToStandAlone
   "self newStandAlone openInWorld"

   super initializeToStandAlone.
   self color: (Color red alpha: 0.5).
   self borderColor: Color orange.
   self borderStyle: (BorderStyle complexRaised width: 4).
   self extent: self brickSize
```

We could define the method `brickSize` as shown in method 3.12 but this would not solve our problem. Because for the method `BreakOutField»initializeToStandAlone` we need to know the brick size before any brick is created.

Method 3.12

```
BrickMorph>>brickSize

   ^ 71@15
```

The solution is to first define a method called for example `defaultBrickSize` on the *class* `BrickMorph` *itself* as explained 4. In the method definition we will use the convention that we append `' class'` to the class name to help you understanding that the method is defined on the class itself as shown in the method method 3.13.

Method 3.13

```
BrickMorph class>>defaultBrickSize

   ^ 71@15
```

Then we define an instance method with the name `brickSize` but on the instance side this time. This method just invokes the method `defaultBrickSize` by first getting the class of the receiver using the message `class` which returns the class of any object in the system. Then sending to the class the message `defaultBrickSize` and returning the returned value.

Method 3.14

```
BrickMorph>>brickSize

   ^ self class defaultBrickSize
```

Designer Hints

We could have written the method `brickSize` as `B̂rickMorph defaultBrickSize`, but this is really bad practice to hardocde class names into method since we may change the class name later and be forced to recompile all the methods having references to the class.

Designer Hints

Now we are ready to invoke the method `brickSize` from the methods of the class `BreakOutField`

Method 3.15

```
BreakOutField>>initializeToStandAlone

   super initializeToStandAlone.
   self bounds: (0@0 corner: BrickMorph defaultBrickSize * self playFieldSize).
   self color: (Color red alpha: 0.5).
   self ball: BallMorph newStandAlone.
   ball positionAtStart.
   self batter: BatterMorph newStandAlone.
   batter positionAtStart.
   self initializeBricks.
```

Method 3.16

```
BreakOutField>>brickLine: y

   | b |
   0 to: self playFieldSize x -1 do:
      [:i | b := BrickMorph newStandAlone.
         b position: (i @ y) * b brickSize.
         self addBrick: b].
```

Now we are done, we should be able to change the size of the bricks and of the play field only on single places (the methods `playFieldSize` and `defaultBrickSize` and automatically the game will be coherent.

4 About the Instance/Class Buttons

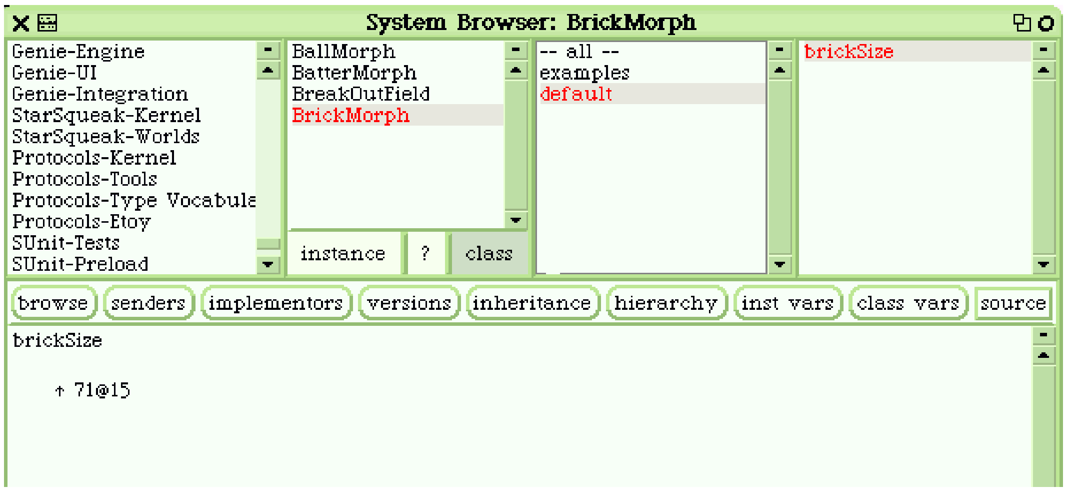
We open a parenthesis to explain an interesting but often not well-understood aspect of Smalltalk. In a similar way that we send message to objects we can send messages to classes. This is what we have been doing since the really first script of this book. Indeed to get a new turtle we send the message `new` to the class `Turtle`. In a similar way that you learn how to define methods for messages send to objects, you can define methods for messages send to classes.

We can define methods and send them messages in exactly the same way that with any other objects. To help you defining class methods (methods defined on classes) the browser has two buttons, instance and class. When you want to edit or see methods that will be executed when a message is sent to an instance you have to select the instance button. For example, the message `go: 100`, `color: Color yellow`, `turn: 90` that you sent to a turtle are methods that are executed on the instance.

Script 3.2 (*Example of messages sent to instances)*

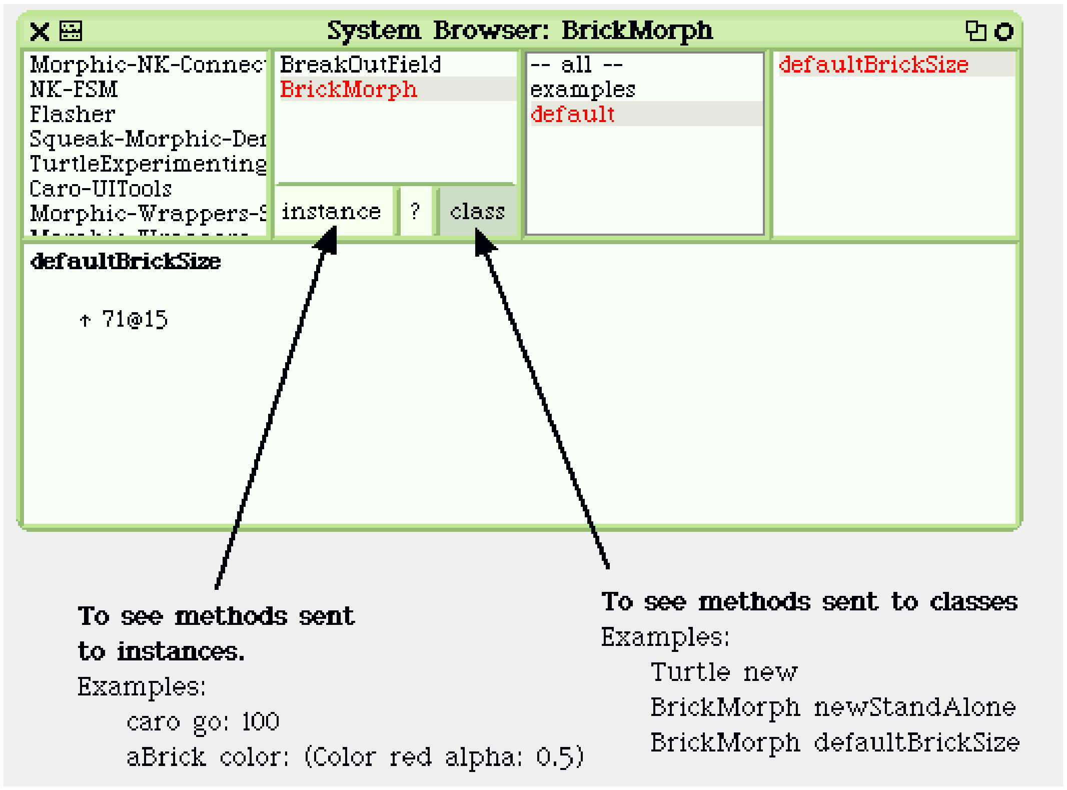
```
caro := Turtle new.
caro go: 100.
caro color.
caro class
```

The message `class` returns the class of the instance as shown in the script 3.3. Try to send this message to the results returned by various messages to see the different classes involved in Squeak.



```
brickSize

    ↑ 71@15
```

Figure 3.3: Browsers buttons



```
defaultBrickSize

    ↑ 71@15
```

To see methods sent to instances.
Examples:
caro go: 100
aBrick color: (Color red alpha: 0.5)

To see methods sent to classes
Examples:
Turtle new
BrickMorph newStandAlone
BrickMorph defaultBrickSize

Figure 3.4: Browsers buttons

Script 3.3 (*Accessing the class of an instance)*

```
100 class
    -Value-> SmallInteger


"Turtle new creates an instance of the class Turtle
so asking its class return Turtle"
| caro |
caro := Turtle new.
caro class
    -Value-> Turtle


"Color yellow return an instance of the class Color
so querying its class returns the class Color"
Color yellow class
    -Value-> Color
```

When you want to see or define methods that will be executed when messages are sent to a *class itself*, you have to select the *class* button.

Script 3.4 (*Example of messages sent to classes)*

```
Color yellow.
Turtle new
BrickMorph newStandAlone
BrickMorph defaultBrickSize
```

5 Destroying Bricks

Finally we arrive at the essence of the game: the destruction of bricks.

Method 3.17

```
BreakOutField>>moveBall

   | nextPosition b |
   nextPosition := ball nextPosition.
   (self ifBallNotBouncedOnWall: nextPosition)
      ifTrue:
         [(batter containsPoint: nextPosition)
            ifTrue: [ball bounceTopBottom.
                        ^ self].
         b := (self brickContainingOrNil: nextPosition).
         b ifNotNil:
               [ball bounceTopBottom.
               b actionWhenBumped].
         ball moveToNextPosition]
```

First when the ball bounces on the batter we simply do not want to check anything else, so we add an explicit return ŝelf. Then we check if the next position of the ball is located into one the bricks. The method `brickContainingOrNil:` returns nil when the ball will not touch a brick. When the ball will be located inside a brick, the ball is asked to bounce then we ask the brick to react to the fact that the ball bounced on it. Note that this way of doing thing is only one solution because we could have ask the brick to be responsible of the ball bumping on it.

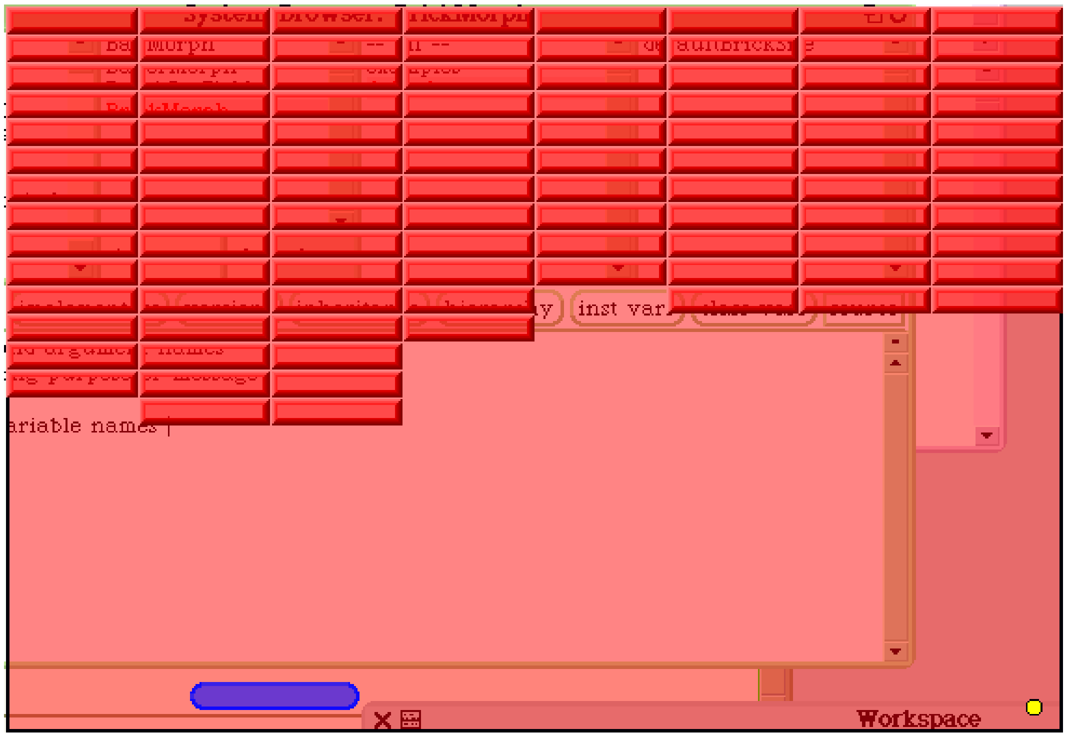


Figure 3.5: Destroying Bricks

Method 3.18

```
BreakOutField>>brickContainingOrNil: aPoint
   "Return the brick containing the point aPoint or nil if none of the bricks contains the point"

   ^ bricks detect: [:each | each containsPoint: aPoint] ifNone: [nil]
```

Method 3.19

```
BreakOutField>>removeBrick: aBrickMorph
   "Remove the brick morph from the collections of bricks"

   ^ bricks remove: aBrickMorph
```

The minimal behavior that the brick should do when touched by a ball is to ask the field that owns it using the message `owner` to remove it from the lists of morph it contains using the message `removeBrick:`. Then the brick destroyed itself using the message `delete` with is part of the basic behavior morphs have and is the opposite of `openInworld`.

Method 3.20

```
BrickMorph>>actionWhenBumped

   self owner removeBrick: self.
   self delete.
```

Method 3.21

```
BallMorph>>positionAtStart

   self position: (owner center x - 15) @ (owner bottom -40)
```

6 Lessons Learnt

- Factor constants
- Say things only once
- Class methods

4 Refining Gameplay as a Pretext to Talk about Design

Until now the game play is a bit draft. In this chapter we add visual and audio feedback, and a new kind of bricks. This will also be the occasion to refine the design and change collaborations between classes. So while this chapter seems at first not important, the changes introduced and discussions are important. We will also look again at the way inheritance helps to define new classes by extending existing ones. The chapter is a good complement of the chapter **??**.

1 Adding Visual and Audio Feedback

To add an audio and visual feedback we redefine the method `actionWhenBumped` as shown in the method 4.1. The method `flash` is defined on the class `Morph` and changes quickly the color of a morph to make it flashing. Then we create a sound and play it. Squeak offers a small library of predefined sounds. Look in the class `SampledSound` class. You can find the names of the sounds available by executing the following expression `SampledSound soundNames`.

Method 4.1

```
(temporary)
BrickMorph>>actionWhenBumped

   self owner removeBrick: self.
   self flash.
   (SampledSound soundNamed: 'scratch') play.
   self delete.
```

To add a feedback when the ball hits the batter, we modify the method `moveBall` (see method 4.2. This solution is not really satisfactory because

Method 4.2

```
(temporary)
BreakOutField>>moveBall

   | nextPosition b |
   nextPosition := ball nextPosition.
   (self ifBallNotBouncedOnWall: nextPosition)
      ifTrue:
         [(batter containsPoint: nextPosition)
            ifTrue:
               [ball bounceTopBottom.
                  (SampledSound soundNamed: 'motor') play.
               ^ self].
         b := (self brickContainingOrNil: nextPosition).
         b ifNotNil:
            [ball bounceTopBottom.
            b actionWhenBumped].
         ball moveToNextPosition]
```

Giving the field the responsibility to make a noise when the ball touches the batter is not good at all. Indeed, it should be the responsibility of the batter to produce a visual or audio effect when it is touched. We easily could imagine that we would like to have different kinds of batter producing different bouncing effect and noise. The field should not be worried about which kinds of batter it contains. It should let the batter decide.

Similarly different bricks may act in different way. Here the field again has the responsibility to make the ball bouncing. However, new kinds of bricks may simply destroy or capture the ball. That is why making the ball bouncing should be a responsibility of the brick itself. This way the field does not have to pay attention to which kind of brick the ball will bounce but just notify the brick that a ball is touching it. A better designed solution to our breakout problem is shown by the method method 4.3

Method 4.3

```
(final)
BreakOutField>>moveBall

   |  b nextPosition |
   nextPosition := ball nextPosition.
   (self performMoveNotInField: nextPosition)
      ifTrue:
            [(batter containsPoint: nextPosition)
                  ifTrue:
                     [batter touchedBy: ball.
                     ^ self].
            b := (self brickContaining: nextPosition).
            b ifNotNil:
                  [ b actionWhenBumpedBy: ball].
            ball moveToNextPosition]
```

In general, you should define in the class the behavior related to the object and not let other object having this responsibility. This is a difficult exercise because you can have a solution that is not satisfactory form its design point of view but will still work perfectly. However, a well designed system will be easier to change and to understand.

Designer Hints

Often it is difficult to decide where put certain functionality. Even professional hesitate but

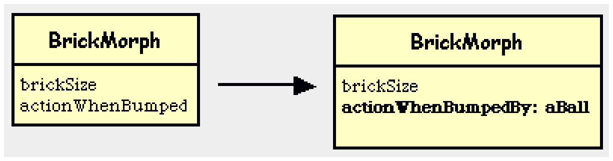


Figure 4.1: Change in `BrickMorph`

they have some techniques to evaluate their decisions. Here are some. First try to identify which object has the responsibility to carry a behavior. Ask you the question whether it is natural for an object to be responsible of the behavior.

Example. Was it the responsibility of the field to make a noise when the ball was hitting the batter. Certainly not.

Second, imagine that an object with a slightly different behavior. By imagining a possible change, you will see that you may end up having to ask yourself what is the kind of object you have to perform certain operation. If you have such question this is really that your behavior is misplaced.

Example. If we introduce a special batter having the possibility of gluing balls. We want to make a special noise. If the field would have the responsibility of making the noise, it would have to know the kind of batter currently in the game. With the second design. i.e., letting the batter controls the noise it makes. The field just notifies the batter that it has been hitted by the ball. The fact that the batter may be gluing or not is not its concern but the batter one.

Designer Hints

We take the opportunity to make the brick having the responsibility of controlling the way the ball behaves when bouncing on it as shown by the figure 4.1. Now the method `BrickMorph»actionWhenBumpedBy: aBall` has the responsibility to call the method `bounceTopBottom` or any other methods that changes the position and direction of the ball (See method 4.4). This was not the responsibility of the field to synchronized the effect of the brick getting touch and the move of the ball.

Method 4.4

```
BrickMorph>>actionWhenBumpedBy: aBall

   self flash.
   (SampledSound soundNamed: 'scratch') play.
   aBall bounceTopBottom.
   self owner removeBrick: self.
   self delete.
```

In a similar way now the method `touchedBy: aBall` is responsible for performing the visual effect when the batter is touched by the ball and to change the movement of the ball.

Method 4.5

```
BatterMorph>>touchedBy: aBall

   self flash.
   (SampledSound soundNamed: 'motor') play.
   aBall bounceTopBottom.
```

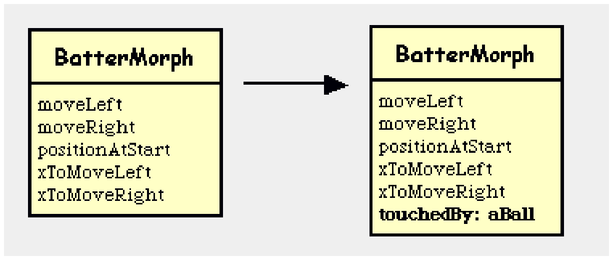


Figure 4.2: Change in `BatterMorph`

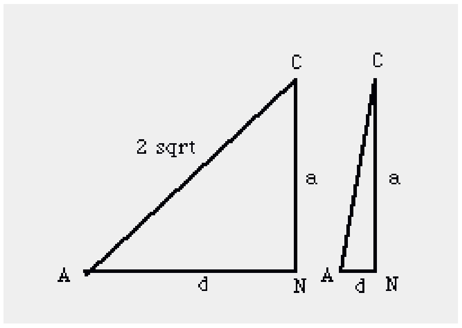


Figure 4.3:

2 Changing the bouncing

The way the ball is bouncing on the batter is quite boring because simply based on a reflection mechanism: if the ball arrives with a direction represented by `1@1` it went back with a direction of `1@ -1`. The definition of the method `BatterMorph»touchedBy: aBall` (shown in method 4.6) implements a different bouncing behavior: the closer to the center of the batter the ball bounces, the smaller is the bouncing angle relative to the vertical axe. Note that this behavior tries to avoid to speed up the ball. If you are not interested in understand how we found this formula skip the rest.

Method 4.6

```
BatterMorph>>touchedBy: aBall

    | d y x directionSign |
     self flash.
    (SampledSound soundNamed: 'motor') play.
    d := ((self center x - aBall position x) abs) * 2 / (self width).
    y := (((d*d) /2) + 2) sqrt.
    x :=  d.
    directionSign := aBall direction x sign @ aBall direction y sign.
    aBall direction: (x@y) * directionSign.
    aBall bounceTopBottom.
```

To explain you a bit the figure 4.3 illustrates the intuition of the solution. Imagine that the hypothenuse (AC) of a squared rectangle is the direction of the ball after bouncing. If we keep the distance CN constant and change the the distance AN the direction of AC will change. The figure 4.3 shows two cases one with AN having a big value and one with AN been small. If AN represents somehow the distance from the place where the ball bounced and the center of the batter we get the direction of the ball.

Now we do not want that the ball accelerates too much because the game would be impossible to play. Just computing the new direction by using the mathematical relations between A, C, and N is not sufficient. Previously the ball was moving from one pixel on x and one pixel on y so a distance of $\sqrt{2}$ pixels per move. Therefore to have a constant speed our new direction should move the ball with the same distance. So we fix the distance AC to be $\sqrt{2}$. The last problem we have is that d can be too big and will give rather flattened triangle. Therefore we decide to decide d by $\sqrt{2}$. The new direction is then $x = d$, and $y = \sqrt{2 + (d^2/2)}$ using the Pytagorus theorem that says that $AC^2 = AC^2 + CN^2$. Finally the ball should bounce so we negate the sign of x and y. As using d as the difference between the batter center and the ball position, we apply some fine tuning to get the value of d proportional to the size of the batter. There are certainly more elegant way of doing this but this is not really imprtant and we let that for you.

Note that we need to know the current direction of the ball, so we define the method `direction` as shown in method 4.7.

Method 4.7

```
BallMorph>>direction

   ^ direction
```

3 Resistant Bricks

Having one single kind of brick is somehow boring. We propose you to add resistant bricks on which the ball should bounce several times before been destroyed.Introducing a new kind of brick will raise a lot of issues and show how object-oriented programming helps to solve them.

In fact a resistant brick is a kind of brick, it should just remember how many times it has be bounced and only accepts to be destroyed when it has be bounced the right number of times. Therefore we do not have to refine from scratch a new class for this new kind of brick but we can define the behavior of a resistant brick as a refinement of the one of `BrickMorph`.

We define then the class `ResistantBrickMorph` as a subclass of the class `BrickMorph` as shown in class 4.1. This new class defines a new attribute named `resistanceNumber` that represents the resistance of the brick, *i.e.,* the number of times the brick should be touched before breaking.

Class 4.1

```
BrickMorph subclass: #ResistantBrickMorph
     instanceVariableNames: 'resistanceNumber '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'BreakOut'
```

Once the class defined, we have to define how the created bricks will be initialized. Per default, we say that a resistantBrick is a more specialized version of brick, so we first ask a resistant brick to be initialized as if it was a brick using the expression `super initializeToStandAlone` then we initialize the resistant brick to have its specific properties. Here we changed its color and shape and initialize the resistance number so that the brick should touched three times before being destroyed.

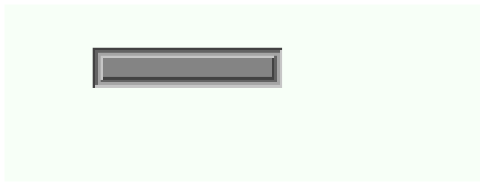


Figure 4.4: A resistant brick

Method 4.8

```
ResistantBrickMorph>>initializeToStandAlone
   "self newStandAlone openInWorld"

    super initializeToStandAlone.
    self color: (Color gray alpha: 1).
    self borderColor: (Color gray alpha: 1).
    self borderStyle: (BorderStyle complexAltFramed width: 4).
    resistanceNumber := 3
```

Resistant bricks are not destroyed immediately after been touched by the ball. To implement such a behavior we specialize the method `actionWhenBumpedBy: aBall` which is invoked when the ball enter in contact with a brick as shown by the method 4.9. First we take into account the fact that the brick has been touched by calling a new method called `decreaseResistanceNumber` that simply decreasing the resistance number (see method 4.10). If the variable is 0, this means that the brick has been touched 3 times, we simply invoke the default behavior using the expression `super actionWhenBumpedBy: aBall`. Indeed we do not want to copy of the method `BrickMorph»actionWhenBumpedBy:` because we may change it in the future and we do not want to have to change it in several places. When the resistance number is not zero then we just make the ball bouncing.

Method 4.9

```
ResistantBrickMorph>>actionWhenBumpedBy: aBall

   self decreaseResistanceNumber.
   hitNumber isZero
      ifTrue: [super actionWhenBumpedBy: aBall]
      ifFalse: [aBall bounceTopBottom]
```

Method 4.10

```
ResistantBrickMorph>>decreaseResistanceNumber

    resistanceNumber :=  resistanceNumber - 1
```

To test wether everything works, we need just be able to add resistant bricks. We just propose for now to add a line of resistant bricks just to check whether what we did is working. In the chapter **??** we will implement a way to specify where and which kind of bricks are. Define the method `resistantBrickLine:` (method 4.11) that creates a line of bricks at the row y.

Method 4.11

```
BreakOutField>>resistantBrickLine: y
   "Create a line of resistant bricks at the row y"

   | b |
   0 to: self playFieldSize x -1 do:
      [:i | b := ResistantBrickMorph newStandAlone.
      b position: (i @ y) * b brickSize.
      self addBrick: b].
```

As shown method 4.12 modify the method `initializeBricks` to invoke the method `resistantBrickLine:`.

Method 4.12

```
BreakOutField>>initializeBricks

   bricks := OrderedCollection new.
   0 to: self playFieldSize y - 12 do: [:y | self brickLine: y].
   self resistantBrickLine: self defaultPlayFieldSize y - 11
```

Now if would be excellent to add a feedback when a resistant brick has been touched. To implement this we define the method `feedbackHitDecrease` as shown in method 4.13 and we should modify the method `actionWhenBumpedBy:` to invoke it. We let you do that.

Method 4.13

```
ResistantBrickMorph>>feedbackHitDecrease

   self color: (self color alpha: (self color alpha) - 0.3).
   (SampledSound soundNamed: 'clink') play.
```

4 Avoiding Repetition

The method `resistantBrickLine:` method **??** and the method `brickLine:` 3.4 only differ by the kind of bricks that are created. Having duplicate logic is always a problem because changing one will imply changing the other. This means that they is an occasion to refactor code. For this purpose we define the method

Method 4.14

```
BreakOutField>>brickLine: y ofKind: aBrickClass

   | b |
   0 to: self playFieldSize x -1 do:
      [:i | b := aBrickClass newStandAlone.
         b position: (i @ y) * b brickSize.
         self addBrick: b].
```

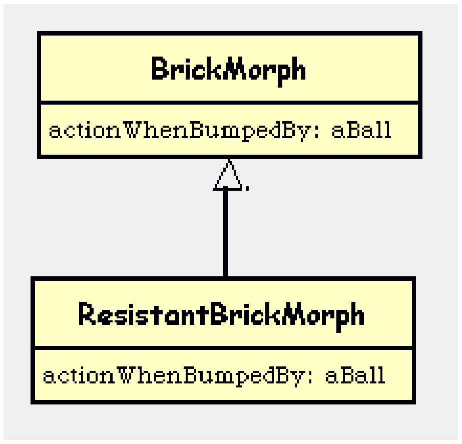


Figure 4.5:

Method 4.15

```
BreakOutField>>resistantBrickLine: y

   self brickLine: y ofKind: ResistantBrickMorph
```

Method 4.16

```
BreakOutField>>brickLine: y

   self brickLine: y ofKind: BrickMorph
```

Note that we passed classes as message arguments as any objects such as the point 10@100 or the number 100. In Smalltalk classes are just objects as any entities in the system.

5 Methods are the Unit of Specialization

In this section we will show you how breaking a big method into smaller ones allows one to have method part that can be easily specialized. The creation of a specific method avoids one to copy an entire method when we only want to specialize a part of it. When a brick or a resistant brick is destroyed, the *same* sound gives some audio feedback. To add more gameplay we would like that both kind of brick destruction is accompanied by a different sound. At first this is not obvious. Let us recap the situation. The method `BrickMorph»actionWhenBumpedBy: aBall` makes the brick flashing, plays a sound, makes the ball bouncing, and delete itself from the game.

Method 4.17

```
BrickMorph>>actionWhenBumpedBy: aBall

   self flash.
   (SampledSound soundNamed: 'scratch') play.
   aBall bounceTopBottom.
   self owner removeBrick: self.
   self delete.
```

As shown in method 4.9, for a resistant brick, the method`ResistantBrickMorph»actionWhenBumpedBy: aBall` checks whether the brick should be destroyed or not as explained before.

A First Wrong Approach. The first natural idea that came is to add in this method the message to play sound as shown in the method 4.18. But this solution does not work because when the brick is destroyed two sounds are played: `'scratch'` the one of the BrickMorph and `'coyote'` the one of the ResistantBrickMorph.

Method 4.18

```
(Wrong Solution)
ResistantBrickMorph>>actionWhenBumpedBy: aBall

   self decreaseResistanceNumber.
   hitNumber isZero
      ifTrue:
         [super actionWhenBumpedBy: aBall.
         (SampledSound soundNamed: 'coyotte') play.]
      ifFalse: [aBall bounceTopBottom]
```

Another wrong solution. The problem is that we cannot control the sound emitted in the `BrickMorph»actionWhenBumpedBy:` method. We could copy it entirely in the `ResistantBrick` method as shown in method 4.19. This solution only produces one sound. But the fact that a solution is working is often not enough. This solution can lead to lot of problems as soon as we decide to change the `actionWhenBumpedBy:` method of the class `BrickMorph`. Indeed we will have to repercut all the changes to this methods. Generalizing such a practice would lead to a system when we would spent all our time propagating changes between copied methods.

Method 4.19

```
(Wrong Solution)
ResistantBrickMorph>>actionWhenBumpedBy: aBall

   self decreaseResistanceNumber.
   hitNumber isZero
      ifTrue:
         [self flash.
         (SampledSound soundNamed: 'coyotte') play.
         aBall bounceTopBottom.
         self owner removeBrick: self.
         self delete]
      ifFalse: [aBall bounceTopBottom]
```

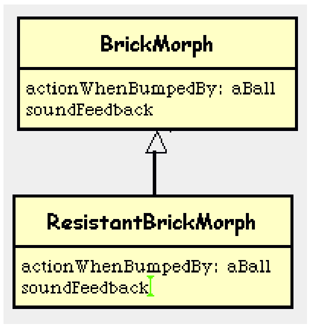
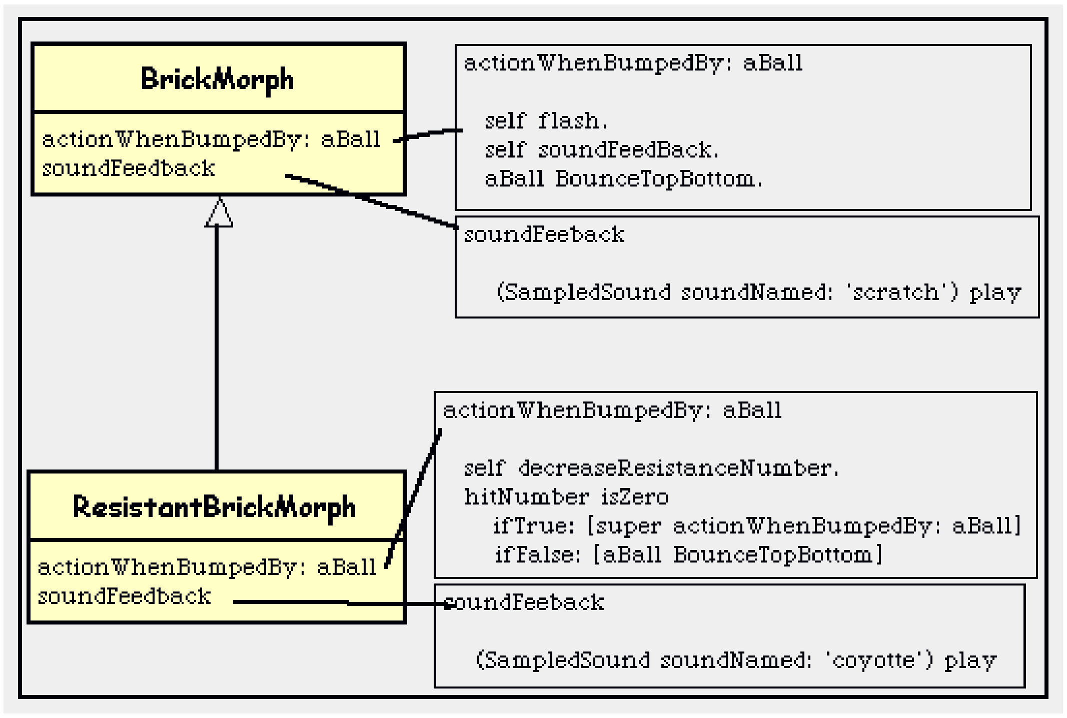


Figure 4.6:



```
actionWhenBumpedBy: aBall

  self flash.
  self soundFeedBack.
  aBall BounceTopBottom.
```

```
soundFeeback

   (SampledSound soundNamed: 'scratch') play
```

```
actionWhenBumpedBy: aBall

  self decreaseResistanceNumber.
  hitNumber isZero
     ifTrue: [super actionWhenBumpedBy: aBall]
     ifFalse: [aBall BounceTopBottom]
```

```
soundFeeback

   (SampledSound soundNamed: 'coyotte') play
```

Figure 4.7:

The Solution. In fact the problem is that we would like to be able to change only the sound emitted. We want to keep all the behavior of the method `BrickMorph»actionWhenBumpedBy: aBall` but be able to specify a different noise on the class `ResistantBrick`. The solution is quite simple in fact, we just have to extract the code emitting the sound from the method `actionWhenBumpedBy: aBall` as shown in method 4.21 and moving it to a new method called `soundFeedback` as in method 4.20.

Method 4.20

```
BrickMorph>>soundFeedback

    (SampledSound soundNamed: 'scratch') play.
```

Method 4.21

```
BrickMorph>>actionWhenBumpedBy: aBall

   self flash.
   self soundFeedback.
   aBall bounceTopBottom.
   self owner removeBrick: self.
   self delete.
```

You may wonder why this is solving our problem since right now we did not introduce anything new. This is right that for the class `BrickMorph` we did not change anything. But we introduced the possibility for subclasses to specialize a part of the brick behavior. Now for the class `ResistantBrickMorph` we just have to define a different method `soundFeedback` as method 4.22 which when the method `actionWhenBumpedBy: aBall` will be invoked *on a resistantBrick* will be invoked instead of the method of `BrickMorph»soundFeedback`.

Method 4.22

```
ResistantBrickMorph>>soundFeedback

    (SampledSound soundNamed: 'coyotte') play.
```

If we step back we can summarize that methods represent potential parts of a whole that can be changed by subclasses. This behavior is so important that the following Section will take the time to let you digest that. In fact we already explained in the chapterchapter **??**. But this is worth that you read it again in the specific context of the BreakOut.

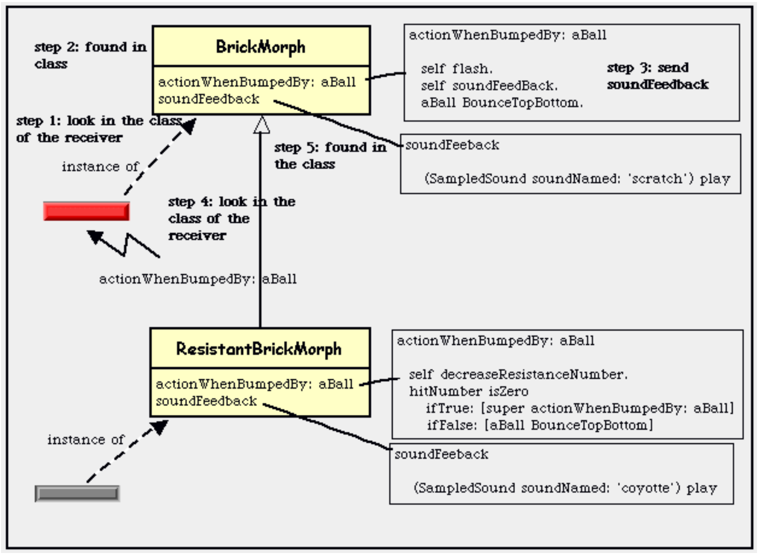
6 A Closer Look at Inheritance

It is now the time to step back and take the time to really understand how the lookup of methods works and the interest of inheritance. These concepts are at the same time simple and complex. We suggest you to read the chapter chapter **??**. Let us look step by step the way different messages are searched and executed.

6.1 `aBrickMorph actionWhenBumpedBy: aBall`

What is happening when a brick, instance of the class `BrickMorph`, receives the message `actionWhenBumpedBy: aBall`? The lookup of the different methods involved `actionWhenBumpedBy:` and `soundFeedback` is illustrated by the Figure 4.8 and follow the steps.

Step 1: the message `actionWhenBumpedBy:` is looked up in the class of the message receiver, here `BrickMorph`.


step 2: found in class
BrickMorph
actionWhenBumpedBy: aBall
soundFeedback
step 3: send soundFeedback
step 1: look in the class of the receiver
instance of
step 5: found in the class
step 4: look in the class of the receiver
actionWhenBumpedBy: aBall
ResistantBrickMorph
actionWhenBumpedBy: aBall
soundFeedback
instance of

```
actionWhenBumpedBy: aBall

  self flash.
  self soundFeedBack.
  aBall BounceTopBottom.
```

```
soundFeeback

  (SampledSound soundNamed: 'scratch') play
```

```
actionWhenBumpedBy: aBall

  self decreaseResistanceNumber.
  hitNumber isZero
     ifTrue: [super actionWhenBumpedBy: aBall]
     ifFalse: [aBall BounceTopBottom]
```

```
soundFeeback

  (SampledSound soundNamed: 'coyotte') play
```

Figure 4.8: When a brick instance of the clas `BrickMorph` receives the message `actionWhenBumpedBy: aBall`.

Step 2: The class `BrickMorph` defines such a method so it is executed.

Step 3: We skip for now the first message `self flash` and imagine that it got found and executed. Then the message `soundFeedback` is sent to `self`.

Step 4:`self` represents the original receiver of the message `actionWhenBumpedBy:`. therefore the method `soundFeedback` is looked up in the class `BrickMorph`.

Step 5: The class `BrickMorph` defines a method `soundFeedback` so it is executed.

This behavior is the one we expected, the message are looked up in the class of the object that receives the message.

6.2 `aResistantBrickMorph actionWhenBumpedBy: aBall`

What is happening when a brick, instance of the class `ResistantBrickMorph`, receives the message `actionWhenBumpedBy: aBall`? The Figure 4.9 illustrates the steps involved.

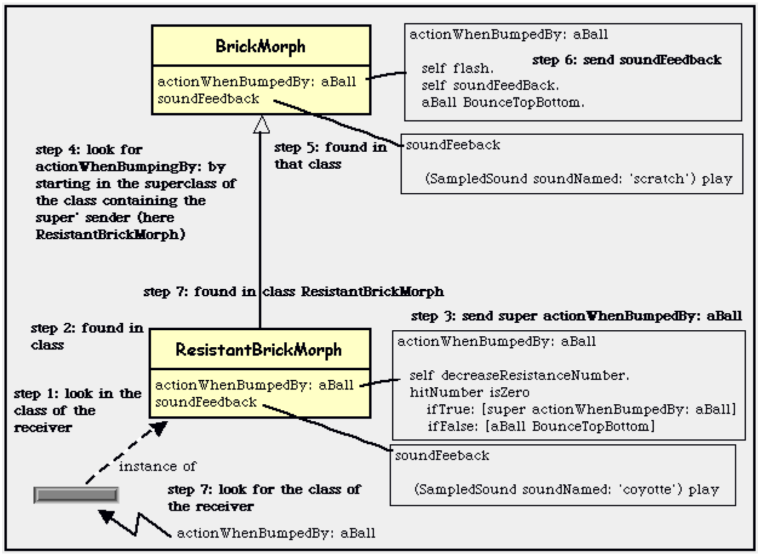
Step1: the message `actionWhenBumpedBy:` is looked up in the class of the message receiver, here `ResistantBrickMorph`.

Step2: The class `ResistantBrickMorph` defines such a method so it is executed.

Step3: We skip for now the first messages and act as if this is the third times that the brick is bumped by the ball. Therefore the message `super actionWhenBumpedBy: aBall` should be executed.

Step 4: `super` represents the original receiver of the message `actionWhenBumpedBy:`, the resistant brick. *But* it indicates that the method lookup should start in the superclass of **the class defining the method that call super** (we will illustrate this subtle point in the following example). So the class of the method containing the super expression is `ResistantBrickMorph`, its superclass is `BrickMorph`.

Step 5: The method `actionWhenBumpedBy:` is looked up in the class `BrickMorph`, and found there.


BrickMorph
actionWhenBumpedBy: aBall
soundFeedback
step 6: send soundFeedback
step 4: look for actionWhenBumpingBy: by starting in the superclass of the class containing the super' sender (here ResistantBrickMorph)
step 5: found in that class
step 7: found in class ResistantBrickMorph
step 3: send super actionWhenBumpedBy: aBall
step 2: found in class
ResistantBrickMorph
actionWhenBumpedBy: aBall
soundFeedback
step 1: look in the class of the receiver
instance of
step 7: look for the class of the receiver
actionWhenBumpedBy: aBall

```
actionWhenBumpedBy: aBall

  self flash.
  self soundFeedBack.
  aBall BounceTopBottom.
```

```
soundFeeback

  (SampledSound soundNamed: 'scratch') play
```

```
actionWhenBumpedBy: aBall

  self decreaseResistanceNumber.
  hitNumber isZero
    ifTrue: [super actionWhenBumpedBy: aBall]
    ifFalse: [aBall BounceTopBottom]
```

```
soundFeeback

  (SampledSound soundNamed: 'coyotte') play
```

Figure 4.9: When a resistant brick instance of the class `ResistantBrickMorph` receives the message `actionWhenBumpedBy: aBall`.

Step 6: The method `BrickMorph»actionWhenBumpedBy:` is executed. In particular, the expression `self soundFeedback` is executed.

Step 7: `self` represents the original receiver of the message `actionWhenBumpedBy:`. therefore the method `soundFeedback` is looked up in the class `ResistantBrickMorph`.

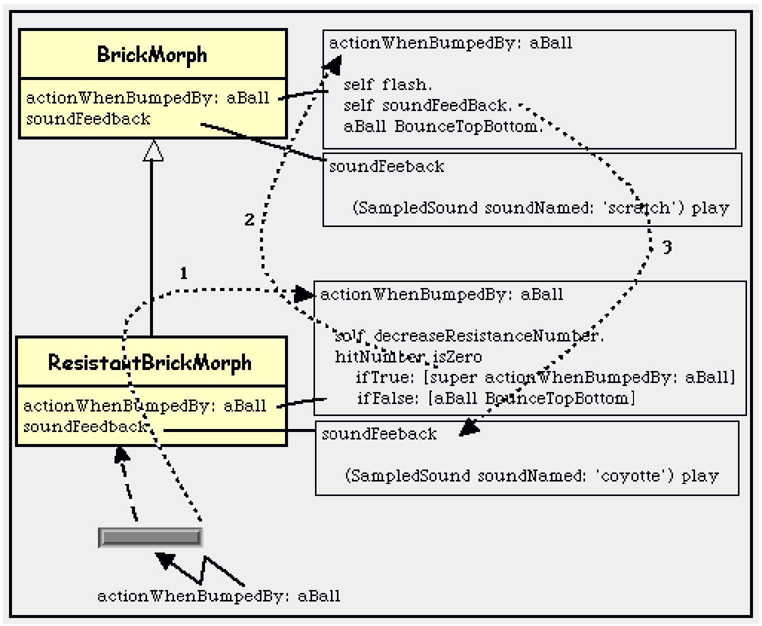
Step 8: The class `ResistantBrickMorph` defines a method `soundFeedback` so it is executed.

Step 9: If the method `ResistantBrickMorph»actionWhenBumpedBy:` would define other expressions after the conditional. They would be executed following these rules

The Figure **??** shows the way the methods are invoked. It shows also that a *self send* works as a hook or hole in the method that uses it. When an method contains a self send, it means that subclasses can specialize the behavior by defining new methods that will be invoked instead of the original method.

Therefore we say that `self` is dynamic in the sense that the method executed really depends on the object that receives it (and where the method is defined). It looks like if the methods of the subclasses would take the place of the original method. However this is not really true because a method in a subclass can still invoked the method it hides or overrides. For example the method `ResistantBrickMorph»actionWhenBumpedBy:` executes some messages that are important realizing the behavior of a resistant brick, then when necessary it invokes the behavior it was hidding, here the default action performed when a brick is touched by a ball (as shown in the The Figure 4.10. `super` is necessary when a method in a subclass wants to do some specific behavior but still be able to perform the original method. Note that

`self` represents the original receiver. What is important to see if that `self` *always* represents the receiver of the original message. In the Step 6 above, when a resistant brick receives the message `actionWhenBumpedBy:` and execute the expression `self soundfeedBack`, even if the expression is defined in the class `BrickMorph`, `self` in this particular execution represents an instance of the class `ResistantBrickMorph` and not `BrickMorph`. Therefore the noise produced is the one associated with the class `ResistantBrickMorph` because this class redefines the method `soundFeedback`. This behavior is illustrated in the Figure 4.10 where the step 3 shows that the method defined in the subclass is called when the `self soundefeedBack` expression is evaluated.


BrickMorph
actionWhenBumpedBy: aBall
soundFeedback
ResistantBrickMorph
actionWhenBumpedBy: aBall
soundFeedback
1
2
3
actionWhenBumpedBy: aBall

```
actionWhenBumpedBy: aBall

  self flash.
  self soundFeedBack.
  aBall BounceTopBottom.
```

```
soundFeeback

   (SampledSound soundNamed: 'scratch') play
```

```
actionWhenBumpedBy: aBall

  self decreaseResistanceNumber.
  hitNumber isZero
    ifTrue: [super actionWhenBumpedBy: aBall]
    ifFalse: [aBall BounceTopBottom]
```

```
soundFeeback

  (SampledSound soundNamed: 'coyotte') play
```

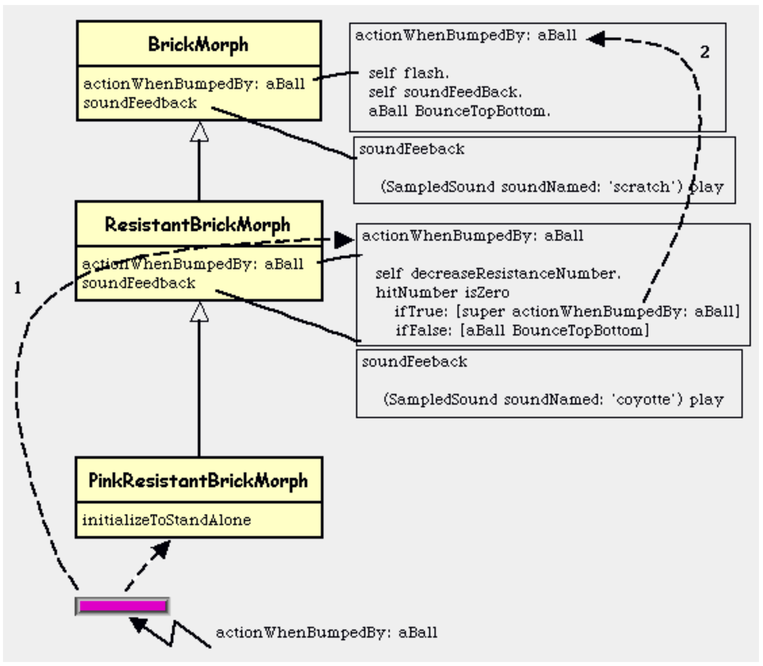
Figure 4.10: `super` allows one to invoke methods that would not be called because of the overriding in subclasses. `self` always represents the original message receiver.

`super` represents the receiver but changes the method lookup. `super` changes the way the methods are looked up. As shown in Figure 4.11, imagine that we have an instance of a subclass of `Resistant-BrickMorph`, the `PinkResistantBrickMorph` class. When an instance of this new class receives the message `actionWhenBumpedBy:`, the method is looked up starting in the class of the instance, so it starts in the class `PinkResistantBrickMorph`. As the class doesn't redefine the method `action-WhenBumpedBy:`, the lookup continues in its superclass `ResistantBrickMorph` which defines the method. This method is executed as explained previously. Now when the expression `super action-WhenBumpedBy:` in the method `ResistantBrickMorph»actionWhenBumpedBy:` is executed, the overridden method is looked up in the superclass of the class containing the super send, i.e., in `BrickMorph` the superclass of `ResistantBrickMorph`. What is important to understand is that `super` does not take use the information about the original receiver to perform the lookup of the method. The lookup initiated by `super` just starts above the class that use it. That's why we say that `super` is static in the sense that the method lookup does not depend on the class of the receiver but only where the method using it is located in the hierarchy.

Note that this behavior is the one we want. Oftentimes people shortens the exact understanding of `super` by saying that `super` makes the lookup starts in the superclass of the class of the receiver. This is plain wrong. In the last example illustrated Figure4.10 following such wrong definition would means looking `actionWhenBumpedBy:` in `ResistantBrickMorph` (because the superclass of the class of an instance of the class `PinkResistantBrickMorph` is `ResistantBrickMorph`) again and would lead to an infinite loop. No object-oriented languages using the concept of super would work with this wrong definition.

7 Lessons Learnt

Sometimes to be able to reuse a class we need to cut its methods in smaller ones.
methods are the unit of reuse
Inheritance allow incremental definition.
self refers to the receiver



```
actionWhenBumpedBy: aBall

  self flash.
  self soundFeedBack.
  aBall BounceTopBottom.
```

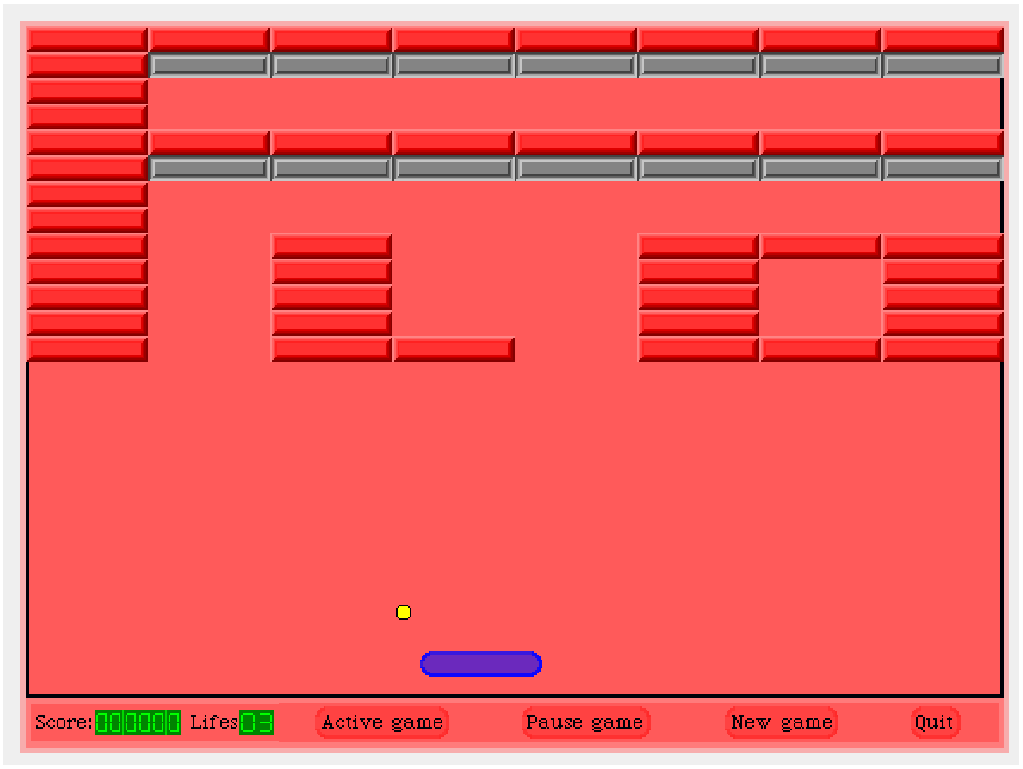
```
soundFeeback

   (SampledSound soundNamed: 'scratch') play
```

```
actionWhenBumpedBy: aBall

  self decreaseResistanceNumber.
  hitNumber isZero
     ifTrue: [super actionWhenBumpedBy: aBall]
     ifFalse: [aBall BounceTopBottom]
```

```
soundFeeback

   (SampledSound soundNamed: 'coyotte') play
```

Figure 4.11: `super` allows one to invoke methods that would not be called because of the overriding in subclasses. `self` always represents the original message receiver. `super` makes the lookup of the method starting in the superclass of the class containing the `super` send.

5
Level Description



In this chapter we introduce a way to define different levels. In the next chapter we will introduce then the possibility to finish a level and pass to the next one. This chapter introduce....

Up until now we position the bricks using different programs. This way of doing things does not go really far because we cannot be forced to program one by one all the levels. Adding a new level should require no coding at all. Instead we would like to have a way to represent the different kind of bricks and having a way to interpret this representations to populate the break out field with the correct bricks.

1 Level Representations

We have to choose a representation for the levels so that after we can create various levels in a easy way and in a completely generic manner. We do not want to have to code one single line of code afterwards to generate a new levels. We chose to represent a level by a string as the one below.

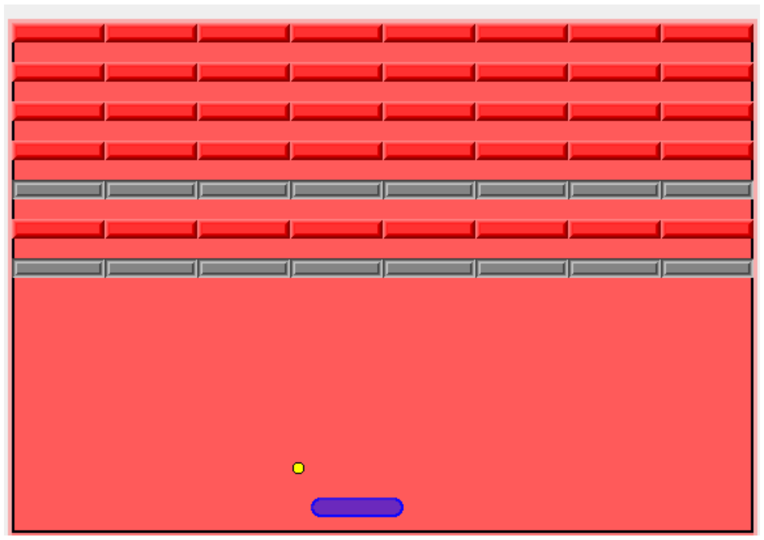


Figure 5.1: The level corresponding to the level description returns by `BreakOutField level0`

Script 5.1 (*Our level representation*)

```
'SSSSSSSS
EEEEEEEE
SSSSSSSS
EEEEEEEE
SSSSSSSS
EEEEEEEE
SSSSSSSS
EEEEEEEE
XXXXXXXX
EEEEEEEE
SSSSSSSS
EEEEEEEE
XXXXXXXX
'
```

The idea is that each line in the string represents a line of bricks. Each element is a code that represents the kind of brick. The game should be able to read and to interpret such a string and create the corresponding bricks. One of the problem we have is how to store the levels descriptions so that we will not have to enter again and agin. We could have save them on file but we chose a simple approach that consist of defining each level descriptions as a method of the class BreakOutField. This way we can edit them and save them with the game. So method

Method 5.1

```
In category levels
BreakOutField class>>level0

^
'SSSSSSSS
EEEEEEEE
SSSSSSSS
EEEEEEEE
SSSSSSSS
EEEEEEEE
SSSSSSSS
EEEEEEEE
XXXXXXXX
EEEEEEEE
SSSSSSSS
EEEEEEEE
XXXXXXXX
'
```

Once you have defined the `level0` method, you can get the level representation by executing `BreakOutField level0`. The Figure 6.1 corresponds to the level description returns by `BreakOutField level0`. For that we interpreted the `S` as normal `BrickMorph`, `E` as empty slot, and `X` as `ResistantbrickMorph`.

2 Template generation

Define another level description in your own method on the class side. As you certainly notice this is boring to count exactly the number of bricks so to ease the generation of level description we would to define a method, called `generateEmptyTemplate` that generates a level description with the right number of bricks but all empty bricks.

In fact the logic of such a method is quite simple, we have to have two loops , one for the lines and one for the row and to put at the end of each line a carriage return character (`Character cr`). The first simple problem we have to solve is that we do not know the number of rows and columns of a field. Right now to know the size of the complete area we have to create an instance of `BreakOutField` and send it the message `playFieldSize` (see method 5.2) as follow |ctBreakOutField new playFieldSize.

Method 5.2

```
In category default
BreakOutField>>playFieldSize

   ^ 8@26
```

However, we do not want to have to create an instance just for the sake of getting the default size of the field. So we propose to define such a method on the class `BreakOutField` itself as the class always exists (See method 5.3). This way we are able to get the size without having to create extra instance.

Method 5.3

```
In category default
BreakOutField class>>playFieldSize

   ^ 8@26
```

Then we change the method method 5.2 into method 5.4 to avoid the duplication of size information. This way changing the size of the break out only requires to specify it at one single place.

Method 5.4

```
In category default
BreakOutField>>playFieldSize

   ^ self class playFieldSize
```

Now to get the number of rows we have to execute `self playFieldSize x` independent of the fact that we are editing an instance or class method. But this is a bit clumsy. Indeed the reader of the code have to understand that `playFieldSize` returns a point. As we do not show that we used a point to encode the size of the field we define simply two methods `maximumColumnNumber` and `maximumRowNumber`. They will help us to write code that does not show the way we encoded the field size and communicate better its intent. The method `maximumRowNumber` does not return the complete size of brick line but only the one where we can have bricks.

Method 5.5

```
In category default
BreakOutField class>>maximumColumnNumber
  "Return the number of rows that the field has"

   ^ self playFieldSize x

BreakOutField class>>maximumRowNumber
  "Return the maximum number of brick lines that the field can have"

   ^ self playFieldSize y - 13
```

The method `generateEmptyTemplates` described by method 5.6 uses a stream. We do not want to go into the detail of stream but a stream is a kind of structure in which other objects can be put sequential and can later retrieve. Here we create a `ReadWriteStream`, *i.e.,* a stream in which we can put and read from element. We only use a limited functionality of stream. First we create a stream using the `on:` that requires a container for the elements we will put in the stream. We can adding character so we create a string with a default size. The size does not really matter because the stream will make the string grows if necessary. Then we use two loops: `self columnNumber timesRepeat: [aStream nextPut: $E].` create one line in the level description, after which we put a carriage return using the expression `Character space`. `nextPut: anElement` puts the element in the stream. We repeat the first loops to create all the lines. Then we return the filled string by asking the stream its contents using the message `contents`.

Method 5.6

```
In category template
BreakOutField class>>generateEmptyTemplate
   "Generate an empty level description template"

   | aStream |
   aStream := ReadWriteStream on: (String new: 100).
   self maximumRowNumber  timesRepeat: [
      self maximumColumnNumber timesRepeat: [ aStream nextPut: $E].
      aStream nextPut: Character cr].
   ^ aStream contents
```

Script 5.2 (*Invoking* `generateEmptyTemplate`)

```
BrekaOutField generateEmptyTemplate

'EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
EEEEEEEE
'
```

3 About Conveying Intention

Oftentimes we are creating methods such as `numberOfBrickLines` instead of using `playFieldSize y -12` and you may wonder because the two expressions are strictly equivalent. This is true however a big part of programming is not only t about designing the correct algorithm but also to write code that other persons or yourself in the future can read. Using the right terms and providing extra methods is a key point in writing applications. Note also that in general you will spend most of the time reading the code of somebody else to understand it and modify it. Therefore conveying the intention of the code is really important and using the correct terms or hiding certain information is an key activity.

4 Interpreting the Level Description

Now we are ready to implement a method that interprets the level description and create the bricks into the field. Let us call this method `installLevel: aLevelDescription`. Such a method is invoked for example as follow `aBreakOut installLevel: aBreakOutField class level0`. Now we have to know the character at a given position in the level description and create the corresponding brick. However, we cannot access the character directly using a row and a column. The only way to access a character inside a string is to specify the character position by reference to the beginning of the string. The message `at: aNumber` sent to string returns the character at the aNumber position. In the Figure **??**, the character `r` which is at the column 3 and row 2, the 12th element of the string: 9 characters per line plus 3 for the column.

We have to take care that in one line of the level description there is not eight characters but nine because we should not forget the carriage return character at the end. That is the 1 character of the second line is the tenth element and not the nineth. Let us implement the method `brickDesccriptionAtColumn:row:` that returns the results presented in the script **??**.

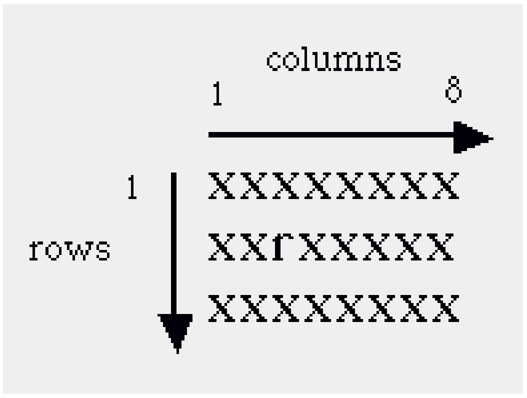


Figure 5.2: the character `r` which is at the column 3 and row 2, the 12th element of the string: 9 characters per line plus 3 for the column.

Script 5.3 (*Results of* `brickDesccriptionAtColumn:row:`)

```
| b |
b := BreakOutField new.
b brickDesccriptionAtColumn: 1 row: 1
    -Print It-> 1

b brickDesccriptionAtColumn: 2 row: 1
    -Print It-> 2

b brickDesccriptionAtColumn: 1 row: 2
    -Print It-> 10

b brickDesccriptionAtColumn: 1 row: 3
    -Print It-> 19

b brickDesccriptionAtColumn: 8 row: 2
    -Print It-> 17


BreakOutField level0 at: (b brickDesccriptionAtColumn: 8 row: 3)
    -Print It-> $S
```

Before looking at our solution, try to implement it because it is a good exercise. To help you you can use a Transcript and an expression that print the method argument and the result you would obtain such as for example `Transcript show: aColumn printString, , aRow printString; show: ' ' ; show: (aColumn + (aRow* self class maximumColumnNumber)) ; cr`

Method 5.7

```
In category level handling
BreakOutField>>brickDesccriptionAtColumn: aColumn row: aRow
  "Return the character description been at the position given by aPoint"

  ^ (aColumn + ((aRow-1)*(self class maximumColumnNumber+1)))
```

In the method `BreakOutField»brickLine:` (see method **??**) and that we show again hereafter, we first create an instance then position it.

Method 5.8

```
BreakOutField>>brickLine: y

  | b |
  0 to: self playFieldSize x -1 do:
    [:i | b := BrickMorph newStandAlone.
      b position: (i @ y) * b brickSize.
      self addBrick: b].
```

Now the class of the created brick depends on the brick description character that we get from the level description. Therefore we have to find a way to get the class associated with a given character. For this purpose we define the method `brickClassFromDescription: aCharacter` as shown in method 5.9 that given a character returns the associated class. Note that such a method does not handle the case of the character `E` because we do not have class associated with empty brick. This method works but is not the best one. We will show later how a much better solution is possible. We let you think for now on the problems of such a solution.

Method 5.9

```
In category level handling
BreakOutField>>brickClassFromDescription: aCharacter

   aCharacter = $S
      ifTrue: [^ BrickMorph].
   aCharacter = $X
      ifTrue: [^ ResistantBrickMorph].
```

The script 5.4 presents how we the method `brickClassFromDescription:` is used to create bricks.

Script 5.4 (*Using brickClassFromDescription)*

```
(BreakOutField new brickClassFromDescription: $S) newStandAlone openInWorld
```

Now we are ready to implement the method `installLevel: aDescription`. The method 5.10 shows a possible implementation. The final point to resolve is to place the bricks at the right place in the field. Here we have to pay attention that the position of the brick at the column 1 and row 1 is `0@0` relatively to the field. The method `placeBrick: aBrick atBrickCoordinate: aPoint` (method 5.11) ensures first that the created brick has the default brick size, then it positions the brick by transforming the logical coordinates given by the level description to concrete coordinate in terms of brick size. Note that the position of a brick is global and not by reference to its container (which a design mistake of the Morphic system), therefore we should not forget to add the position of the field.

Method 5.10

```
In category level handling
BreakOutField>>installLevel: aDescription
  "Create all the bricks as specified by the description"

  | char brick |
  1 to: self class maximumRowNumber do:
    [:row |
    1 to: self class maximumColumnNumber do:
      [:column |
      char := aDescription at: (self brickDescriptionAtColumn: column row: row).
      char = $E
        ifFalse: [ brick := (self brickClassFromDescription: char) newStandAlone.
            self addBrick: brick.
            self placeBrick: brick atBrickCoordinate: (column-1)@(row-1)]]]
```

Method 5.11

```
In category bricks
BreakOutField>>placeBrick: aBrick atBrickCoordinate: aPoint
  "Place the given brick at the given position taking into account the
  default size of bricks"

  | brickSize |
  brickSize := BrickMorph defaultBrickSize.
  aBrick extent: brickSize.
  aBrick position: (aPoint * brickSize) + self position.
```

Now we have to invoke the method `installLevel:` when the field is initialized. In a following chapter we will introduce the fact that we change levels when we finish them. Therefore we change the method `BreakOutField»initializeToStandAlone` to invoke the method `installLevel:` and `initializeBricks` to remove the line of brick creation as shown hereafter.

Method 5.12

```
In category initialization
BreakOutField>>initializeToStandAlone

  super initializeToStandAlone.
  self bounds: (0@0 corner: BrickMorph defaultBrickSize * self playFieldSize).
  self color: (Color red alpha: 0.5).
  self ball: BallMorph newStandAlone.
  ball positionAtStart.
  self batter: BatterMorph newStandAlone.
  batter positionAtStart.
  self initializeBricks.
  self installLevel: self class level0

BreakOutField>>initializeBricks

  bricks := OrderedCollection new.
```

5 About Design

This section goes a bit far in comparing different choices we have to represent level description management and then can be skipped in a first reading.

We took the choice to define the behavior related to level generation in the class level of the class `BreakOutField`. That way we could simply send messages to the class `BreakOutField` itself as for example `BreakOutField level0`. However, doing that we have mixed two different responsibilities: playing the game and managing level descriptions. What is implied in the choice we made is that we only have one set of level descriptions. Indeed when we send the message `BreakOutField gatherAllDefinedLevels` we do not not distinguish between different sets we get all the defined levels.

In our case this is not a real problem, because we defined only a couple of methods but you should be aware that having too much methods at the class level not related to strict class responsibilities is a sign of bad design. One of the most striking example of bad design in the squeak 3.2 environment is the class `@@Wiki@@`. Because it represents...and its instance method....

Alternatively in our case we could have created a separated class named `LevelDescriptionManager` whose role would have been to manage (support definition and store) of description levels.

The choice we made is valid while we do not have to have multiple instances managing different sets of level description. For example, if we would like to have multiple objects responsible for storing different sets of level descriptions then our choice would start to be clumsy because we would have started to need instance variables at the class side representing something else than class properties and it would be much better to have one new class with clearly identified state and responsibilities.

A basic implementation of the same behavior we had implemented on the class side of the class `BreakOutField` is to define the class `LevelDescriptionManager`.

Class 5.1

```
Object subclass: #LevelDescriptionManager
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

Try to introduce such a class in your implementation to identify the change you would have to make. Introducing different level description sets would require to add some extra state and a different way of storing the level descriptions.

The method `brickDescriptionAtColumn: aColumn row: aRow` is not using any of the instance variables of a breakOutField instance and nearly no functionality. This is often the sign that the method is not defined on the correct class. It might mean that a class is missing, the class `LevelDescription` that would represent a level description and provide such a kind of behavior. Here we decided not to define it because having a string and using a method to store it was the simple thing to do. However, in other applications defining methods that do not access nor instance variables nor methods is a sign that a class is missing or that the method is defined on the wrong class. This does not mean that you have to create a class or change the method class immediately but that you should pay attention at other signs that would indicate it.

About Changes. What is important when designing a system is that we should always be ready to change it when new functionality are required. Oftentimes people try to forsee all the possibilities upfront. Therefore they often design systems that are bloated with unnecessary code and cases. Note that having tests help to check whether the changes introduced break the functionality already working is an important help. In this book we did not introduce testing as we focus more on teaching the concept of object-oriented than good software engineering practices.

6 UndestroyableBricks

Now we want to introduce another kind of brick: bricks that cannot be destroyed. Now you should be able to add such a functionality without looking at our solution. An undestroyable brick is a brick that only produces a sound when it is bumped by the ball. We do not present in detail our solution.

Class 5.2

```
BrickMorph subclass: #UndestroyableBrickMorph
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

Method 5.13

```
In category [initialization]
UndestroyableBrickMorph>>initializeToStandAlone

   super initializeToStandAlone.
   self color: (Color r: 0.6 g: 1.0 b: 1.0).
   self borderStyle: (BorderStyle complexAltFramed width: 3).
```

Method 5.14

```
In category initialization
UndestroyableBrickMorph>>initializeToStandAlone

   super initializeToStandAlone.
   self color: (Color r: 0.6 g: 1.0 b: 1.0).
   self borderStyle: (BorderStyle complexAltFramed width: 3).
```

Method 5.15

```
In category action
UndestroyableBrickMorph>>actionWhenBumped

   (SampledSound soundNamed: 'clink') play
```

Now if we want to be able to use this new kind of brick we have to change the method `BreakOut-Field»brickClassFromDescription: aCharacter` so the system knows that we use the character `$U` to represent `UndestroyableBrickMorph` in level description.

Method 5.16

```
In category level handling
BreakOutField>>brickClassFromDescription: aCharacter

   aCharacter = $S
      ifTrue: [^ BrickMorph].
   aCharacter = $X
      ifTrue: [^ ResistantBrickMorph].
   aCharacter = $U
      ifTrue: [^UndestroyableBrickMorph]
```

Edit the method `BreakOutField class»level0` and check that the game works well.

7 Automatic Brick Registration

As we saw in the previous section adding a new brick requires to change the method `brickClassFromDescription: aCharacter` and this is the sign that our design is not that good. In fact there are several problems: first we have to recompile the method `brickClassFromDescription:` each time we add a new kind of bricks, second the association between a character and a class is not clearly explictly. This is the fact logic of the method `brickClassFromDescription` while it would be better that the brick itself specify such an association. In fact we would like to show you how a simple registration mechanism can solve this problem. The idea is that each kind of brick knows the character that represents it in a level description. Then each kind of brick tells this information to the `BreakOutField` which uses it to know which kind of brick to create. This way new kinds of bricks do not have to modify the class `BreakOutField`. This example is also the opportunity to show you how dictionaries, a collection that associates a value to a key, work.

Associating a Character to Brick Classes

The first step is to let the responsibility to associate a character to the brick classes

Method 5.17

```
In category description
BrickMorph class>>descriptionCharacter

   ^ $S
```

Method 5.18

```
In category description
ResistantBrickMorph class>>descriptionCharacter

   ^ $X
```

Method 5.19

```
In category description
UndestroyableBrickMorph class>>descriptionCharacter

   ^ $U
```

The first expression of the script 5.5 shows how we can access the description character declared by a class. The second expression returns all the subclasses of the class `BrickMorph`. The third expression use the method `withAllSubclasses` which returns a collection of all the subclasses of the receiver including itself. The last expression shows how we can get all the subclasses of the class `BrickMorph` including itself and collect their declared description character.

Script 5.5 (*Accessing description characters)*

```
BrickMorph descriptionCharacter
    -Print It-> $S


BrickMorph allSubclasses
    -Print It-> a Set(UndestroyableBrickMorph ResistantBrickMorph)


BrickMorph withAllSubclasses
    -Print It-> a Set(UndestroyableBrickMorph BrickMorph ResistantBrickMorph)


BrickMorph withAllSubclasses collect:
   [:aClass | aClass descriptionCharacter]
    -Print It-> a Set($U $X $S)
```

Setting the Registration

The idea is that we use a dictionary, a collection that associates a value to a key. Each kind of brick then register its class associated with the character that represents it. Then by querying the dictionary we automatically know which class to instantiate. The script 5.6 shows a typical use of dictionary: we create one, put some values associated with key, here we associated names of animal with their weight and we query it.

Script 5.6 (*Example of Dictionary Use)*

```
| d |
d := Dictionary new.

"Now we associate animal names and their weight"
d at: #monkey put: 50.
d at: #elephant put: 1000.
d at: #ant put: 0.001.
d at: #dog put: 50.

"Now we can get the weight of an animal by asking the value associated
with the animal name"
d at: #monkey
    -Print It-> 50

d at:#elephant
    -Print It-> 1000
```

We first add a new instance variable `brickDescriptions` to the class `BreakOutField`. It will contain the mapping between the description characters and the classes.

Class 5.3

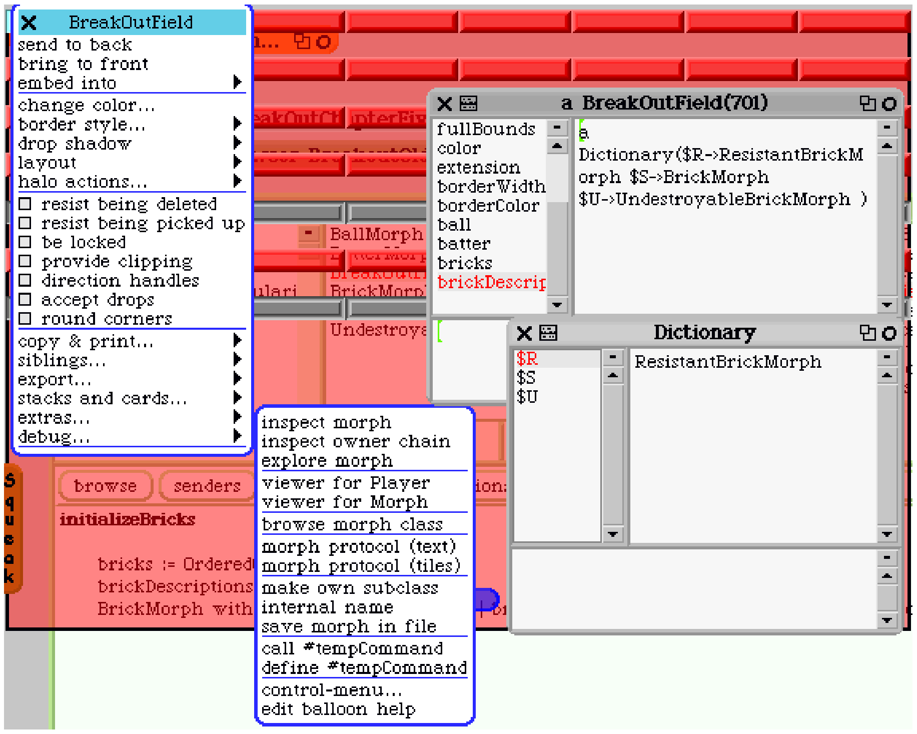
```
BorderedMorph subclass: #BreakOutField
   instanceVariableNames: 'ball batter bricks brickDescriptions '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

We then modify the `initializeBricks` method, to initialize the variable `brickDescriptions` with an empty dictionary, then to fill the dictionary with the description characters and the classes. We get all the subclasses of BrickMorph including itself, for each of them we add as association in the dictionary the description character and the class is it associated with (see method 5.20). If you want to check what has been put inside the dictionary, create a breakOut game using the usual expression `BreakOutField newStandAlone openInWorld`, then using the red halo, access the inspect morph menu item and browse the instance variables of the inspected object as shown in the Figure 5.3.

Method 5.20

```
In category bricks
BreakOutField>>initializeBricks

   bricks := OrderedCollection new.
   brickDescriptions := Dictionary new.
   BrickMorph withAllSubclasses do:
      [:aClass | brickDescriptions at: aClass descriptionCharacter put: aClass]
```

Now we can change the method `brickClassFromDescription:` to query the dictionary `brickDescriptions` to get the class associated with the description character as shown in the method 5.21.



```
a
Dictionary($R->ResistantBrickMorph $S->BrickMorph $U->UndestroyableBrickMorph )
```

```
ResistantBrickMorph
```

Figure 5.3: Inspecting the dictionary containing the association between a description character and a brick class.

Method 5.21

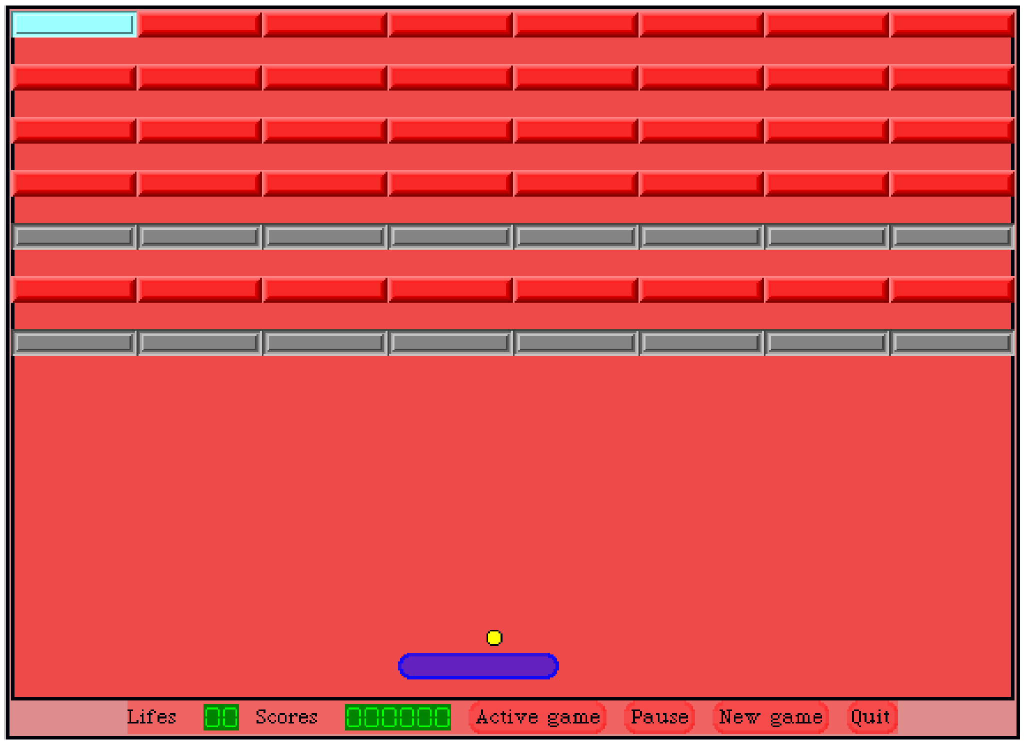
```
In category level handling
BreakOutField>>brickClassFromDescription: aCharacter
   "Return the class associated with the description character"

   ^ brickDescriptions at: aCharacter
```

So we pass from an ad-hoc way of delaing with different bricks to a explicit and clear exchange of messages between two major kind of objects: the bricks and the field.

8 What you learned

6

Score, Lifes, and New Game



Now we would like to add some This will be the pretext to present some important issues related to lowering the coupling between classes and the use of accessor methods. @@ More @@

1 Scores

We would like to add a score to the game. The idea is that eveyrtimes a brick is destroyed or bumped, the score is increased. For this purpose, add the instance variable `score` to the class `BreakOutField`. Do not forget to initialize this new variable to zero. Then define the methods `increaseScoreBy: aNumber` (method 6.1).

Method 6.1

```
In category score
BreakOutField>>increaseScoreBy: aNumber

   score := score + aNumber
```

Now change the method `actionWhenBumpedBy: aBall` to increase the score number of the breakout field.

Method 6.2

```
In category action
BrickMorph>>actionWhenBumpedBy: aBall

   self flash.
   self soundFeedBack.
   self owner increaseScoreBy: 10.
   aBall bounceTopBottom.
   self owner removeBrick: self.
   self delete.
```

We let you do the same for the resistant brick if you want.

2 Game Suspending, Restarting and Reseting

We would like to introduce the fact that the game can be paused, restarted and reset.

Class 6.1

```
BorderedMorph subclass: #BreakOutField
   instanceVariableNames: 'ball batter bricks brickDescriptions paused '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

Method 6.3

```
In category game logic
isPaused

   ^ paused
```

Method 6.4

```
In category game logic
BreakOutField>>pauseGame

   paused := true.
   ball stopStepping.
```

Method 6.5

```
In category game logic
restartPausedGame

   paused := false.
   ball startStepping.
```

Method 6.6

```
In category initialization
BreakOutField>>initializeToStandAlone

   super initializeToStandAlone.
   self bounds: (0@0 corner: BrickMorph defaultBrickSize * self playFieldSize).
   self color: (Color red alpha: 0.5).
   self ball: BallMorph newStandAlone.
   ball positionAtStart.
   self batter: BatterMorph newStandAlone.
   batter positionAtStart.
   self initializeBricks.
   self installLevel: self class level0.
   self pauseGame.
```

The code of the method initializeToStandAlone starts to be messy. Indeed we have different levels of details, sometimes really low level such as `color:` or at the opposite `initializeBricks` which represents several low-level operations. We will refactor it to the same level of abstraction once we have finish the pause and restart.

Now we decide that we can pause and restart the game by pressing space bar. We change the method `keyDown: event` that handles the keyboard event to implement this functionality (see method 6.7).

Method 6.7

```
In category event handling
BreakOutField>>keyDown: evt

   | char |
   char := evt keyCharacter.
   self isPaused
      ifFalse: [char = $n
               ifTrue: [batter moveLeft].
            char = $m
               ifTrue: [batter moveRight]].
   char = Character space
      ifTrue: [self isPaused
               ifTrue: [self restartPausedGame]
               ifFalse: [self pauseGame]]
```

Now that pausing the game works, we take the time to clean the method `BreakOutField»initializeToStandAlone`. You can wonder why this is interesting to change a method that is working. The idea is that if the method starts to be unreadable and its logic complex then we will have problem to use it, extend it to take into account new requirements, the complete system will start to be more fragile and adding new functionality will start to be quite difficult. Therefore the fact that a method works and its quality in terms of readibility and logic are the two faces of the same coin. Having badly written methods working does not prove that you will be able to go fast and change easily your system.

Method 6.8

```
In category initialization
BreakOutField>>initializeToStandAlone

   super initializeToStandAlone.
   self initializeFieldProperties.
   self initializeBallMorph.
   self initializeBatterMorph.
   self setBallAndBatterPositionAtStart.
   self initializeBricks.
   self installLevel: self class level0.
   self initializeGame.
```

Method 6.9

```
In category initialization
BreakOutField>>initializeFieldProperties

   self bounds: (0@0 corner: BrickMorph defaultBrickSize * self playFieldSize).
   self color: (Color red alpha: 0.5).
```

A not really satisfactory version of the method `initializeBatterMorph` is shown in method 6.10.

Method 6.10

```
In category initialization
BreakOutField>>initializeBatterMorph

   self batter: BatterMorph newStandAlone.
```

However, when we analyse the class we see that the method `batter:` is only called by this method, so this is an opportunity to avoid to have small methods implementing separate but related functionality. Therefore we merge them, i.e., remove batter: and change the logic of `initializeBatterMorph` as shown in method 6.11.

Method 6.11

```
In category initialization
BreakOutField>>initializeBatterMorph
   "Create and add a batter as contained by the receiver"

   batter := BatterMorph newStandAlone.
   self addMorph: batter
```

We do the same for the method `ball::` we remove it and include its logic into the method `initializeBallMorph`.

Method 6.12

```
In category initialization
BreakOutField>>initializeBallMorph
   "Create and add a ball as contained by the receiver"

   ball := BallMorph newStandAlone..
   self addMorph: ball.
```

Method 6.13

```
In category initialization
BreakOutField>>setBallAndBatterPositionAtStart

   batter positionAtStart.
   ball positionAtStart.
```

Method 6.14

```
In category initialization
BreakOutField>>initializeGame

   score := 0.
   self pauseGame.
```

3 Lifes

Now we are ready to add multiple lifes to the game. First we add a new instance variable named `lifes` to the class `BreakOutField` and modify the method `initializeGame`

Method 6.15

```
In category initialization
BreakOutField>>initializeGame

   score := 0.
   lifes := 4.
   self pauseGame.
```

Now we can easily write the method `lostABall` as shown in method 6.16.

Method 6.16

```
In category initialization
BreakOutField>>lostABall

   lifes isZero
      ifTrue: [self pauseGame. ^self].
   lifes := lifes - 1.
   self setBallAndBatterPositionAtStart.
   self pauseGame.
```

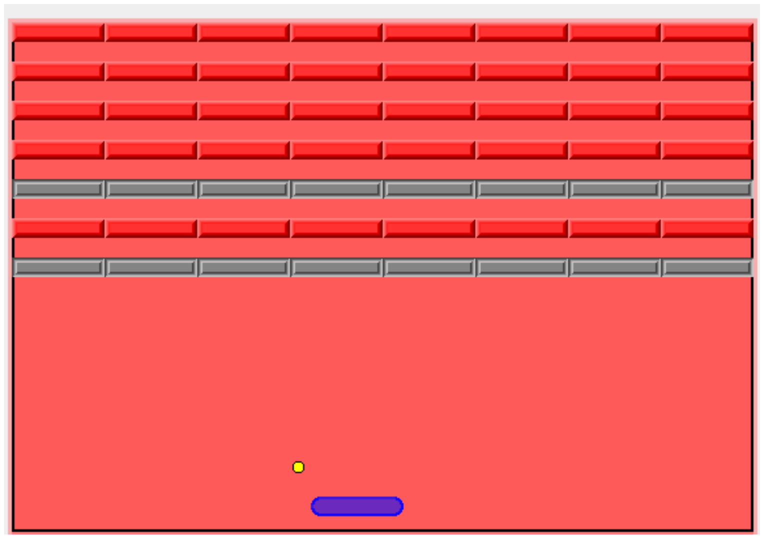


Figure 6.1: The level corresponding to the level description returns by `BreakOutField level0`

Now when we lose a ball, we check whether we can continue to play. If we can we decrement the number of lifes, reset the batter and the ball and pause the game waiting for the user.

4 A Board

Now we would like to see our score and the lifes left. So we create a board game for the breakout. However we do not explain the following code because it would lead us in too much details.

Class 6.2

```
RectangleMorph subclass: #BreakOut
   instanceVariableNames: 'field scoreDisplay lifeDisplay '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

Method 6.17

```
In category initilization
BreakOut>>initializeToStandAlone
   "BreakOut newStandAlone openInWorld"

   super initializeToStandAlone.
   self initializeBreakOutProperties.
   self addMorph: self createBar.
   self initializeField.
   self addMorph: field.
```

Figure 6.2:

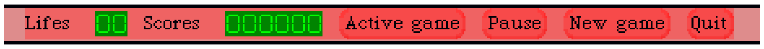


Figure 6.3:

Method 6.18

```
In category initialization
BreakOut>>initializeBreakOutProperties

   self layoutPolicy: TableLayout new.
   self layoutInset: 1@1.
   self listDirection: #topToBottom ; wrapDirection: #none.
   self hResizing: #shrinkWrap; vResizing: #shrinkWrap.
   color := Color red alpha: 0.3.
```

Method 6.19

```
In category initialization
BreakOut>>initializeField

   field := BreakOutField newStandAlone.
```

Method 6.20

```
In category ui elements
BreakOut>>createButtonLabel: label target: target actionSelector: sel

   ^ SimpleButtonMorph new
       target: target;
       label: label;
       actionSelector: sel;
       borderColor: #raised;
       borderWidth: 2;
       color: color.
```

Method 6.21

```
In category ui elements
BreakOut>>createBar

   | bar |
   bar := RectangleMorph new.
   bar color: color.
   bar borderWidth: 0.
   bar layoutPolicy: TableLayout new.
   bar listDirection: #leftToRight.
   bar hResizing: #shrinkWrap ;vResizing: #shrinkWrap.
   bar  cellInset: 10.
   bar addMorph:
          (self createButtonLabel: 'Quit'
               target: self
               actionSelector: #delete).
   bar addMorph:
          (self createButtonLabel: 'New game'
                target: field
               actionSelector: #newGame).
   bar addMorph:
          (self createButtonLabel: 'Pause'
               target: field
               actionSelector: #pauseGame).
   bar addMorph:
          (self createButtonLabel: 'Active game'
               target: field
               actionSelector: #restartPausedGame).
   scoreDisplay := (LedMorph new digits: 6; extent: (6*10@15)).
   bar addMorph: scoreDisplay.
   bar addMorph: (StringMorph contents: 'Scores ').
   lifeDisplay := (LedMorph new digits: 2; extent: (2*10@15)).
   bar addMorph: lifeDisplay.
   bar addMorph: (StringMorph contents: 'Lifes ').
   ^ bar
```

Method 6.22

```
In category game logic
BreakOutField>>newGame

   self setBallAndBatterPositionAtStart.
   self initializeBricks.
   self installLevel: self class level0.
   self initializeGame.
```

We create then duplication between the method `initializeToStandAlone` and `newGame`, so refactor the method `initializeToStandAlone` to call `newGame`.

5 Updating Board Information

Now the final problem we have to solve is that the score displayed is not updated when the score instance variable of theBreakOutField changes.

In fact we could change the BreakOutField class to have a reference to its board and changes directly the value of the score display. However this way of doing think is bad because we would be hardcoding in the code of the BreakOutField information about other objects that are not related to it. For example, we could build several different boards and we do not want to change the breakOutField class every times. The BreakOutField class works well completely in a autonomous manner from the board.

In fact what we need is a way for the board to be aware that some elements of the breakoutField change. In a similar fashion, when we register to a online publishing house, the house does not have to change, it just notifies us when something new is available, then we go to read it if we want. Squeak offers the following solution. The BreakOutField class should trigger events telling which parts of his internal state changes, then the board or any other objects interested register some interests into the events that it needs and performs certain actions if necessary. This way we decouple the breakOutField class from the BreakOut class. The BreakOutField does not have to know or to take care that the board exists. We could have multiple boards presenting different information at the same time without having to change the breakOutField class.

Method 6.23

```
In category initialization
BreakOut>>initializeField

   field := BreakOutField newStandAlone.
   field when: #lifesChanged send: #updateLifes to: self.
   field when: #scoreChanged send: #updateScore to: self.
```

Method 6.24

```
In category update
BreakOut>>updateLifes
   "change the value represented by the ledMorph"

   lifeDisplay value:  field lifes
```

Method 6.25

```
In category update
BreakOut>>updateScore

   scoreDisplay value:  field score
```

Method 6.26

```
In category accessing
BreakOutField>>score

   ^ score
```

Method 6.27

```
In category update
BreakOutField>>lifes

   ^ lifes
```

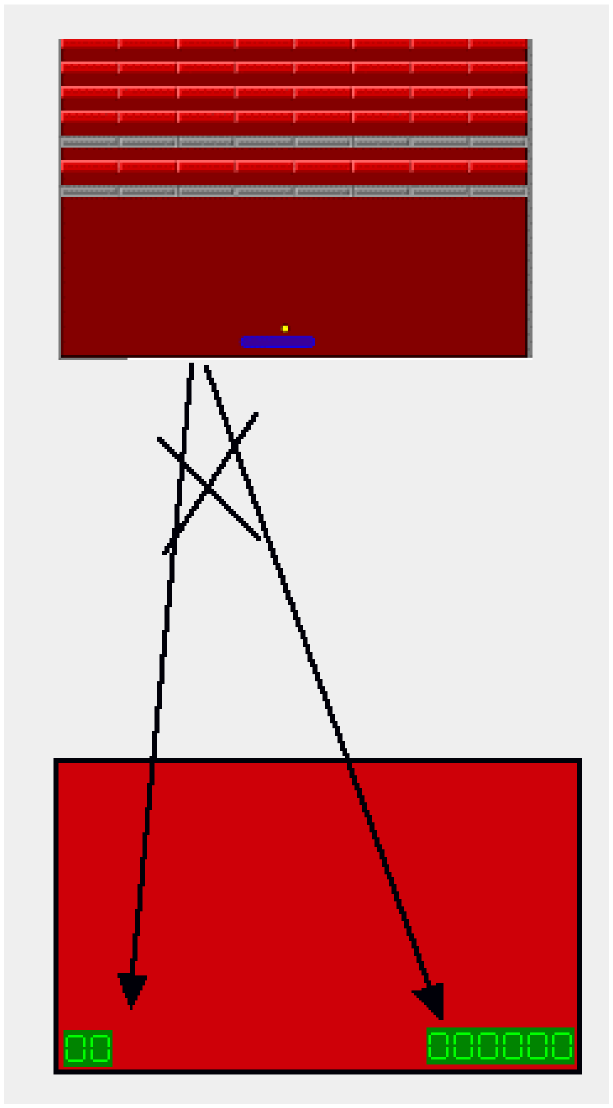


Figure 6.4: Directly referencing the UI elements from the BrekaOutField is bad design because we cannot reuse it with another one without modifying it.

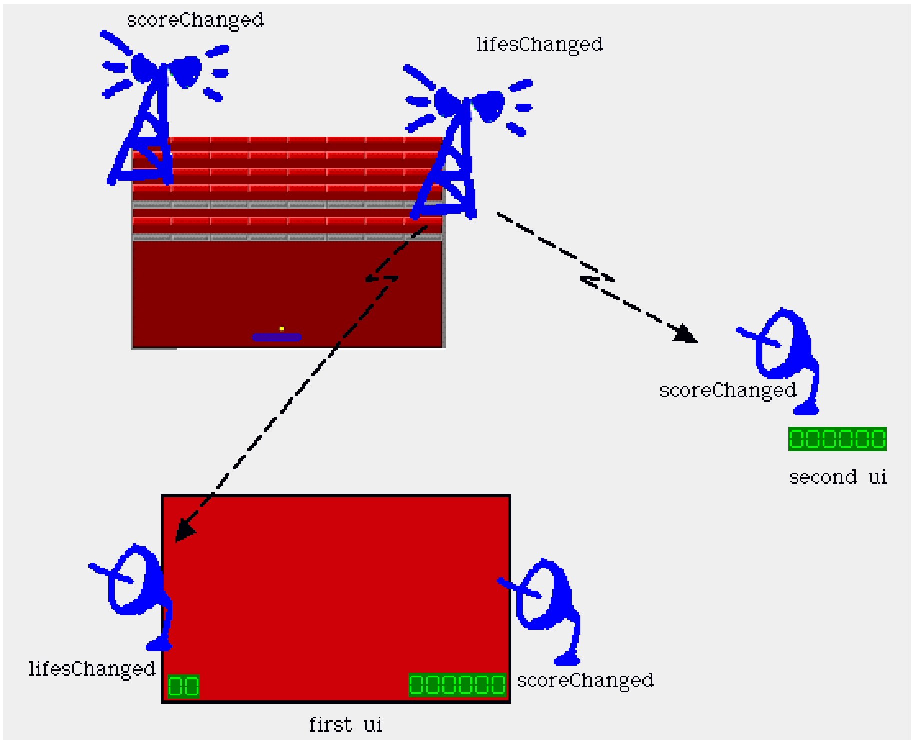


Figure 6.5: Based on events the BreakOufField can have multiple and different UI without the need to tmodify it.

Method 6.28

```
In category score
BreakOutField>>increaseScoreBy: aNumber

   score :=  score + aNumber.
   self trigger: #scoreChanged.
```

Method 6.29

```
In category score
BreakOutField>>score: aNumber

   score := aNumber.
   self trigger: #scoreChanged.
```

Method 6.30

```
In category initialization
BreakOutField>>initializeGame

   score := 0.
   lifes := 4.
   self trigger: #lifeChanged.
   self pauseGame.
```

Method 6.31

```
In category game logic
BreakOutField>>lostABall

   lifes isZero
      ifTrue: [self pauseGame. ^self].
   lifes := lifes - 1.
   self trigger: #lifesChanged.
   self setBallAndBatterPositionAtStart.
   self pauseGame.
```

The final small problem we have is that the field is created, the event `lifesChanged` is triggered but the board does not really exist yet so the life number is not correctly displayed. Therefore once its creation ends, we explicitly tell it to update the life number as show below.

Method 6.32

```
In category initialization
BreakOut>>initializeField

   field := BreakOutField newStandAlone.
   field when: #lifesChanged send: #updateLifes to: self.
   field when: #scoreChanged send: #updateScore to: self.
   self updateLifes
```

6 About Encapsulation and Accessors

To be sure that the number of lifes represented in the break out board are always a correct representation of the number of lifes currently in the game, we have to find all the places in the class `BreakOutField` that change the value of the instance variable `lifes`. Missing only one of those would lead to inconsistent states. The methods `lostABall` (see method 6.31) and `initializeGame` (see method 6.30) show this problem. You could think that as it is working this is ok. The problem is that if we add bonuses that give life as we will do it in the future, we will have again to keep in mind that we have to trigger an event, else the graphical interface could be in an inconsistent state. Again having a duplicated logic is the sign that there is potentially a problem.

The solution is to have only one point where the `lifes` instance variable value is changed and to trigger the event from this place. For that purpose we define the method `lifes: anInteger` as shown in method 6.33.

Method 6.33

```
In category private
BreakOutField>>lifes: anInteger

   lifes := anInteger.
   self trigger: #lifesChanged.
```

Then we change the method `initializeGame` and `lostABall` accordingly (see method 6.34 and method 6.35).

Method 6.34

```
In category initialization
BreakOutField>>initializeGame

   score := 0.
   self lifes: 4.
   self pauseGame.
```

Method 6.35

```
In category game logic
BreakOutField>>lostABall

   lifes isZero
      ifTrue: [self pauseGame. ^self].
   self lifes: lifes - 1.
   self setBallAndBatterPositionAtStart.
   self pauseGame.
```

As we presented in the chapter **??** presenting objects, an object is responsible of the data it contains. It can specifies various behavior in the way it wants. It hides its internal representation from clients so that he can change it without impacting them. Clients should only rely on the interface provided by the object. This property is called data encapsulation. Now in Smalltalk all the methods are public, this means that clients can invoke any methods the class implements. There is no way that we can forbid it. Therefore, when we define a method that simply provides access or change the value of an instance variables, usually called *accessor methods*, clients can completely breaks the encapsulation or internal logic of an object. The solution in Smalltalk is to rely on conventions to know whether a client should or not used methods. We define the method `lifes: anInteger` in the private protocol or method categories to indicate that clients should not use this method.

In fact using accessors or not was a big debate, most of the Smalltalk programmers use accessors. In this book, we took the decision not to use them, because we want to promote the idea of encapsulation and we wanted to avoid methods doing nearly nothing. Kent Beck in his excellent book Best Smalltalk Practices [] provides really useful discussions about the use of accessors. Now when using accessors two points have to be taken into account: first this is not because accessors are methods that they do not break encapsulation, so you should take care when you are invoking accessor methods of a class and especially multiple of them in sequence. Expressions such as `anObject address street number` are a real problem because if one of the class `Person`, `Address`, `Street` changes the client code breaks. In chapter **??** we present the Law of Demeter that helps designing decoupled classes. The idea is to avoid sending messages to result of messages. Second if you use accessors you should use them consistently for all the instance variables of a class and never mixed them with direct access use. To that regard the method method 6.35 may shoke purists because it mixes direct instance variable access and accessors. However here we follow our non use of accessor and only call the method `lifes:` that is more than an accessor.

7 Lessons Learnt

Duplicated code or logic is always the symptom of a potential problems

Accessors do not

decoupling classes: events...

7 Managing Multiple Levels

1 Collecting Levels

In the chapter **??** we show how we represent a level as a level description. We define a class method `level0` that was returning a level description that was used to create effectively the bricks of a level. We use now the same technique to define all the level of a game. We create several class methods that represent all the levels of the game. The method 7.1 shows how another level is defined and returned by the method `level1`. To be able to automatically identify all the levels defined, we follow the conventions that a level is always associated with a method starting with the word `level`.

Method 7.1

```
In category levels
BreakOutField class>>level1

^
'SSSSSSSS
SXXXXXXX
SEEEEEEE
SEEEEEEE
SSSSSSSS
SXXXXXXX
SEEEEEEE
SEEEEEEE
SESEESSS
SESEESES
SESEESES
SESEESES
SESSESSS
'
```

Once the level description are defined and associated with methods we can get them. The idea is that we will collect all the class methods that start with the word 'level', then we will invoke these and keep the returned level descriptions. The scripts 7.1 and 7.3 shows several operations that help us to gather all the defined levels. The method `selectors` returns all the methods defined by a class.

Script 7.1 (*Accessing methods)*

```
BreakOutField selectors
-Print It-> a Set(#handlesKeyboard: #score #mouseLeave: #newGame
#setBallAndBatterPositionAtStart  #isPaused
#initializeBallMorph #brickClassFromDescription: #initializeGame
#removeBrick: #brickLine: #restartPausedGame #installLevel:
#brickDescriptionAtColumn:row: #mouseEnter: #handlesMouseOver:
#addBrick: #placeBrick:atBrickCoordinate: #playFieldSize
#lostABall #increaseScoreBy: #initializeToStandAlone #resistantBrickLine:
#ifBallNotBouncedOnWall: #lifes: #positionBrick:atBrickCoordinate:
#initializeBricks #initializeFieldProperties #lifes #initializeBatterMorph
#keyDown: #brickContainingOrNil: #pauseGame #moveBall)

BreakOutField class selectors
-Print It-> a Set(#level0 #maximumRowNumber #level1 #gatherAllDefinedLevels
#maximumColumnNumber #afterChapterOneSectionTwo #DoIt
 #generateEmptyTemplate #playFieldSize #level3)
```

Script 7.2 (*Sorting 'level' methods)*

```
BreakOutField class selectors
   select: [:each | 'level*' match: each asString]
-Print It-> a Set(#level1 #level0 #level3)

(BreakOutField class selectors
   select: [:each | 'level*' match: each asString]) asSortedCollection
-Print It-> a SortedCollection(#level0 #level1 #level3)
```

Now that we have a way to know all the methods returning a level description we have to execute them to effectively get the returned level descriptions. Smalltalk offers a way to invoke method explicitly using the method `perform: aSymbol` where aSymbol is the method selector of the method been invoked.

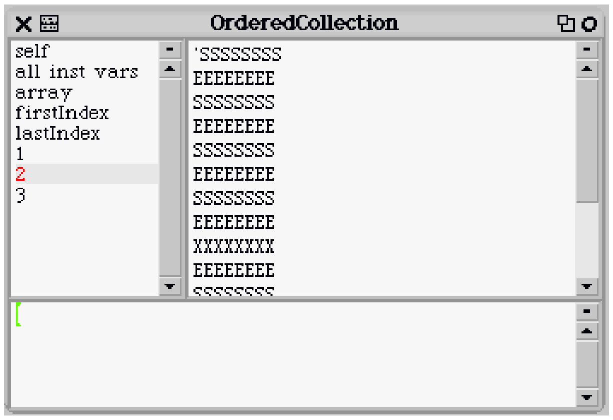
Script 7.3 (*Examples of* `perform:`*)*

```
BreakOutField level0.
"is equivalent to"
BreakOutField perform: #level0

1 + 2
"is equivalent to"
1 perform: #+ with: 2
```

Now we can combine the two functionality to write the method `gatherAllDefinedLevels` as shown in method 7.2. We use the method `collect:` that execute the block specified as argument on all the elements of the receiver and returns a collection containing all the results.

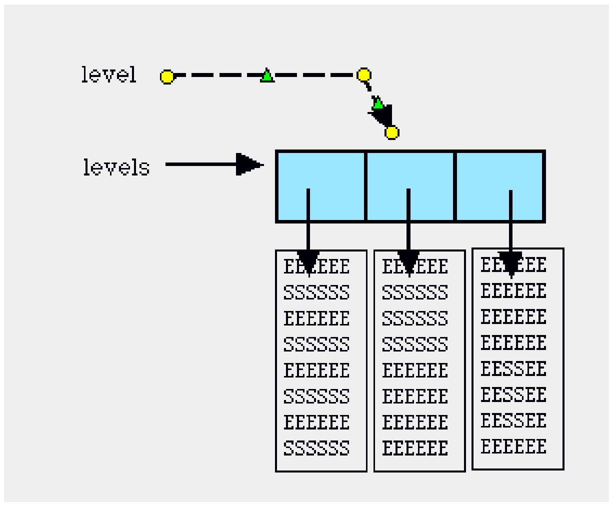
Method 7.2

```
In category level management
BreakOutField class>>gatherAllDefinedLevels

   | sortedSelector |
   sortedSelector := (self class selectors
                       select: [:each | 'level*' match: each asString])
                              asSortedCollection.
   ^ sortedSelector collect: [:levelSel | self perform: levelSel]
```



```
'SSSSSSSS
EEEEEEEE
SSSSSSSS
EEEEEEEE
SSSSSSSS
EEEEEEEE
SSSSSSSS
EEEEEEEE
XXXXXXXX
EEEEEEEE
SSSSSSSS
```

Figure 7.1: Inspecting the results of `BreakOutField gatherAllDefinedLevels`



```
EEEEEE  EEEEEE  EEEEEE
SSSSSS  SSSSSS  EEEEEE
EEEEEE  SSSSSS  EEEEEE
SSSSSS  SSSSSS  EEEEEE
EEEEEE  EEEEEE  EESSEE
SSSSSS  EEEEEE  EESSEE
EEEEEE  EEEEEE  EESSEE
SSSSSS  EEEEEE  EEEEEE
```

Figure 7.2:

2 Managing Levels

Now we are ready to have a breakout with multiple levels. The idea is that we use a collection holding all the level descriptions, and we use a number to represent the current level. When we change level we increment the level number and install the level description corresponding. For this purpose, we need to add two new instance variables to the class `BreakOutField` to represent the current level and the levels available.

Class 7.1

```
BorderedMorph subclass: #BreakOutField
   instanceVariableNames: 'ball batter bricks brickDescriptions score paused
                           lifes level levels '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BreakOut'
```

Now we are ready to define how the levels are used in the game and how we pass from one level to the next one. We have to collect all the defined levels, initialize the game, and install the first level by creating all the bricks as specified by a level description. The first thing to do is to collect all the defined levels and initialize the instance variables `levels` to refer to the collection of level description. We change the method `initializeToStandAlone` (see method 7.3) to invoke a new method called `installLevels` (see method **??** which will have this responsibility. Note again that the method `initializeToStandAlone` is composed conceptually by method having the same level of abstractions. This helps the reading and understanding of its functionality. Imagine if all the code of each method would be just included direclty in the method `initializeToStandAlone`. This would lead to a really complex method that would be difficult to change and extend.

Method 7.3

```
In category initialization
BreakOut>>initializeToStandAlone

   super initializeToStandAlone.
   self initializeFieldProperties.
   self initializeBallMorph.
   self initializeBatterMorph.
   self setBallAndBatterPositionAtStart.
   self initializeBricks.
   self installLevels.
   self initializeGame.
```

Method 7.4

```
In category initialization
BreakOut>>installLevels

   levels := self class gatherAllDefinedLevels.
```

Now we have to modify the method `initializeGame` to specify that the game starts at the first level and (method 7.5) install the current level.

Method 7.5

```
In category initialization
BreakOut>>initializeGame

   score := 0.
   self lifes: 4.
   level := 1.
   self installCurrentLevel.
   self pauseGame.
```

Installing a level is just invoking the method `installLevel: aLevelDescription` that we defined in the chapter **??** with the level description corresponding with the current level number.

Method 7.6

```
In category level handling
installCurrentLevel

     self installLevel: (levels at: level).
```

The method `isLastLevel` just uses the fact that the length of the level description collection represent the total number of levels. Therefore we are at the last level if the length is equal to the level number.

Method 7.7

```
In category level handling
BreakOut>>isLastLevel
   "Return whether the current level is the last one defined"

   ^ levels size = level
```

Next Level

From now you can start playing and the system should display the first level you define on the class side. The second step is to make sure that when the last brick is destroyed, the game passes to the next level. For that we have to change the method `moveBall` (see method 7.8) so that it checks when a brick is touched if it was the last one.

Method 7.8

```
In category game logic
BreakOut>>moveBall

   |  b nextPosition |
   nextPosition := ball nextPosition.
    (self ifBallNotBouncedOnWall: nextPosition)
      ifTrue:
         [(batter containsPoint: nextPosition)
            ifTrue:
               [batter touchedBy: ball.
               ^ self].
         b := (self brickContainingOrNil: nextPosition).
         b ifNotNil:
            [b actionWhenBumpedBy: ball].
         self isLevelTerminated
            ifTrue: [self installNextLevel].
         ball moveToNextPosition]
```

As every times a brick is destroyed it is removed from the `bricks` instance variable collection, we can consider that a level is terminated when this instance variable holds an empty collection (see method 7.9).

Method 7.9

```
In category level handling
BreakOut>>isLevelTerminated

   ^ bricks isEmpty
```

Installing the next level is then pausing the game, increasing the level number, installing the level, and resetting the ball and batter position. In method 7.10 we check that if the level was the last one we restart at the beginning.

Method 7.10

```
In category level handling
BreakOut>>installNextLevel

   self pauseGame.
   self isLastLevel
      ifTrue: [level := 1]
      ifFalse: [level := level + 1].
   self installCurrentLevel.
   self setBallAndBatterPositionAtStart
```

Now when you finish a level, you pass to the next one and when you are at the end of the last level you restart at the beginning.

3 The Case of Undestroyable Bricks

In fact we forgot the case of the bricks that we cannot destroy. Right now our solution does not work because as soon as a level contains one of such a brick, the method `isLevelTerminated` described above always returns false. In this section we propose to solve that problem. There are several ways to solve this problem. Before reading our approach think about one or two ways of solving it and evaluate the pros and cons in terms of complexity, *i.e.,* how much information you should keep synchronized, or in terms of speed.

Our idea is that at the installation of a level we count all the bricks that have to be destroyed and then every time a brick is destroyed we decrement the number we originally calculated. Therefore add a new instance variable called `destroyableBrickNumber` to the class `BreakOutField`. The number of destroyable bricks should always be initialized when a new level is installed, therefore we obvious place to compute it is the method `installCurrentLevel` (see method 7.11).

Method 7.11

```
In category initialization
(Bad design)
BreakOutField>>installCurrentLevel

   self installLevel: (levels at: level).
   destroyableBrickNumber := (bricks reject: [:each | each class = UndestroyableBrickMorph]) size
```

We then have to change the implementation of the method `isLevelTerminated` to check whether all the bricks that should be destroyed are effectively.

Method 7.12

```
In category level handling
BreakOutField>>isLevelTerminated

   ^ destroyableBrickNumber isZero
```

The implementation of the method `installCurrentLevel` (see method 7.11) is not really good. Indeed we hardcode the name of a class directly in the method. This means that we as a client of a brick takes decision while the brick itself should be responsible of such decision. Here the decision taken by the client, the class `BreakOutField` is simple but we can easily imagine cases where the logic of the decision would imply checking different conditions. Therefore the tests could depend on several classes. In addition referring explicitly to class names is a bad practice because if we change the name of the class

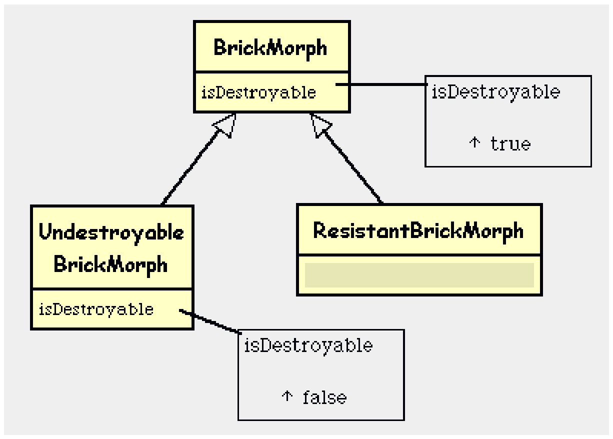


Figure 7.3:

we have to change all the references too.

The solution is to let the brick tell us whether they are destroyable or not (see method 7.13). To do that we define the method `isDestroyable` which on the class `BrickMorph` returns `true`. This means that all the subclasses of BrickMorph will be destroyable except if they redefine this method. On the class `UndestroyableBrickMorph` the method is redefined to return the value `false` indicating that the bricks of this class and its subclasses are not destroyable.

Method 7.13

```
In category initialization
(Not that bad Design )
BreakOutField>>installCurrentLevel

   self installLevel: (levels at: level).
   destroyableBrickNumber := (bricks reject: [:each | each isDestroyable]) size
```

Method 7.14

```
In category testing
BrickMorph>>isDestroyable

   ^ true
```

Method 7.15

```
In category testing
UndestroyableBrickMorph>>isDestroyable

   ^ false
```

Now the solution we propose in method 7.13 is not really good because we create first a collection with all the bricks that should be destroyed but we do not do anything with it. So instead of collecting objects counting them is enough. Indeed collecting objects can be costly. The solution is to just count the bricks. The method 7.16 shows one way to do it.

Method 7.16

```
BreakOutField>>installCurrentLevel


   |sum|
   self installLevel: (levels at: level).
   sum := 0.
   destroyableBrickNumber := bricks do: [:each | sum := sum + (each isDestroyable ifTrue: [1] ifF
```

Smalltalk collections offers a better way to do the same using the `inject:into:` method. However, this method lead to code that may be difficult to read, therefore if you do not feel confortable with it do not use it. The script 7.4 shows some examples of use of `inject:into:`. The idea is that the first argument is the starting value of the accumulator named `sum` or `prod` and that is the first argument of the block passed as second argument. The second argument of the block (the second argument) is the element of the collection.

Script 7.4 (*Using* `inject:into:`*)*

```
#($a $b $a $b $a $b) inject: 0 into: [:sum :each  | sum + (each = $a ifTrue: [1] ifFalse: [0])].
    -Print It-> 3


#(1 2 3 4 5 6 7) inject: 0 into: [:sum :each | sum + each ]
    -Print It-> 28

#(1 2 3 4 5 6 7) inject: 1 into: [:prod :each | prod * each ]
    -Print It-> 5040
```

Method 7.17

```
BreakOutField>>installCurrentLevel

   self installLevel: (levels at: level).
   destroyableBrickNumber := bricks inject: 0 into: [:sum :each | sum + (each isDestroyable ifTru
```

Try and you will discover that we have still two problems. First we forgot to change the number of bricks left to destroy.

Method 7.18

```
BreakOutField>>removeBrick: aBrickMorph

  bricks remove: aBrickMorph.
  destroyableBrickNumber := destroyableBrickNumber - 1.
```

Then we forgot to remove the undestroyable bricks once the level is terminated. The method 7.19

Method 7.19

```
BreakOutField>>installNextLevel

   self pauseGame.
   self isLastLevel
      ifTrue: [level := 1]
      ifFalse: [level := level + 1].
   bricks do: [:each | each delete].
   self installCurrentLevel.
   self setBallAndBatterPositionAtStart
```

We let you change the `BreakOut` to reflect the current level using the event-based techniques we presented in the previous chapter.

4 Lessons Learnt

You may wonder why we did not take into account the problem before. We did this on purpose to show you how a good method decomposition helps to change a system. An application that cannot change is a dead or useless application. Therefore you should always think that you system can change. However you should not try too much to see in which directions it will change. Having a clean code with short methods having clear responsibility and no duplication is the minimal way of been prepared to change. Another way is to write tests for each important responsibility you implement. We do not present this aspect in this book except in the chapter for lack of time and space.

level or levelNumber, destroyableBricks or destroyableBrickNumber

8
Bonuses

Bonuses are extra functions that we gain during the game. We decide to implement a simple bonus system. We attach a bonus to a brick and when this brick is destroyed the bonus it contained starts its effect, after a certain moment the bonus effect vanished if necessary.

Again there are several ways to introduce bonuses in the game, one would be to use morphs because they are objects having the notion of time passing via their `step` method. However the main point of a morph is that it is a graphical objects so we chose not to implement as a subclass of Morph. We let you as an exercise to convert our solution to use morph and for example change so that destroying a brick would produce a bonus falling down the screen and that its effect would be effective only when we would succeed to get the falling bonus.

@@ need a picture with a large batter@@

1 Bonus

A bonus is an object that may have a certain *duration* once actived. Therefore define the class `Bonus` with two instance variables `duration` and `timeLeft`. As we do not plan to have graphical representation for the bonuses, subclass the class `Bonus` from the class `Object`.

A bonus has a given duration that is initialized is the method `initialize` (see method 8.1).

Method 8.1

```
In category initialize
Bonus>>initialize
   "initialize is not defined on Object so we do not do a super initialize"

   duration := 10.
```

@@need a sequence diagram here @@

Once activated the effect of a bonus may last a certain period of time. We need a way to start it, notify it that the time is passing and terminate the effect. Therefore we define the method `start: aField`, the method `oneStep: aField`, and the method `end: aField`. We pass as argument the game in which the bonus is activated to have the possibility to change all the aspects of the game.

The method `start: aField` shown in method 8.2 initializes the `timeLeft` value. This value is then decreased by the method `oneStep: aField` that will be called a regular interval by the field (see method 8.3). When the timeLeft is zero it calls the method `end: aField` (see method 8.3) which has the responsibility to notify the field that the bonus is terminated.

Method 8.2

```
In category actions
Bonus>>start: aField

   timeLeft := duration.
```

Method 8.3

```
In category actions
Bonus>>oneStep: aField

   timeLeft := timeLeft -1.
   timeLeft isZero
      ifTrue: [self end: aField]
```

Method 8.4

```
In category actions
Bonus>>end: aField

   aField bonusTerminated: self
```

Hot Pepper

Per default in Squeak the method `new` does not call automatically the method `initialize`. The method `initialize` is not defined on the class `Object`. Therefore if you define a method `initialize` as shown hereafter you get an errror because the super initialize does not find any method in the class `Object`. We provide a extension of the system that make the method `new` calling the method `initialize` and we define a method `initialize` on the class `Object`. With this extension loaded, you can define a method `initialize` for your own class as follow, where the expression `super initialize` is only necessary if you need to invoke the `initialize` method of superclasses.

Method 8.5

```
MyClass>>initialize

   super initialize.
   my init stuff
```

Hot Pepper End

If you did not load the extension automatic initialize we proposed, you should define the class method `new` to invoke the method `initialize` on the newly created object.

Method 8.6

```
In category instance creation
Bonus class>>new
   "create an instance, initialize it then return it"

   | inst |
   inst :=  super new.
   inst initialize.
   ^ inst
```

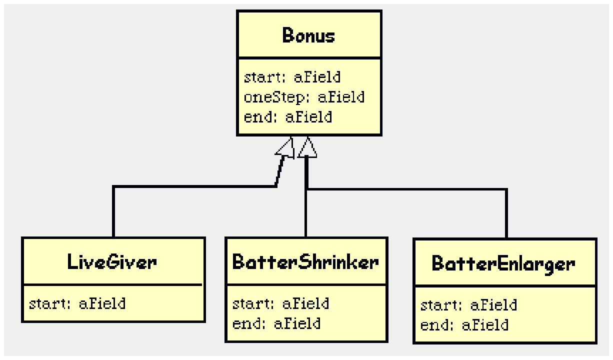


Figure 8.1: Bonuses specialize the methods of the class `Bonus`.

2 Two Bonuses

To specify bonuses now we just have to subclass the class `Bonus` and specialize the methods that control the life time of a bonus (`start:`, `end:`, or `oneStep:`). Now are ready to define the bonuses that shrink and enlarge the batter. Define one subclass of the class `Bonus` named `BatterShrinker`. The principle is the following: redefine the methods `start: aField` and ctend: so that the default behavior is still invoked and then define the bonus specific behavior.

For example for the `BatterShrinker` the method `start:` is defined as shown in method 8.12, the default behavior defined on the class `Bonus` is invoked and the batter is shrunk. When the bonus activity period is over, the method `end:` invoke the default behavior and then gives the batter its normal size.

Method 8.7

```
In category actions
BatterShrinker>>start: aField

   super start: aField.
   aField batter shrink
```

Method 8.8

```
In category actions
BatterShrinker>>end: aField

   super end: aField.
   aField batter normalSize
```

We need then to define the methods `shrink`, `scaleBy:`, and `normalSize` on the class `BatterMorph` that we let you discover.

Method 8.9

```
In category transformation
BatterMorph>>shrink

   self extent > (self owner defaultBatterSize / 2)
      ifTrue: [ self scaleBy: 0.5@1 ]
```

Method 8.10

```
In category transformation
BatterMorph>>scaleBy: aPoint

    self extent: (self extent scaleBy: aPoint)
```

Method 8.11

```
In category transformation
BatterMorph>>normalSize

   self extent: self owner defaultBatterSize
```

For the bonus `BatterEnlarger` we use the same strategy. Here we just give you the code.

Method 8.12

```
In category actions
BatterEnlarger>>start: aField

   | batter |
   super start: aField.
   batter := aField batter.
   batter enlarge.
   batter right > aField right
      ifTrue: [batter position: (aField right - batter extent x)@batter position y]
```

Method 8.13

```
In category actions
BatterEnlarger>>end: aField

   super end: aField.
   aField batter normalSize
```

Method 8.14

```
In category transformation
BatterMorph>>enlarge

   self extent <= self owner defaultBatterSize
      ifTrue: [ self scaleBy: 2@1]
```

We let you as an exercise to implement the bonus that gives extra lifes. Note that this bonus does not use a duration so do not forget to change its initialization. It could use the following method defined on the class `BreakOutField`

Method 8.15

```
In category bonus
BreakOutField>>increaseLife

   self lifes: self lifes + 1
```

Now we have defined bonuses but they are not managed by the field.

3 Managing Active Bonuses

In our solution the field acts as a clock for the bonuses, it has the responsibility to ask every bonus to do one step in this behavior. Therefore the field should know the active bonuses. Add the instance variable `activeBonuses` to the class `BreakOutField`. Define the method `initializeActiveBonuses` (see method 8.16) and modify the method `initialize` to invoke it.

Method 8.16

```
In category initialize
BreakOutField>>initializeActiveBonuses

   activeBonuses := OrderedCollection new
```

Method 8.17

```
In category initialize
BreakOutField>>initializeToStandAlone

   super initializeToStandAlone.
   self initializeFieldProperties.
   self initializeBallMorph.
   self initializeBatterMorph.
   self setBallAndBatterPositionAtStart.
   self initializeBricks.
   self initializeActiveBonuses.
   self installLevels.
   self initializeGame.
```

Now we define the method `addActiveBonus: aBonus` (method 8.18). This method will be called when a brick containing a bonus will be destroyed. It adds the bonus to the list of the active bonuses and send the message `start:` to the bonus.

Method 8.18

```
In category bonuses
BreakOutField>>addActiveBonus: aBonus

   activeBonuses add: aBonus.
   aBonus start: self
```

The method `step` of the `BreakOutField` class is invoked by the system regularly. This method askes all the bonuses to execute one step by sending them the message `oneStep:`.

Method 8.19

```
In category bonuses
BreakOutField>>step

   activeBonuses do: [:each | each oneStep: self].
```

Finally the method `bonusTerminated: aBonus` that is invoked when a bonus terminated its activity period, just remove the bonus from the list of active bonuses (see method 8.20).

Method 8.20

```
In category bonuses
BreakOutField>>bonusTerminated: aBonus

   activeBonuses remove: aBonus
```

4 Enhancing Brick with Bonus

Now a brick should have the possibility to have an hidden bonus. Add the instance variable `bonus` to the class `BrickMorph` and define the following methods

Method 8.21

```
In category bonus
BrickMorph>>hasBonus

   ^ bonus isNil not
```

Method 8.22

```
In category bonus
BrickMorph>>bonus: aBonus

   bonus := aBonus
```

Method 8.23

```
In category bonus
BrickMorph>>bonus

   ^ bonus
```

5 Dispatching Bonuses

Now we are ready to distribute the bonuses in the bricks. We could have fixed the number of bonuses dispatched in the bricks in the `BreakOutField` but we can do better. We let the bonus decide their percentage of distribution among the brick. For that reason, we define on each bonus class a *class* method named `percentage` that returns the percentage of the bonus inside the bricks of a level. Note that the default percentage defined on the class `Bonus` is 20%, subclasses only need to define the method `percentage` if they want a different percentage.

Method 8.24

```
In category properties
Bonus class>>percentage

   ^ 0.2
```

Method 8.25

```
In category properties
BatterShrinker class>>percentage

   ^ 0.05
```

Method 8.26

```
In category properties
LiveGiver class>>percentage

   ^ 0.05
```

Now every time we install a new level, we just have to dispatch bonuses inside the destroyable bricks of the level as shown by the method method 8.27, method 8.28, method 8.29, and method 8.30. Note that we change the definition of the method `installCurrentLevel` and create the method `destroyableBricks` because we need it in another place (see method 8.30).

Method 8.27

```
In category initialize
BreakOutField>>installCurrentLevel

   self installLevel: (levels at: level).
   destroyableBrickNumber := self destroyableBricks size.
   self dispatchBonuses.
```

Method 8.28

```
In category bonus
BreakOutField>>destroyableBricks

   ^ bricks select: [:each | each isDestroyable]
```

The method `dispatchBonus` method 8.29 collect all the subclasses of the class `Bonus` and for each of them the bonuses are dispatched as shown by the method `dispatchBonus:`. The percentage of the bonus is converted into the number of bricks in the current level that should have a bonus of this kind. Then we pick a brick at random and associate it a bonus. We repeat this operation the right number of times. To help debugging this method we define the method `revealBonuses` that changes the color of the bricks having a bonus (see method 8.31).

Method 8.29

```
In category bonuses
BreakOutField>>dispatchBonuses

   Bonus allSubclasses do: [:aClass | self dispatchBonus: aClass]
```

Method 8.30

```
In category bonuses
BreakOutField>>dispatchBonus: aBonusClass

   | destroyable bonusPercentage |
   bonusPercentage := aBonusClass percentage.
   destroyable := self destroyableBricks.
   (destroyable size * bonusPercentage) rounded
      timesRepeat: [(destroyable at: destroyable size atRandom) bonus: aBonusClass new].
   self revealBonuses.
```

Method 8.31

```
In category bonus
BreakOutField>>revealBonuses

   self destroyableBricks do: [:each | each hasBonus ifTrue: [each color: Color green]]
```

Now you should be able to play and have bonuses. This terminates the project on the breakOut.

6 Lessons Learnt

minimize duplicated code, event for minimizing coupling, change in collaboration,