Building a memory game with Bloc

Andrei Chiș, Stéphane Ducasse and Aliaksei Syrel

November 6, 2023

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Objectives of this book

Now that Bloc's design is stabilizing, this book presents its first tutorial. Future changes should mostly be minor (e.g. method renaming).

In this tutorial, you will build a memory game. Given a provided model of a game, we will focus on creating a UI for it.

1.1 Memory game

Let us have a look at what we want to build with you: a simple Memory game. In a memory game, players need to find pairs of similar cards. In each round, a player turns over two cards at a time. If the two cards show the same symbol they are removed and the player gets a point. If not, they are both returned facedown.

For example, Figure 1-1 shows the game after the first selection of two cards. Facedown cards are represented with a cross and turned cards show their number. Figure 1-2 shows the same game after a few rounds. While this game can be played by multiple players, in this tutorial we will build a game with just one player.

Our goal is to have a functional game with a model and a simple graphical user interface. In the end, the following code should be able to build, initialize, and launch the game:

```
game := MgdGameModel new initializeForSymbols: '12345678'.
grid := MgdGameElement new.
grid memoryGame: game.

space := BlSpace new.
space extent: 420@420.
```

Figure 1-1 The game after the player has selected two cards: facedown cards are represented with a cross and turned cards with their number.

```
space root addChild: grid.
space show
```

- First, we create a game model and ask you to associate the numbers from 1 to 8 with the cards. By default, a game model has a size of 4 by 4, which fits eight pairs of numbered cards.
- Second, we create a graphical game element.
- Third, we assign the model of the game to the UI.
- Finally, we create and display a graphical space in which we place the game UI.

1.2 Getting started

This tutorial is for Pharo 11.0 (`https://pharo.org/download`) running on the latest compatible Virtual machine. You can get them at the following address: `http://www.pharo.org/`

To load Bloc, execute the following snippet in a Pharo Playground:

Figure 1-2 Another state of the memory game after the player has correctly matched two pairs.

```
[ Metacello new
  baseline: 'Bloc';
  repository: 'github://pharo-graphics/Bloc:dev-1.0/src';
  onConflictUseIncoming;
  ignoreImage;
  load ]
    on: MCMergeOrLoadWarning
    do: [ :warning | warning load ]
```

1.3 Loading the Memory Game

To make the demo easier to follow and help you if you get lost, we already made a full implementation of the game. You can load it using the following code:

```
Metacello new
    baseline: 'BlocTutorials';
    repository: 'github://pharo-graphics/Tutorials/src';
    load
```

After you have loaded the BlocTutorials project, you will get two new packages: `Bloc-MemoryGame` and `Bloc-MemoryGame-Demo`. `Bloc-MemoryGame` contains the full implementation of the game. Just browse to the class side of `MgExamples` and click on the green triangle next to the `open` method to start

the game. `Bloc-MemoryGame-Demo` contains a skeleton of the game that we will use in this tutorial.

CHAPTER 2

Game model insights

Before starting with the actual graphical elements, we first need a model for our game. This game model will be used as the Model in the typical Model View architecture. On the one hand, the model does not communicate directly with the graphical elements; all communication is done via announcements. On the other hand, the graphic elements are communicating directly with the model.

In the remainder of this chapter, we describe the game model in detail. If you want to move directly to building graphical elements using Bloc, you can find this model in the package `Bloc-MemoryGame-Demo`.

2.1 Reviewing the card model

Let us start with the card model: a card is an object holding a symbol to be displayed, a state representing whether it is flipped or not, and an announcer that emits state changes. This object could also be a subclass of Model which already provides announcer management.

```
Object << #MgdCardModel
  slots: { #symbol . #flipped . #announcer};
  package: 'Bloc-MemoryGame-Demo-Model'
```

After creating the class we add an `initialize` method to set the card as not flipped, together with several accessors:

```
MgdCardModel >> initialize
  super initialize.
  flipped := false
```

```
MgdCardModel >> symbol: aCharacter
  symbol := aCharacter
```

```
MgdCardModel >> symbol
  ^ symbol
```

```
MgdCardModel >> isFlipped
  ^ flipped
```

```
MgdCardModel >> announcer
  ^ announcer ifNil: [ announcer := Announcer new ]
```

2.2 Card simple operations

Next we need two API methods to flip a card and make it disappear when it is no longer needed in the game.

```
MgdCardModel >> flip
  flipped := flipped not.
  self notifyFlipped
```

```
MgdCardModel >> disappear
  self notifyDisappear
```

2.3 Adding notification

The notification is implemented as follows in the `notifyFlipped` and `notifyDisappear` methods. They simply announce events of type `MgdCardFlippedAnnouncement` and `MgdCardDisappearAnnouncement`. The graphical elements will have to register subscriptions to these announcements as we will see later.

```
MgdCardModel >> notifyFlipped
  self announcer announce: MgdCardFlippedAnnouncement new
```

```
MgdCardModel >> notifyDisappear
  self announcer announce: MgdCardDisappearAnnouncement new
```

Here, `MgdCardFlippedAnnouncement` and `MgdCardDisappearAnnouncement` are subclasses of `Announcement`.

```
Announcement << #MgdCardFlippedAnnouncement
  package: 'Bloc-MemoryGame-Demo-Events'
```

```
Announcement << #MgdCardDisappearAnnouncement
  package: 'Bloc-MemoryGame-Demo-Events'
```

We add one final method to print a card in a nicer way and we are done with the card model!

```
MgdCardModel >> printOn: aStream
  aStream
    nextPutAll: 'Card';
    nextPut: Character space;
    nextPut: $(;
    nextPut: self symbol;
    nextPut: $)
```

2.4 Reviewing the game model

The game model is simple: it keeps track of all the available cards and all the cards currently selected by the player.

```
Object << #MgdGameModel
  slots: { #availableCards . #chosenCards};
  package: 'Bloc-MemoryGame-Demo-Model'
```

The `initialize` method sets up two collections for the cards.

```
MgdGameModel >> initialize
  super initialize.
  availableCards := OrderedCollection new.
  chosenCards := OrderedCollection new
```

```
MgdGameModel >> availableCards
  ^ availableCards
```

```
MgdGameModel >> chosenCards
  ^ chosenCards
```

2.5 Grid size and card number

For now, we'll hardcode the size of the grid and the number of cards that need to be matched by a player.

```
MgdGameModel >> gridSize
  "Return grid size, total amount of cards is gridSize^2"
  ^ 4
```

```
MgdGameModel >> matchesCount
  "How many chosen cards should match in order for them to disappear"
  ^ 2
```

```
MgdGameModel >> cardsCount
  "Return how many cards there should be depending on grid size"
  ^ self gridSize * self gridSize
```

2.6 Initialization

To initialize the game with cards, we add an `initializeForSymbols:` method. This method creates a list of cards from a list of characters and shuffles it. We also add an assertion in this method to verify that the caller provided enough characters.

```
MgdGameModel >> initializeForSymbols: characters

  self
    assert: [ characters size = (self cardsCount / self
    matchesCount) ]
    description: [ 'Amount of characters must be equal to possible
    all combinations' ].
  availableCards := (characters asArray collect: [ :aSymbol |
    (1 to: self matchesCount) collect: [ :i |
       MgdCardModel new symbol: aSymbol ] ])
          flattened shuffled asOrderedCollection
```

2.7 Game logic

Next, we need `chooseCard:`, a method that will be called when a user selects a card. This method is actually the most complex method of the model and implements the main logic of the game. First, the method makes sure that the chosen card is not already selected. This could happen if the view uses animations that give the player the chance to click on a card more than once. Next, the card is flipped by sending it the message `flip`. Finally, depending on the actual state of the game, the step is complete and the selected cards are either removed or flipped back.

```
MgdGameModel >> chooseCard: aCard
  (self chosenCards includes: aCard)
    ifTrue: [ ^ self ].
  self chosenCards add: aCard.
  aCard flip.
  self shouldCompleteStep
    ifTrue: [ ^ self completeStep ].
  self shouldResetStep
    ifTrue: [ self resetStep ]
```

The current step is completed if the player selects the right amount of cards and they all show the same symbol. In this case, all selected cards receive the message `disappear` and are removed from the list of selected cards.

```
MgdGameModel >> shouldCompleteStep
  ^ self chosenCards size = self matchesCount
    and: [ self chosenCardMatch ]
```

```
MgdGameModel >> chosenCardMatch
  | firstCard |
  firstCard := self chosenCards first.
  ^ self chosenCards allSatisfy: [ :aCard |
    aCard isFlipped and: [ firstCard symbol = aCard symbol ] ]
```

```
MgdGameModel >> completeStep
  self chosenCards
    do: [ :aCard | aCard disappear ];
    removeAll.
```

The current step should be reset if the player selects a third card. This will happen when a player already selected two cards that do not match and clicks on a third one. In this situation, the two initial cards will be flipped back. The list of selected cards will only contain the third card.

```
MgdGameModel >> shouldResetStep
  ^ self chosenCards size > self matchesCount
```

```
MgdGameModel >> resetStep
  |lastCard|
  lastCard := self chosenCards  last.
  self chosenCards
    allButLastDo: [ :aCard | aCard flip ];
    removeAll;
    add: lastCard
```

2.8 Ready

We are now ready to start building the game view.

CHAPTER 3

Building card graphical elements

In this chapter we will build the visual appearance of the cards step by step. In Bloc, visual objects are called elements, which are usually subclasses of `BlElement`, the inheritance tree root. In subsequent chapters we will do the same for the game and add interaction using event listeners.

3.1 First: the card element

Our graphic element representing a card will be a subclass of the `BlElement` which has a reference to a card model.

```
BlElement << #MgdRawCardElement
  slots: { #card };
  package: 'Bloc-MemoryGame-Demo-Elements'
```

The message `backgroundPaint` will be used later to customize the background of our card element. Let us define a nice color.

```
MgdRawCardElement >> backgroundPaint
  ^ Color lightGray
```

We mentioned the accessors since the setter will be a place to hook registration for the communication between the model and the view.

```
MgdRawCardElement >> card
  ^ card
```

```
MgdRawCardElement >> card: aMgCard
  card := aMgCard
```

We initialize it to get a

```
MgdRawCardElement >> initialize
  super initialize.
  self size: 80 @ 80.
  self background: self backgroundPaint.
  self card: (MgdCardModel new symbol: $a)
```

3.2 Starting to draw a card

In Bloc, BlElements draw themselves onto the integrated canva of the inspector as we inspect them, take a look at our element by executing this:

```
MgdRawCardElement new
```

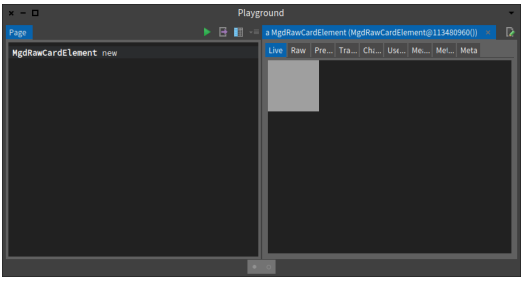

Figure 3-1 A first extremely basic representation of face down card.

3.3 Improving the card visual

Instead of displaying a full rectangle, we want a better visual. Bloc let us decide the geometry we want to give to our elements, it could be a circle, a triangle or a rounded rectangle for example, you can check avaible geometries by looking at subclasses of `BlElementGeometry`.

We can start giving a circle shape to our element, we will need to use the `geometry:` message and give a `BlCircleGeometry` as a parameter.

```
MgdRawCardElement >> initialize

  super initialize.
  self size: 80 @ 80.
  self background: self backgroundPaint.
  self geometry: BlCircleGeometry new.
  self card: (MgdCardModel new symbol: $a)
```

However, we don't want the card to be a circle either. Ideally, it should be a rounded rectangle. So let's first add a helper method that would provide us with a corner radius:

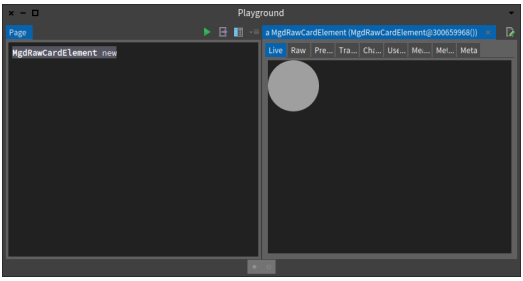

Figure 3-2 A card with circular background.

```
MgdRawCardElement >> cornerRadius
  ^ 10
```

We would like to have a rounded rectangle so we use the `BlRoundedRectangleGeometry` class. However, we need to give the corner radius we just defined as a parameter of the `cornerRadius:` class message :

```
MgdRawCardElement >> initialize

  super initialize.
  self size: 80 @ 80.
  self background: self backgroundPaint.
  self geometry: (BlRoundedRectangleGeometry cornerRadius: self
    cornerRadius).
  self card: (MgdCardModel new symbol: $a)
```

You should get then a visual representation close to the one shown in Figure 3-3.

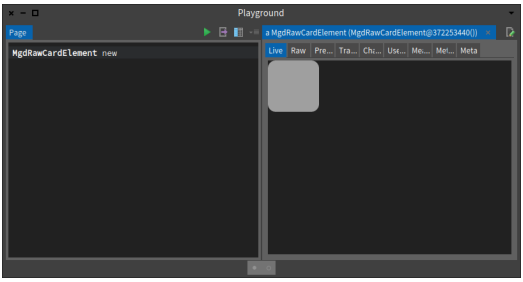

Figure 3-3 A rounded card.

3.4 Preparing flipping

We define now two methods

```
MgdRawCardElement >> drawBackside
  "nothing for now"
```

```
MgdRawCardElement >> drawFlippedSide
  "nothing for now"
```

We can now define the method that will draw our card

```
MgdRawCardElement >> drawCardElement

  self card isFlipped
      ifTrue: [ self drawFlippedSide ]
      ifFalse: [ self drawBackSide ]
```

Now we are ready to implement the backside and flipped side

3.5 Adding a cross

Now we are ready to define the backside of our card. We will start by drawing a line. To draw a line we should use the `BlLineGeometry`. At the end, we will create two lines and therefore two elements with a line geometry that will be added as children of the card Element.

```
Bloc uses parent-child relations between its elements thus leaving us with trees of e
```

A line is obviously defined between two points, we then need to give two points as parameters of the `from:to:` message from the `BlLineGeometry` class. Lines created using BlLineGeometry are a bit special as considered as "open geometries" meaning we don't define their color with the usual `background:` message like any other `BlElement`. Instead we define a border for our line and give this border the color we wanted (here we chose light green), we also define the thickness of our line with the border's width. Another particularity of open geometries is that they don't fit well with default outskirts in the current verision of Bloc, this is why we redefine them to be centered

```
MgdRawCardElement >> initializeFirstLine

  | line |
  line := BlElement new
            border: (BlBorder paint: Color lightGreen width: 3);
      geometry: BlLineGeometry new;
            outskirts: BlOutskirts centered.
  line
    when: BlElementLayoutComputedEvent
    do: [ :e | line geometry from: 0 @ 0 to: line parent extent ].
```

```
  ^ line
```

The message `when:do:` is used here to wait for the line parent to be drawn for the line to be defined, otherwise the `line parent extent` will be 0@0 and our line will not be displayed.

We can redefine `drawBackSide` and add the line we just created.

```
MgdRawCardElement >> drawBackSide

  self addChild: self initializeFirstLine.
  ^ self
```

Once this method is defined, refresh the inspector and you should get a card as in Figure 3-4.

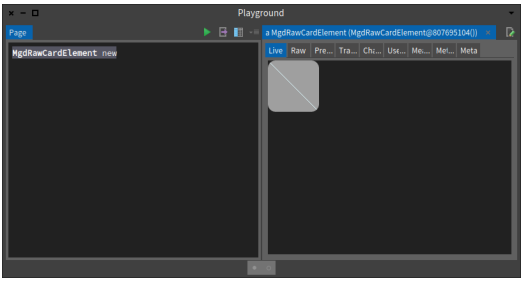

Figure 3-4 A rounded card with half of the cross.

3.6 Full cross

Now we can add the second line to build a full cross. We will add another instance variable holding our back side so that the lines are created only once during initialization. Our solution is defined as follows:

```
BlElement << #MgdRawCardElement
  slots: { #card #backSide };
  package: 'Bloc-MemoryGame-Demo-Elements'
```

```
MgdRawCardElement >> backSide: aBlElement
  backside := aBlElement
```

Before creating the getter of backside, we will create the method that will be responsible for adding the two lines together as a cross `initializeBackSide`. Let's start by creating the second line the same way we created the first one.

```
MgdRawCardElement >> initializeSecondLine

  | line |
  line := BlElement new
            border: (BlBorder paint: Color lightGreen width: 3);
        geometry: BlLineGeometry new;
            outskirts: BlOutskirts centered.
  line when: BlElementLayoutComputedEvent do: [ :e |
    line geometry from: 0 @ line parent height to: line parent width
    @ 0 ].
  ^ line
```

```
MgdRawCardElement >> initializeBackSide

  | firstLine secondLine cross |
  firstLine := self initializeFirstLine.
  secondLine := self initializeSecondLine.

  cross := BlElement new
             addChildren: {
                 firstLine.
                 secondLine };
             constraintsDo: [ :c |
               c horizontal matchParent.
               c vertical matchParent ].

  ^ cross
```

Our backside is then an Element holding both lines, we tell this element to match its parent using constraints, meaning the element size will scale according to the parent size, this also makes our lines defined to the correct points.

We can now define the backSide getter but with a little twist, using lazy initialization. This will create our element only when accessed the first time and not right after the initialization of the card element. This concept is very useful in certain situations and is great to know in case you need it, we define it as such :

```
MgdRawCardElement >> backSide
  ^ backSide ifNil: [ self initializeBackSide ]
```

We can finally redefine `drawBackSide` and call it in our initialization to draw our backside when the card is created.

```
MgdRawCardElement >> drawBackSide
  self removeChildren.
  self addChild: self backSide
```

```
MgdRawCardElement >> initialize

  super initialize.
  self size: 80 @ 80.
  self background: self backgroundPaint.
  self geometry:
    (BlRoundedRectangleGeometry cornerRadius: self cornerRadius).
  self card: (MgdCardModel new symbol: $a).
  self drawCardElement.
```

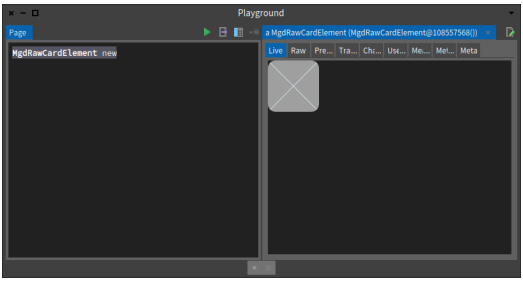

Figure 3-5 A card with a complete backside.

Now our backside is fully implemented and when you refresh your view, you should get the card as shown in Figure 3-5.

3.7 Flipped side

Now we are ready to develop the flipped side of the card. To see if we should change the card model you can use the inspector to get the card element and send it the message `card flip` or directly recreate a new card as follows:

```
| cardElement |
cardElement := MgdRawCardElement new.
cardElement card flip.
cardElement
```

You should get an inspector in the situation shown in Figure 3-6. Now we are ready to implement the flipped side.

Let us redefine `drawFlippedSide` as follows:

- First, we create a text element that holds the symbol of the card, we also give properties to this text by changing the font of the text but also its size and its color.
- Then we add the text element as a child of our card element

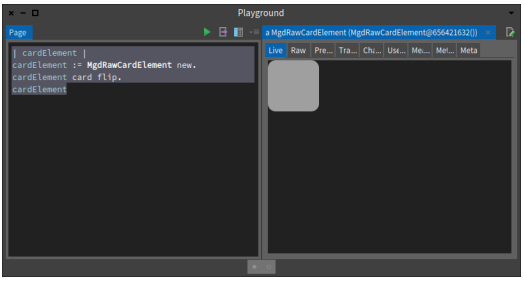

Figure 3-6 A flipped card without any visuals.

We will add an instance variable 'flippedSide' to our `MgdRawCardElement` class so that we create the text only once during the initialization. We don't forget about getter and setter.

```
BlElement << #MgdRawCardElement
  slots: { #card #backSide #flippedSide };
  package: 'Bloc-MemoryGame-Demo-Elements'
```

```
MgdRawCardElement >> flippedSide

  ^ flippedSide ifNil: [ self initializeFlippedSide ]
```

We can now create the method that will create the text for the flipped side, this method will be called during initialization.

```
MgdRawCardElement >> initializeFlippedSide

  | elt |
  elt := BlTextElement new text: self card symbol asRopedText.
  elt text fontName: 'Source Sans Pro'.
  elt text fontSize: 50.
  elt text foreground: Color white.
  ^ elt
```

Now we can redefine `drawFlippedSide` to add our text element as a child of our card element

```
MgdRawCardElement >> drawFlippedSide

  self removeChildren.
  self addChild: self flippedSide
```

When we refresh the display we can see the letter 'a' appear but it is positioned in the top left corner of our element, just as shown in Figure 3-7

Let's change that ! We will have to use constraints on our text element to tell him to get aligned in the center of its parent. To achieve this goal, we need to define a layout on the parent which is our card Element. We will use a Frame

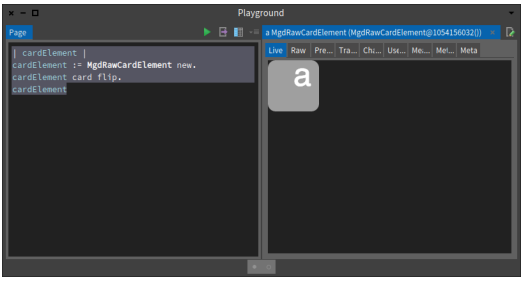

Figure 3-7 Not centered letter.

Layout that acts just like a real frame, meaning the text element will be able to get centered into the frame of its parent. Let's add the frame layout in the initialization of the card Element.

```
MgdRawCardElement >> initialize

  super initialize.
  self size: 80 @ 80.
  self background: self backgroundPaint.
  self geometry:
    (BlRoundedRectangleGeometry cornerRadius: self cornerRadius).
  self card: (MgdCardModel new symbol: $a).
  self drawCardElement.
  self layout: BlFrameLayout new.
```

We can now add the constraints to the text element.

```
MgdRawCardElement >> initializeFlippedSide

  | elt |
  elt := BlTextElement new text: self card symbol asRopedText.
  elt text fontName: 'Source Sans Pro'.
  elt text fontSize: 50.
  elt text foreground: Color white.
  elt constraintsDo: [ :c |
    c frame horizontal alignCenter.
    c frame vertical alignCenter ].
  ^ elt
``


With this definition, we get a centered letter as shown in Figure
    *@figCardCentered@*.

![Centered letter.](figures/CardCentered.png
    width=60&label=figCardCentered)
```

```
Now we are ready to work on the board game.

## Adding a board view

In the previous chapter, we defined all the card visualization. We
    are now ready to define the game board visualization.
Basically, we will define a new element subclass and set its layout.


Here is a typical scenario to create the game: we create a model and
    its view and we assign the model as the view's model.
```

game := MgdGameModel numbers. grid := MgdGameElement new. grid memoryGame: game.

```

### The GameElement class

Let us define the class `MgdGameElement` that will represent the
    game board.
As for the `MgdRawCardElement`, it inherits from the `BlElement`
    class.
This view object holds a reference to the game model.
```

BlElement « #MgdGameElement slots: { #memoryGame }; package: 'Bloc-MemoryGame-Demo-Elements'

```

We define the `memoryGame:` setter method. We will extend it to
    create
all the card elements shortly.
```

MgdGameElement » memoryGame: aMgdGameModel memoryGame := aMgdGameModel

-

MgdGameElement » memoryGame ^ memoryGame

```

During the object initialization, we set the layout \(i.e., how
    sub-elements are placed inside their container\).
Here we define the layout to be a grid layout and we set it as
    horizontal.
```

MgdGameElement » initialize super initialize. self background: Color veryLightGray. self layout: BlGridLayout horizontal.

```
### Creating cards


When a model is set for a board game, we use the model information
    to perform the following actions:
- we set the number of columns of the layout
- we create all the card elements paying attention to set their
    respective model.

Note in particular that we add all the card graphical elements as
    children of the board game using the message `addChild:`.
```

MgdGameElement » memoryGame: aGameModel memoryGame := aGameModel. memoryGame availableCards do: [:aCard | self addChild: (MgdRawCardElement card: aCard)]

MgdRawCardElement class » card: aCard ^ self new card: aCard

```
![A first board - not really working.](figures/BoardStarted.png
    width=60&label=figBoardStarted)

When we refresh the inspector we obtain a situation similar to the
    one of Figure *@figBoardStarted@*.
It shows that only a small part of the game is displayed. This is
    due to the fact that the game element
did not adapt to its children.


### Updating the container to its children


A layout is responsible for the layout of the children of a
    container but not of the container itself.
For this, we should use constraints.
```

MgdGameElement » initialize super initialize. self layout: BlGridLayout horizontal. self constraintsDo: [:aLayoutConstrants | aLayoutConstraints horizontal fitContent. aLayoutConstraints vertical fitContent]

```
Now when we refresh our view we should get a situation close to the
    one presented in Figure*@figBoardOneRow@*, i.e., having
just one row. Indeed we never mentioned to the layout that it should
    layout its children into a grid, wrapping after four.
```

```
![Displaying a row.](figures/BoardOneRow.png
    width=60&label=figBoardOneRow)


### Getting all the children displayed


We modify the `memoryGame:` method to set the number of columns
that the layout should handle.
```

MgdGameElement » memoryGame: aGameModel memoryGame := aGameModel. self layout columnCount: memoryGame gridSize. memoryGame availableCards do: [:aCard | self addChild: (self newCardElement card: aCard)]

```


Once the layout is set with the correct information we obtain a full
    board as shown in Figure *@figBoardFull@*.

![Displaying a full board.](figures/BoardFull.png
    width=60&label=figBoardFull)


### Separating cards


To offer a better identification of the cards, we should add some
    space between each of them.
We achieve this by using the message `cellSpacing:` as shown below.

We take the opportunity to change the background color using the
    message `background:`.
Note that a background is not necessarily a color but that color is
    polymorphic to a background
therefore the expression `background: Color gray darker` is
    equivalent to `background: (BlBackground paint: Color gray
    darker)`.
```

MgdGameElement » initialize super initialize. self background: (BlBackground paint: Color gray darker). self layout: (BlGridLayout horizontal cellSpacing: 20). self constraintsDo: [:aLayoutConstraints | aLayoutConstraints horizontal fitContent. aLayoutConstraints vertical fitContent]

```


Once this method is changed, you should get a situation similar to
    the one described by Figure *@figBoardFullSpace@*.
![Displaying a full board with space.](figures/BoardFullSpace.png
    width=60&label=figBoardFullSpace)
```

```
Before adding interaction let's define a method `openWith:` that
    will open our game element with a given model.
```

MgdGameElement class » openWith: aMgdGameModel

| space gameElement | space := BlSpace new. gameElement := self new memoryGame: aMgdGameModel. space root addChild: gameElement.

space show

```
If we try to open this, we clearly see our game element with all its
    cards but there's still some blank space around it, we can deal
    with this by changing the size of the space we put our game
    element into.
```

MgdGameElement class » openWith: aMgdGameModel

| space gameElement | space := BlSpace new. gameElement := self new memoryGame: aMgdGameModel. space root addChild: gameElement. space pulse. space extent: gameElement extent.

space show

```
Notice we send `pulse` to our space before changing its extent, it
    is required to tell the space to be prepared to change.

We are now ready for adding interaction to the game.
```

CHAPTER 4

Adding Interaction

Now we will add interaction to the game. We want to flip the cards by clicking on them. Bloc supports such situations using two mechanisms: on one hand, event listeners handle events and on the other hand, the communication between the model and view is managed via the registration to announcements sent by the model.

4.1 Adding events and event listeners

In Bloc, there is of course plenty of Events and we will focus on `BlClickEvent`. We can also say that events are easily managed through event handlers

Now we should add an event handler to each card because we want to know which card will be clicked and pass this information to the game model.

```
MgdGameElement >> initialize

  super initialize.
  self size: 80 @ 80.
  self background: self backgroundPaint.
  self geometry:
    (BlRoundedRectangleGeometry cornerRadius: self cornerRadius).
  self drawBackSide.
  self layout: BlFrameLayout new.
  self addEventHandlerOn: BlClickEvent do: [ :anEvent | self click ]
```

We can easily see that whenever our card Element will receive a click Event, we will send the `click` message to this element

Please note, that if we created an instance of BlEventListener and added it as an event handler, we can reuse the same event handler for all card elements. It allows us to reduce overall memory consumption and improve game initialization time.

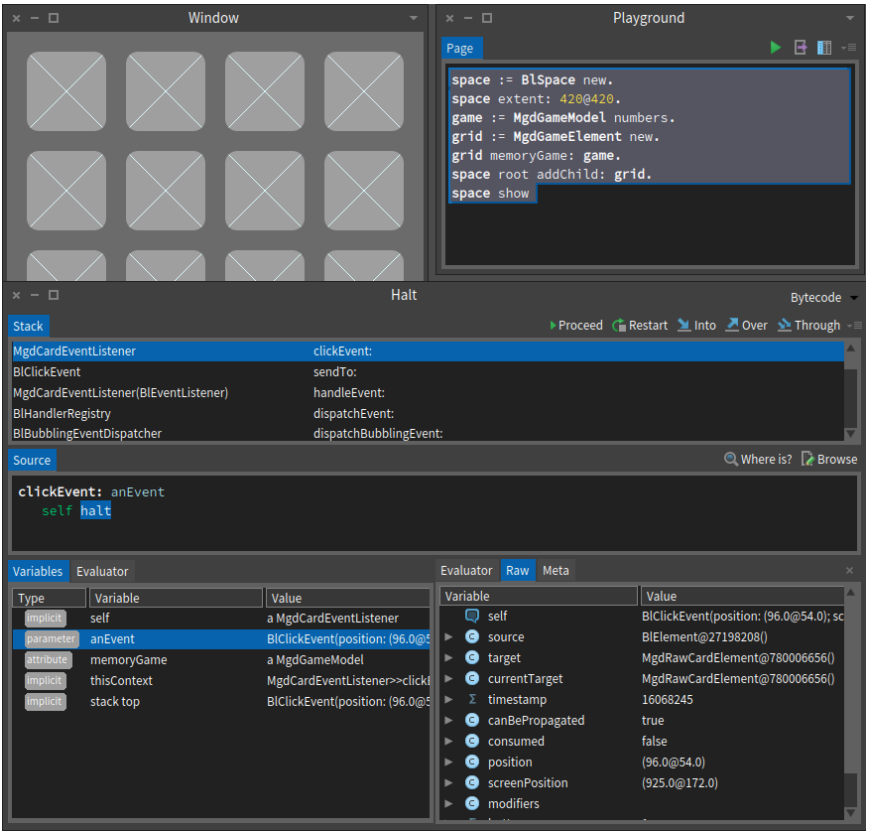

Figure 4-1 Debugging the clickEvent: anEvent method.

Now the preview is not enough and we should create a window and embed the game element. Then when you click on a card you should get a debugger as shown in Figure 4-1.

4.2 Specialize click

Now we can specialize the `click` method as follows:

- We tell the model we just chose this card
- We draw our cardElement according to its card state

```
MgdRawCardElement >> click

  self parent memoryGame chooseCard: self card.
  self drawCardElement
```

It means that the memory game model is changed but we cards don't flip back after mistaking the symbols. Indeed this is normal. We never made sure that visual elements were listening to model changes except for when we click on it. This is what we will do in the following chapter.

4.3 Connecting the model to the UI

Now we show how the domain communicates with the user interface: the domain emits notifications using announcements but it does not refer to the UI elements. It is the visual elements that should register to the notifications and react accordingly. We can prepare the message that will tell our elements to disappear we both cards match, otherwise we just tell our cards to flip back and draw their backside

```
MgdRawCardElement >> disappear
  "nothing for now"
```

Now we can modify the setter so that when a card model is set to a card graphical element, we register to the notifications emitted by the model. In the following methods, we make sure that on notifications we invoke the method just defined.

```
MgdRawCardElement >> initializeAnnouncements

  card announcer
    when: MgdCardDisappearAnnouncement
    send: #disappear
    to: self.
  card announcer
    when: MgdCardFlipBackAnnouncement
    send: #drawBackSide
    to: self
```

```
MgdRawCardElement >> card: aMgdCard

  card := aMgdCard.
  self initializeAnnouncements
```

4.4 Handling disappear

There are two ways to implement the disappearance of a card: Either setting the opacity of the element to 0 (Note that the element is still present and receives events.)

```
MgdRawCardElement >> disappear
  self opacity: 0
```

Or changing the visibility as follows:

```
MgdRawCardElement >> disappear
  self visibility: BlVisibility hidden
```

Note that in the latter case, the element no longer receives events. It is used for layout.

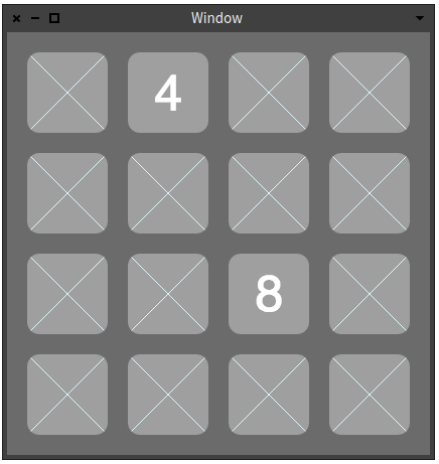

Figure 4-2 Selecting two cards that are not in pair.

4.5 Refreshing on missed pair

When the player selects two cards that are not a pair, we present the two cards as shown in Figure 4-3. Now clicking on another card will flip back the previous cards.

Remember, a card will raise a notification when flipped in either direction.

```
MgdCardModel >> flip
  flipped := flipped not.
  self notifyFlipped
```

In the method `#resetStep` we see that all the previous cards are flipped (toggled).

```
MgdGameModel >> resetStep
  | lastCard |

  lastCard := self chosenCards  last.
```

```
self chosenCards
  allButLastDo: [ :aCard | aCard flip ];
  removeAll;
  add: lastCard
```

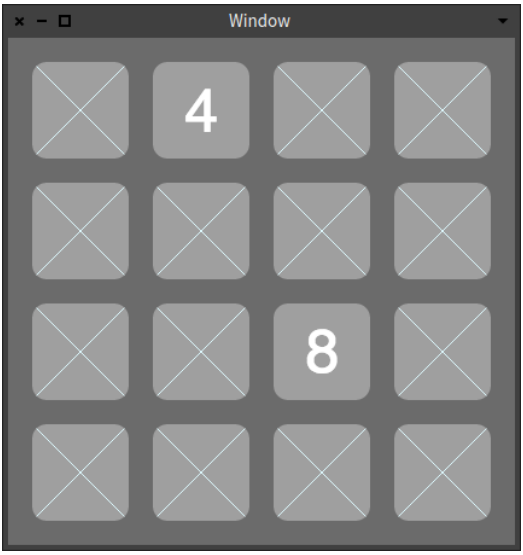

Figure 4-3 Selecting two cards that are not a pair.

4.6 Conclusion

At this stage, you are done for the simple interaction. Future versions of this document will explain how to add animations.

Bibliography