A memory game: A simple tutorial with Bloc

S. Ducasse, E. Demeulenaere, and M. Dias

May 22, 2024

Keepers of the lighthouse
Édition : BoD - Books on Demand,
12/14 rond-point des Champs-Élysées,75008 Paris
Impression : Books on Demand GmbH, Norderstedt, Allemagne
ISBN: XXXXXXXXXXXXXXX
Dépôt légal : Month/YEAR
Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

This booklet is the second iteration on the Memory game tutorial. It is inspired from the 2017 original Memory Game tutorial written by A. Chis, A. Syrel and S. Ducasse and entitled "Building a memory game with Bloc". This old tutorial is available on https://books.pharo.org. It used the Sparta canvas to draw the card visual elements. In the Bloc version distributed by the Pharo consortium and that will be part of Pharo in the future, the Sparta canvas has been replaced by Alexandrie the graphical canvas used in Pharo. In addition in the current tutorial, instead of using the canvas low-level API, we use basic bloc elements to display the card visual elements. We could draw some visual using the Alexandrie canvas but this low-level API is not the best for an introduction to Bloc so this is why we decided to take a more pedagogical path.

CHAPTER 1

Objectives of this book

Bloc is the new graphics library for Pharo. A graphics library implies several aspects such as coordinate systems, drawing shape, clipping, and event management.

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 working 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 := MGGame withNumbers.
visual := MGGameElement new.
visual memoryGame: game.

space := BlSpace new.
```

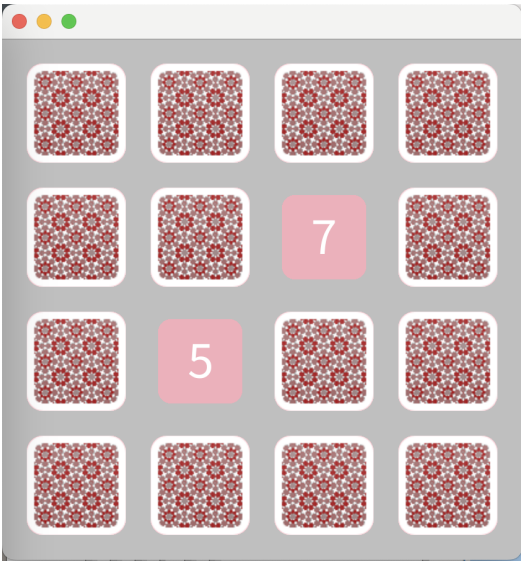

Figure 1-1 The game after the player has selected two cards: facedown cards are represented with a cross and turned cards with their number.

```
space extent: 420@420.
space root addChild: visual.
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. Note that this last sequence should be better packaged as a message to the `MGGameElement`.

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/`

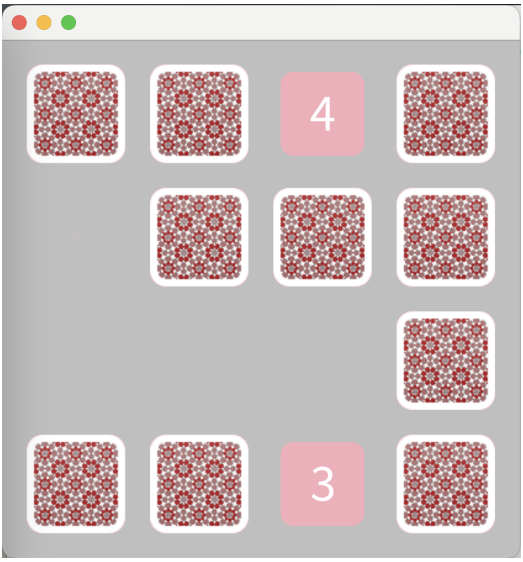

Figure 1-2 Another state of the memory game after the player has correctly matched two pairs.

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: 'BlocMemoryTutorial';
    repository: 'github://pharo-graphics/Bloc-Memory-Tutorial/src';
    load
```

After you have loaded the MemoryTutorial project, you will get two new packages: `Bloc-Memory` and `Bloc-MemoryGame-Tests`. `Bloc-Memory-Tests` contains the full implementation of the game.

You can browse a model of the game just executing the following code snippet:

```
MGGame withEmoji
```

To get a working game just execute the following expression.

```
MGGameElement openWithNumber
```

Since we give you all the code, if you want to write it by your own, use a different prefix for the classes.

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 communicate 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, this model is fully defined in the package.

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 << #MGCard
  slots: { #symbol . #flipped . #announcer};
  tag: 'Model';
  package: 'Bloc-Memory'
```

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

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

```
MGCard >> symbol: aCharacter
  symbol := aCharacter
```

```
MGCard >> symbol
  ^ symbol
```

```
MGCard >> isFlipped
  ^ flipped
```

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

2.2 Card simple operations

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

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

```
MGCard >> 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 `MGCardFlippedAnnouncement` and `MGCardDisappearAnnouncement`. The graphical elements have to register subscriptions to these announcements as we will see later.

```
MGCard >> notifyFlipped
  self announcer announce: MGCardFlippedAnnouncement new
```

```
MGCard >> notifyDisappear
  self announcer announce: MGCardDisappearAnnouncement new
```

Here, `MGCardFlippedAnnouncement` and `MGCardDisappearAnnouncement` are subclasses of `Announcement`.

```
Announcement << #MGCardFlippedAnnouncement
  tag: 'Events';
  package: 'Bloc-Memory'
```

```
Announcement << #MGCardDisappearAnnouncement
  tag: 'Events';
  package: 'Bloc-Memory'
```

We add one final method to print a card more nicely and we are done with the card model!

```
MGCard >> 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 << #MGGame
  slots: { #availableCards . #chosenCards};
  tag: 'Model';
  package: 'Bloc-MemoryGame'
```

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

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

```
MGGame >> availableCards
  ^ availableCards
```

The `chosenCards` collection will hold at max two cards in this version of the game.

```
MGGame >> 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. Later this could be turned into an instance variable and be configured.

```
MGGame >> gridSize
  "Return grid size"
  ^ 4
```

The method `matchesCount` indicates that two identical cards are needed to match.

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

```
MGGame >> 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 to fill up the game board.

```
MGGame >> initializeForSymbols: characters

  aCollectionOfCharacters size = (self cardsCount / self
    matchesCount)
    ifFalse: [ self error: 'Amount of characters must be equal to
    possible all combinations' ].

  aCollectionOfCharacters do: [ :aSymbol |
    1 to: self matchesCount do: [ :i |
    availableCards add: (MGCard new symbol: aSymbol) ] ].
  availableCards := availableCards shuffled
```

2.7 Game logic

Next, we define the method `chooseCard:`. It will be called when a user selects a card. This method is 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.

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

Completed.

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.

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

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

Note that the logic of chosenCardMatch looks more complex than expected but it works with matches that require more than two cards.

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

Reset.

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.

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

```
MGGame >> 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

Basic 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. Elements are the basic visual building blocks of Bloc. In subsequent chapters, we will add interaction using event listeners.

3.1 First: the card element

Our graphic element representing a card will be a subclass of the `BlElement`. This element has, in addition, a reference to a card model (as defined in the previous chapter).

```
BlElement << #MGCardElement
  slots: { #card };
  tag: 'Elements';
  package: 'Bloc-Memory'
```

We define the corresponding accessors since the setter methods will be the place to hook registration for the communication between the model and the view, as we will show later.

```
MGCardElement >> card
  ^ card
```

```
MGCardElement >> card: aMgCard
  card := aMgCard
```

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

```
MGCardElement >> backgroundPaint
  "Return a BlPaint that should be used as a background (fill)
  of both back and face sides of the card. Colors are polymorphic
  with BlPaint and therefore can be used too."

  ^ Color pink darker
```

We define a method `initialize` to set the size and the default color as well as a card model object.

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

3.2 Starting to draw a card

In Bloc, `BlElements` draw themselves onto the integrated canvas of the inspector as we inspect them, take a look at our element by executing this (See Figure 3-1).

```
MGCardElement new inspect
```

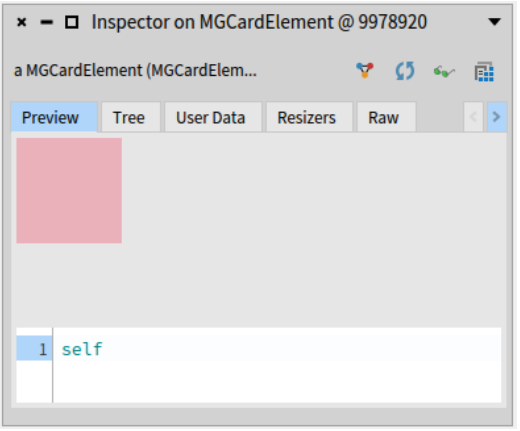

Figure 3-1 A first extremely basic representation of face down card.

3.3 Improving the card visual

Instead of displaying a full rectangle, we want a better visual. Bloc lets 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 available geometries by looking at subclasses of `BlElementGeometry`. We can also add a png as we will show later.

We can start giving a circle shape to our element, we will need to use the `geometry:` message and give a `BlCircleGeometry` as an argument. You should obtain an inspector as shown in Figure 3-2.

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

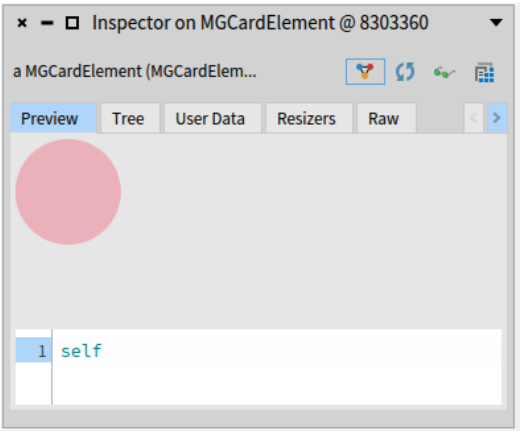

Figure 3-2 A card with circular geometry.

However, we don't want the card to be a circle either. We would like to have a rounded rectangle so we use the `BlRoundedRectangleGeometry` class. We need to give the corner radius as a argument of the `cornerRadius:` class message. This is what we do in the following `initialize` method.

```
MGCardElement >> initialize
  super initialize.
  self size: 80 @ 80.
  self background: self backgroundPaint.
  self geometry: (BlRoundedRectangleGeometry cornerRadius: 12).
  self card: (MGCard new symbol: $a)
```

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

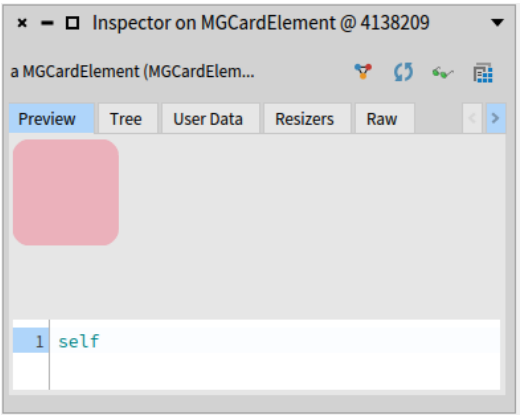

Figure 3-3 A rounded card.

3.4 Two faces of a card

A card has two faces: its back and its front. There are several approaches to manage this. In this tutorial, we will compose i.e, add/remove elements as children of the `MGCardElement` instance.

- For the back we will add an element containing a png.
- For the front we will add a text element with a number.

3.5 Defining some utilities

Since we do not want to duplicate size card logic, we define a simple method to return the extent of a card.

```
MGCardElement >> cardExtent
  ^ 80@80
```

And we use it to initialize the card element:

```
MGCardElement >> initialize
  super initialize.
  self size: self cardExtent.
  self background: self backgroundPaint.
  self geometry: (BlRoundedRectangleGeometry cornerRadius: 12).
  self card: (MGCard new symbol: $a)
```

We could do the same for the corner radius but this is not important now.

To manage the back of a card, we will read a png file and define it as a method to be able to version it. You can find the current definition on the class side of `MGCardElement`.

Since this is a large method that contains the textual serialization of a png we only show the beginning of its definition:

```
MGCardElement class >> cardbackForm
  ^ Form
    extent: 80@80
    depth: 32
    bits: (Bitmap newFrom: #(16777215 16777215 16777215 16777215
    16777215 16777215 16777215 16777215 318767103 1526726655
    2936012799 3439329279 3942645759 4294967295 4294967295
    4294967295 4294967295
    ...
```

You can read Section 3.10 to get more information about form and PNG handling.

3.6 Defining elements for card faces

We add two instance variables `backElement` and `frontElement` to refer to the children that represent the contents of the two card faces.

```
BlElement << #MGCardElement
  slots: { #card . #backElement . #frontElement };
  tag: 'Elements';
  package: 'Bloc-Memory'
```

We redefine the `initialize` method to create the `backElement` as well as adding a layout for placement of the children of the `MGCardElement` instances.

```
MGCardElement >> initialize
  super initialize.
  backElement := BlElement new
    background: self class cardbackForm;
    size:self cardExtent;
    yourself.
  self size: self cardExtent.
  self layout: BlLinearLayout new alignCenter.
  self background: self backgroundPaint.
  self geometry: (BlRoundedRectangleGeometry cornerRadius: 12).
  self card: (MGCard new symbol: $a).
```

In the following added part:

```
backElement := BlElement new
  background: self class cardbackForm;
  size:self cardExtent;
  yourself.
```

```
frontElement := BlTextElement new.
```

We simply set a form as the background of this new element. The method `cardbackForm` is the method illustrated above and that you can get loading the code of this tutorial. If you want to create your own method, copy the logic of the method and paste the contents of the stream passed to the `storeOn:` method as in the following script:

```
String streamContents: [ :str | myForm storeOn: str ]
```

About layouts

In addition we initialized the layout to be a linear layout so that each child of our card element is centered.

```
self layout: BlLinearLayout new alignCenter.
```

Better readability

We extract the back element creation in its own method `initializeBackElement`.

```
MGCardElement >> initializeBackElement
  backElement := BlElement new
    background: self class cardbackForm;
    size: self cardExtent;
    yourself
```

```
MGCardElement >> initialize
  super initialize.
  self initializeBackElement.
  self size: self cardExtent.
  self layout: BlLinearLayout new alignCenter.
  self background: self backgroundPaint.
  self geometry: (BlRoundedRectangleGeometry cornerRadius: 12).
  self card: (MGCard new symbol: $a)
```

3.7 Handling the front face

Now we will work on the front face visual. First we will initialize a text element to a simple text element. This element will be updated for each card model.

```
MGCardElement >> initializeFrontElement
  frontElement := BlTextElement new
```

```
MGCardElement >> initialize
  super initialize.
  self initializeBackElement.
  self initializeFrontElement.
```

```
  self size: self cardExtent.
  self layout: BlLinearLayout new alignCenter.
  self background: self backgroundPaint.
  self geometry: (BlRoundedRectangleGeometry cornerRadius: 12).
  self card: (MGCard new symbol: $a)
```

Second, we define the method `fillUpFrontElement` as follows:

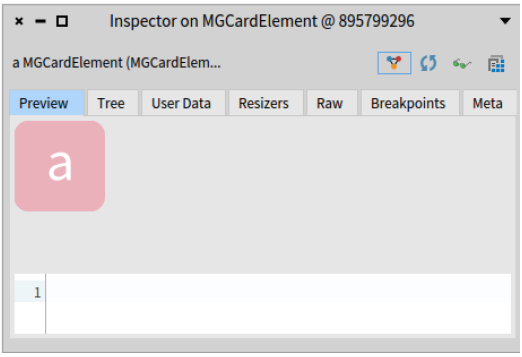

Figure 3-4 Front face with a letter: inspect MGCardElement new.

```
MGCardElement >> fillUpFrontElement
  frontElement text: (card symbol asString asRopedText
    fontSize: self fontPointSize;
    foreground: self fontColor;
    yourself)
```

To see the actual effect (See Figure 3-4) we redefine the method `card:` as follows and inspect the result of `MGCardElement new`.

Let us explain it a bit, when a new card model is set, the text element representing the front face is updated, the current children are emptied and the front element is added as child of the card element.

```
MGCardElement >> card: aCard
  card := aCard.
  self fillUpFrontElement.
  self removeChildren.
  self addChild: frontElement
```

We will use the same logic to switch back to the back face. We are now ready to implement the flipping of the card.

3.8 Flipping faces

Let us define two methods `showBackFace` and `showFrontFace` to encapsulate the logic of switching to a different face visual.

```
MGCardElement >> showBackFace
  self removeChildren.
  self addChild: backElement
```

```
MGCardElement >> showFrontFace
  self removeChildren.
  self addChild: frontElement
```

We redefine the method `card:` to put in place the corresponding visual based on the flipped status of the card model.

```
MGCardElement >> card: aCard
  card := aCard.
  self fillUpFrontElement.
  card isFlipped
    ifTrue: [ self showFrontFace ]
    ifFalse: [ self showBackFace ].
```

We can do another step to factor nicely the behavior of the method `card:`. We define a new method called `showCardFace`

```
MGCardElement >> showCardFace
  card isFlipped
    ifTrue: [ self showFrontFace ]
    ifFalse: [ self showBackFace ]
```

```
MGCardElement>> card: aCard
  "Attach a card model and subscribe to its announcements."

  card := aCard.
  self fillUpFrontElement.
  self showCardFace.
```

3.9 From the model 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 := MGCardElement new.
cardElement card flip.
cardElement
```

3.10 Resources

This section complements Section 3.5. If you want to use your own pngs, have a look at the class `ReaderWriterPNG` that converts PNG files into Forms. A form is a piece of graphical memory internally used by Pharo. So you have to convert your graphics from or to Forms.

Here are some little scripts (that you should execute in order if you want to reproduce their effect.)

To save a form as a PNG on your disk:

```
PNGReadWriter putForm: MGCardElement cardbackForm onFileNamed:
    'CardBack.png'
```

To save a form as a text (as shown above) that you can later execute to recreate the original form.

```
| text |
text := String streamContents: [ :str |
  (PNGReadWriter formFromFileNamed: 'CardBack.png') storeOn: str ].

"to recreate the form from its textual representation"
text := MGCardElement
Object readFrom: text readStream
```

Using Uuencoded strings

Storing a form in a plain text can produce large files, you can also use uuencoded of them. This is what IconFactory project is doing.

If you want to manage forms as the method cardbackForm provided in the project, you can have a look at the IconFactory project on github.

```
Metacello new
    baseline: #IconFactory;
    repository: 'github://pharo-graphics/IconFactory';
    load
```

This project supports the definition of form as textual resources in methods that can be then versioned altogether with the code.

Given a base64 encoded string you can get a form with the following expression, here we take the base64 encoded string from `IconFactoryTest new exampleIconContents`

```
Form fromBinaryStream: IconFactoryTest new exampleIconContents
    base64Decoded asByteArray readStream
```

Following this you can generate a method body with a cache (here named icons) as follows:

```
iconMethodTemplate
  ^ '{1}
  "Private - Generated method"
  ^ self icons
    at: #{1}
    ifAbsentPut: [ Form fromBinaryStream: IconFactoryTest new
    exampleIconContents
           base64Decoded asByteArray readStream ]'
```

Where the first argument is part of a method name for example 'tintin'.

3.11 Conclusion

We have all the visual elements for the card, so we are ready to work on the board game.

CHAPTER 4

Adding a board view

In the previous chapter, we defined all the card visualizations. We are now ready to define the game board visualization. Basically, we will define a new `BlElement` 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 := MGGame withNumbers.
board := MGGameElement new.
board memoryGame: game.
```

4.1 The GameElement class

Let us define the class `MGGameElement` that will represent the game board. As for the `MGCardElement`, it inherits from the `BlElement` class. The instance variable `memoryGame` holds a reference to the game model.

```
BlElement << #MGGameElement
  slots: { #memoryGame };
  package: 'Bloc-MemoryGame'
```

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

```
MGGameElement >> memoryGame: aGameModel
  memoryGame := aGameModel
```

```
MGGameElement >> 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 with a little extra space around the card element and we set it as horizontal.

```
MGGameElement >> initialize
  super initialize.
  self background: Color veryLightGray.
  self layout: (BlGridLayout horizontal cellSpacing: 20).
```

4.2 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 and
- we create all the card elements paying attention to set their respective model.

```
MGGameElement >> memoryGame: aGameModel
  memoryGame := aGameModel.
  memoryGame availableCards
    do: [ :aCard | self addChild: (MGCardElement new card: aCard) ]
```

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

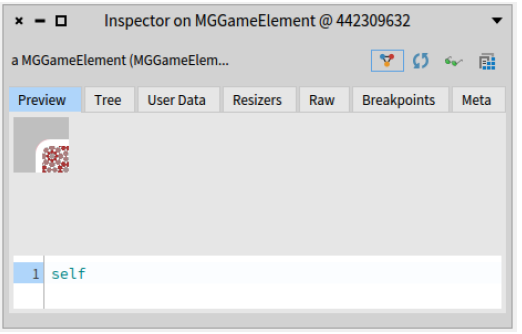

Figure 4-1 A first board - not really working.

```
game := MGGame withNumbers.
board := MGGameElement new.
board memoryGame: game.
board
```

When we inspect the previous code snippet, we obtain a situation similar to the one of Figure 4-1. It shows that only a small part of the game is displayed.

This is due to the fact that the board game element did not adapt to its children.

4.3 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.

```
MGGameElement >> initialize
  super initialize.
  self background: Color veryLightGray.
  self layout: (BlGridLayout horizontal cellSpacing: 20).
  self
    constraintsDo: [ :aLayoutConstrants |
      aLayoutConstraints horizontal matchParent.
      aLayoutConstraints vertical matchParent ]
```

Now when we refresh our view we should get a situation close to the one presented in Figure4-2, 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.

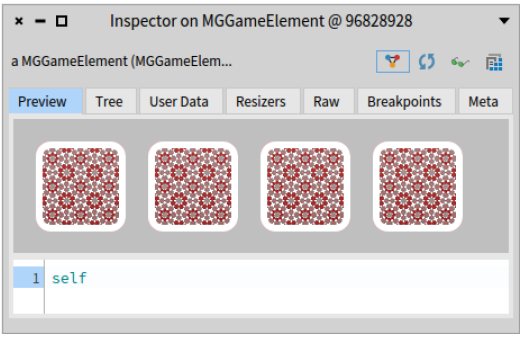

Figure 4-2 Displaying a row.

4.4 Getting all the children displayed

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

```
MGGameElement >> memoryGame: aGameModel
  memoryGame := aGameModel.
  self layout columnCount: memoryGame gridSize.
  memoryGame availableCards
    do: [ :aCard | self addChild: (MGCardElement new card: aCard) ]
```

Once the layout is set with the correct information we obtain a full board as shown in Figure 4-3.

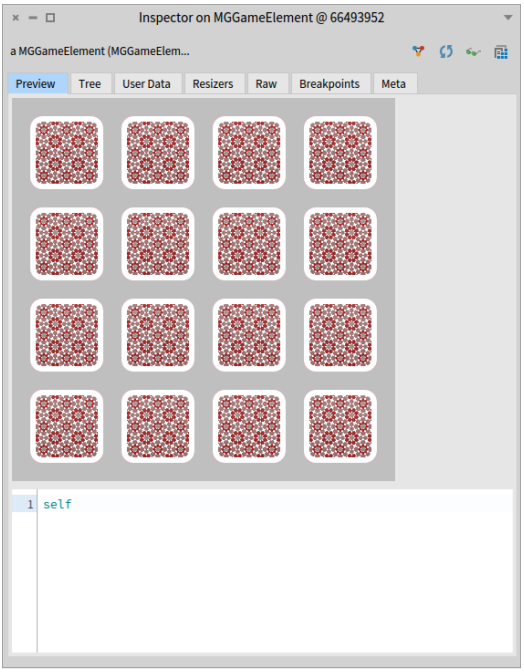

Figure 4-3 Displaying a full board.

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

```
MGGameElement class >> openWithNumber
  | aGameElement space |
  aGameElement := MGGameElement new
    memoryGame: MGGame withNumbers;
    yourself.
  space := BlSpace new.
  space root addChild: aGameElement.
  space root whenLayoutedDoOnce: [ space extent: 420 @ 420 ].
  space show
```

We are now ready to add interaction to the game.

4.5 Conclusion

The board is now ready to display the full game but the user interaction is still missing. This is what we will investigate in the next chapter.

CHAPTER 5

Adding game interactions

In this chapter, 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.

5.1 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.

```
MGGameElement >> initialize
  super initialize.
  self initializeBackElement.
  self initializeFrontElement.
  self size: self cardExtent.
  self layout: BlLinearLayout new alignCenter.
  self background: self backgroundPaint.
  self geometry: (BlRoundedRectangleGeometry cornerRadius: 12).
  self card: (MGCard new symbol: $a).
  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

5.2 Defining a click method

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

- We tell the model we just chose this card.
- We switch the card visual according to its card state.

```
MGCardElement >> click
  self parent memoryGame chooseCard: self card.
  self showCardFace
```

It means that the memory game model is changed but the 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 .

5.3 Alternate design

Alternatively to use `self addEventHandlerOn: BlClickEvent do: [ :anEvent | self click ]`, we can define a specific event listener and reuse it over all cards. In this case we can reuse the 444 event handler for all card elements. It allows us to reduce overall memory consumption and improve game initialization time.

```
BlEventListener << #MGCardEventListener
  slots: { #memoryGame };
  package: 'Bloc-Memory'
```

```
MGCardEventListener >> clickEvent: anEvent
  memoryGame chooseCard: anEvent currentTarget card
```

```
MGGameElement >> memoryGame: aGame

  | aCardEventListener |
  game := aGame.
  aCardEventListener :=
    MGCardEventListener new
      memoryGame: aGame;
      yourself.

  self layout columnCount: game gridSize.

  game availableCards do: [ :aCard |
    | cardElement |
    cardElement :=
      MGCardElement new
        card: aCard;
        addEventHandler: aCardEventListener;
        yourself.
    self addChild: cardElement ]
```

5.4 Connecting the model to the UI

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.

```
MGCardElement >> onDisappear
  "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.

```
MGCardElement >>card: aCard
  card := aCard.
  self fillUpFrontElement.
  self showCardFace.

  card announcer
    when: MGCardFlippedAnnouncement
    send: #showCardFace to: self;
    when: MGCardDisappearAnnouncement
    send: #onDisappear to: self
```

5.5 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.)

```
MGCardElement >> onDisappear
  self opacity: 0
```

Or changing the visibility as follows:

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

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

5.6 Reminder on missed pair

Remember that when the player selects two cards that are not a pair, we present the two cards as shown in Figure 5-1. Now clicking on another card

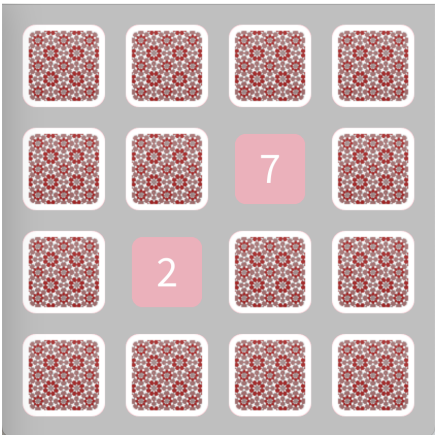

Figure 5-1 Selecting two cards that are not in pair.

will flip back the previous cards. In addition, a card raises a notification when flipped in either direction.

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

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

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

5.7 Conclusion

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

CHAPTER 6

Adding animations

In this chapter, we will add animations to the game. We will define animations and add them to the queue of card animation. Then Bloc logic will execute the animation queue on the receiver. This chapter will illustrate how animations can be composed or run in parallel.

6.1 Card flipping animations

When the user flip the two cards we want to shrink them a bit to make them stand out. We use a simple scaling transformation. An animation will modify the attributes (size, color, position...) of the element on which it is applied. Once the transformation is defined it is added to the animation queue of the receiver using the message `addAnimation:`.

```
MGGameElement >> onFlippedFace
  | animation |
  animation := BlTransformAnimation scale: 0.85 @ 0.85.
  animation
    absolute;
    easing: BlQuinticInterpolator default;
    duration: 0.3 seconds.
  self addAnimation: animation.
  self showFrontFace
```

We define another similar animation to put back the full size of flipped back card.

```
MGGameElement >> onFlippedBack
  | animation |
  animation := BlTransformAnimation scale: 1@1.
  animation
    absolute;
    easing: BlEasing bounceOut;
    duration: 0.35 seconds.
  self addAnimation: animation.
  self showBackFace
```

We modify the `showCardFace` method to invoke the method performing the animations prior to changing the visual of the cards.

```
MGGameElement >> showCardFace
  card isFlipped
    ifTrue: [ self onFlippedFace ]
    ifFalse: [ self onFlippedBack ]
```

6.2 Card disappearing animation

When two cards match we want them to enlarge a bit to get player's attention. When a card disappears, we will compose several animations: one that will grow the element, another that will change its opacity and in parallel its size.

The opacity animation will slowly make the card transparent while the size will make it is smaller.

We replace the previous `onDisappear` method by the following one.

```
MGCardElement >> onDisappear
  | vanish enlarge minimize disappear |
  enlarge := BlTransformAnimation scale: 1.15 @ 1.15.
  enlarge
    absolute;
    easing: BlEasing bounceOut;
    duration: 0.5 seconds.
  vanish := BlOpacityAnimation new
              opacity: 0;
              duration: 0.35 seconds.
  minimize := BlTransformAnimation scale: 0.01 @ 0.01.
  minimize
    absolute;
    easing: BlEasing linear;
    duration: 0.35 seconds.
  disappear := BlParallelAnimation withAll: { vanish . minimize }.

  self addAnimation: (BlSequentialAnimation withAll: { enlarge.
    disappear })
```

6.3 Conclusion

This chapter presented how animations are simple to be defined in Bloc and how they are useful to enhance user experience.

CHAPTER 7

Graphical alternatives

While in the previous chapters, we used a PNG for the back of the card, in this chapter we show alternative solutions: (1) draw using a low-level API such as the one proposed by the Alexandrie canvas or (2) compose elementary `BlElements`.

7.1 Using the Alexandrie Canvas

Alexandrie is a Cairo-based optimized canvas. It is the default canvas supported by Bloc. Every element is ultimately drawn with Alexandrie and Cairo. Alexandrie abstracts the Cairo interface for low-level operations.

Now you can also draw the visual aspect of a BlElement directly using Cairo operations. For this, it is enough to override the method `aeDrawOn:`.

We give an example of a `ClockElement`. It shows that the developer should use a cairo context provided by a factory and configure such context to draw elementary paths or other properties such as borders.

```
ClockElement >> aeDrawOn: aeCanvas
  "draw clock tick on frame"
  super aeDrawOn: aeCanvas.
  aeCanvas setOutskirtsCentered.
  0 to: 11 do: [ :items |
    | target |
    target := (items * Float pi / 6) cos @ (items * Float pi / 6)
    sin.
    items % 3 == 0
      ifTrue: [ aeCanvas pathFactory: [ :cairoContext |
          cairoContext
            moveTo: center;
```

```
            relativeMoveTo: target * 115;
            relativeLineTo: target * 35;
            closePath ].
          aeCanvas setBorderBlock: [
            aeCanvas
              setSourceColor: Color black;
              setBorderWidth: 8 ] ]
        ifFalse: [ aeCanvas pathFactory: [ :cairoContext |
            cairoContext
              moveTo: center;
              relativeMoveTo: target * 125;
              relativeLineTo: target * 25;
              closePath ].
          aeCanvas setBorderBlock: [
            aeCanvas
              setSourceColor: Color black;
              setBorderWidth: 6 ] ].
      aeCanvas drawFigure ]
```

7.2 Using simple BlElements

We can also compose `BlElements` and place them as children of the main element. As such they form a tree of objects. Here is for example the same `ClockElement`, this time defined using elementary `BlElements`.

The `initClockFrame` method is invoked when initializing the object. Developers use normal element API such as geometry and border to define the visual aspect.

```
lClock >> initClockFrame
  "draw small lines around clock frame"
  0 to: 11 do: [ :items |
    | target |
    target := (items * Float pi / 6) cos @ (items * Float pi / 6)
    sin.
    items % 3 == 0
      ifTrue: [ self addChild: (BlElement new
          geometry: (BlLineGeometry from: center + (target * 115)
    to: center + (target * 150));
          outskirts: BlOutskirts centered;
          border: (BlBorder paint: Color black width: 8)) ]
      ifFalse: [ self addChild: (BlElement new
          geometry: (BlLineGeometry from: center + (target * 125)
    to: center + (target * 150));
          outskirts: BlOutskirts centered;
          border: (BlBorder paint: Color black width: 6)) ] ]
```

7.3 Using elements to add a cross

We can apply the same technique that presented in previous section to define the backside of our card. We start by drawing a line. To draw a line we should use the `BlLineGeometry`. At the end, we create two lines and therefore two elements with a line geometry that are added as children of the `MGCardElement`.

Bloc uses parent-child relations between its elements thus leaving us with trees of elements where each node is an element, connected to a single parent and with zero to many children.

A line is defined between two points given as parameters of the `from:to:` message of the `BlLineGeometry` class. Lines created using `BlLineGeometry` are a bit special because they are 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 this is why we redefine them to be centered

```
MGCardElement >> buildFirstLine
  | 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.

7.4 Full cross

Now we can add the second line to build a full cross.

```
MGCardElement >> buildSecondLine

  | 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
```

We redefine the back side element using the two lines.

```
MGCardElement >> buildBackSide

  backElement := BlElement new
    addChildren: { self buildFirstLine .  self buildSecondLine };
    constraintsDo: [ :c |
      c horizontal matchParent.
      c vertical matchParent ]
```

Then we make sure that we invoke the `buildBackSide` method in the card creation method `card:`.

```
MGCardElement >> card: aCard
  card := aCard.
  self fillUpFrontElement.
  self buildBackSide.
  card isFlipped
    ifTrue: [ self showFrontFace ]
    ifFalse: [ self showBackFace ]
```

Once this method is defined, refresh the inspector and you should get a card as in Figure 7-1.

Figure 7-1 Using crossed back side cards.

The backside is then an `BlElement` 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.

7.5 Conclusion

We show that you can have different ways to define a graphical representation of your element.