cards

Design Patterns Team

November 18, 2015

Contents

1 Card shape

- Rectangle, circle, etc
- Corners, rounded, sharp?

2 Decal

- Generic art, gets an art object
- Decals will need to be patterned, one decal might want to be on each corner, or might want to go in a row, would be very awkward to have to manually do all this patterning : Decal shouldn't do patterning, let the transformation class do it.

2.1 Decal types

- Image decals
- Number decals
- Text decals
- Shape decal

2.2 Nesting

Layouts will want to be nested, a text box might want to be on top of a back ground.

2.3 Layout class

Composite, leafs hold decal.

- Could then handle patterning
- Composite

2.3.1 General patterns

- Textbox
- Border
- Etc?

3 Theming

- Sub decks
- Some sets will have the same decal applied to the same spot
- Others will have the same decal, but used in a different spot per card
  - EX) Cards have some number of $family decals, but those decals are in different spots, they don't know what image they are until we tell it its family. Thus, would say something like $family = spades, and all the $family decals will use the spades image.

    ```
    clone =  prototype.clone();
    clone.setDecal("family",spades);
    ```

    Families will still need per card information. So perhaps...

    ```
    clone.setCharacter(charizard.jpg)
    clone.setHP(120);
    ```

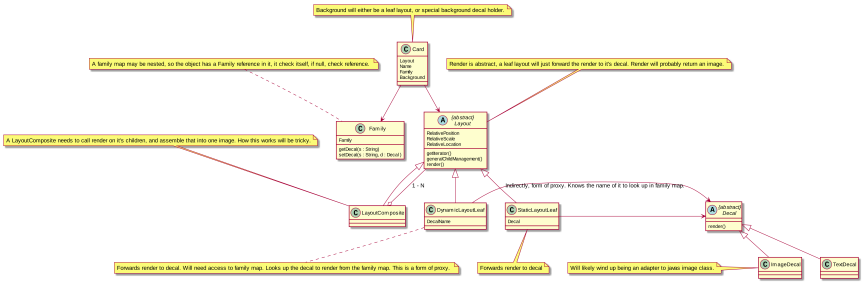
4 Cards

Card has a layout. Layouts can be cloned. Cards can be cloned. Everything is immutable, so not anymore really.

- Card will have a name.
- Card will have a layout, front and back.
- Card have a family? Family is just a map, string -> decal, Two types of layout leafs. FamilyLeaf, which just has a string, asks its family. DecalLeaf, which holds a decal.
  - Cards will not have a family, family given to render.
  - A family can be nested, will query its map, then parent map.
    - Families aren't nested, instead render takes in a list, checks all, uses first it finds.

5 Example UML

This should probably be split into multiple diagrams.



- Some patterns used here
  - Proxy : the dynamicleaflayout
  - Composite : The layouts
  - Adapter : Image Decal
  - Iterator : For the composite

6 Older javascripty idea

6.1 Scripting

6.1.1 Configuration file

```
decal = ...; //Some image file...
LeafDecalFoo = { "position" : ..., "decal" : decal }
LeafDecalBar = { "position" : ..., "decal" : "suite" }
positionA = [ 50, 100, 50, 100 ]
positionB = [ 50, 100, 50, 100 ]
LayoutB = { "position" : positionB, "leafa":LeafDecalBar, "leafb":LeafDecalFoo };

LayoutC = {
    "position" : positionA;
    "layoutA" : {"position" : positionA, "foo" :LeafDecalFoo, "bar":LeafDecalBar },
    "layoutB" : LayoutB
}

LayoutD = {
    "position" : positionB,
```

```
    "layoutA" : LayoutC, //"layoutA" is a clone of LayoutC
    "layoutB" : LayoutC["layoutA"] //"layoutB" is a clone of LayoutC["layoutA"]
}
LayoutD["layoutA"]["position"] = [0,100,0,100];


//Layouts are always cloned! But they point to the same, immutable decals,
//so they are cheap to clone.
backgroundDecal = ..;
heartDecal = ...;
clubDecal = ...;
familyBackground = {
    family : null,
    "background" : backgroundDecal
}
heartFamily = {
    family : familyBackground,
    "suite" : heartDecal
}

clubFamily = {
    family : familyBackground,
    "suite" : clubDecal
}

CardOne = {
    "name" = "cardA",
    "LayoutFront" = LayoutD,
    "LayoutBack" = LayoutC,
    family = heartFamily
}
```

6.1.2 Script

```
CardSetA = [CardOne,CardOne];
CardSetB = CardOne.clone(10);
CardSetA.setFamily(clubFamily);
CardSetA.add(CardOne);
```

7 Example with new lisp for 52 cards

```
spadeImage = Image("spadeImage.jpg")
heartImage = Image("heartImage.jpg")
diamondImage = Image("diamondImage.jpg")
clubImage = Image("clubImage.jpg")

jackImage = Image("jack.jpg")
queenImage = Image("queen.jpg")
kingImage = Image("king.jpg")

whiteRectangle = Rectangle("white")
wholeCard = Scaled-Position(0,0,100,100)
cardSize = Size(1000,1000)
//..Moreposiitons

ace = character(A) //This needs to be replaced with string.
two = character(2)
three = character(3)
four = character(4)
five = character(5)
six = character(6)
seven = character(7)
eight = character(8)
nine = character(9)
10 = character(10)
J = character(J)
Q = character(Q)
K = character(K)

(define emptyFamily (family))
(define heartFamily (family (cons "suite" heartImage)))
(define diamondFamily (family (cons "suite" diamondImage)))
(define spadeFamily (family (cons "suite" spadeImage)))
(define clubFamily (family (cons "suite" clubImage)))

(define backgroundLayout
  (layout
  (cons (leaf-layout whiteRectangle) wholeCard)
  (cons (leaf-layout "suite") topRightCorner)
```

```
  (cons (leaf-layout "suite") bottomLeftCorner))
)

(define jackLayout
  (layout
  (cons backGroundLayout wholeCard)
  (cons (leaf-layout J) topRightCorner)
  (cons (leaf-layout J) bottomLeftCorner)
  (cons (leaf-layout jackImage) center))
)

(define queenLayout
  (layout
  (cons backGroundLayout wholeCard)
  (cons (leaf-layout Q) topRightCorner)
  (cons (leaf-layout Q) bottomLeftCorner)
  (cons (leaf-layout queenImage) center))
)

(define kingLayout
  (layout
  (cons backGroundLayout wholeCard)
  (cons (leaf-layout K) topRightCorner)
  (cons (leaf-layout K) bottomLeftCorner)
  (cons (leaf-layout kingImage) center))
)

(define KingCard
  (card cardSize "King" kingLayout kingLayout)
)

(define JackCard
  (card cardSize "Jack" JackLayout JackLayout)
)

(define QueenCard
  (card cardSize "Queen" QueenLayout QueenLayout)
)

(define oneLayout
```

```
  (layout
  (cons backGroundLayout wholeCard)
  (cons (leaf-layout one) topRightCorner)
  (cons (leaf-layout one) topLeftCorner)
  (cons (leaf-layout "suite") center)
)

(define oneCard (card cardSize "One" oneLayout oneLayout))
...

(define tenLayout
  (layout
  (cons backGroundLayout wholeCard)
  (cons (leaf-layout ten) topRightCorner)
  (cons (leaf-layout ten) topLeftCorner)
  (cons (leaf-layout "suite") center-10-1)
  (cons (leaf-layout "suite") center-10-2)
  (cons (leaf-layout "suite") center-10-3)
  (cons (leaf-layout "suite") center-10-4)
  (cons (leaf-layout "suite") center-10-5)
  (cons (leaf-layout "suite") center-10-6)
  (cons (leaf-layout "suite") center-10-7)
  (cons (leaf-layout "suite") center-10-8)
  (cons (leaf-layout "suite") center-10-9)
  (cons (leaf-layout "suite") center-10-10)
)

(define tenCard (card cardSize "Ten" tenLayout tenLayout))


(define suite (list QueenCard JackCard KingCard oneCard ... tenCard))

(render suite heartFamily)
(render suite clubFamily)
(render suite diamondFamily)
(render suite spadeFamily)
```

8 Render method

The idea is that we

- Calculate a size for our image, call this MainImage.
  - Calculate the size of this layout with the size given to us of parent, plus our relative width + height to that
- If a composite, then
  - Make MainImage a transparent image with the earlier calculated size.
  - For each child, call render again, passing in the position it is paired with in the composite
    - Pass through the family, and the subsize.
  - For each child, take the image retrieved from the above step, and place it in the MainImage.
    - It will be placed at MainImage.size().x*x%, MainImage.size().y*y%
  - Return MainImage

```
class CompositeLayout {
    //...Stuff elided
    Image render(ArrayList<Family> families, PositionScaled position, Size size) {
        Size sub_size = size;
        sub_size.x*= (position.width/100);
        sub_size.y*= (position.height/100);
        Image image = new Image(sub_size.x,sub_size.y);
        Size sub_size = size;
        for (LayoutAndPositionHolder pair : this.data) {
            SubImage = pair.layout.render(families,pair.position,sub_size);
            image.insertImage(subImage,
                              pair.position.x*(sub_size.x/100),
                              pair.position.y*(sub_size.y/100));
        }
        return image;
    }

}
```

- If a LeafLayout

  - Tell the decal to render at that calculated size.
  - Return this image

```
class LeafLayout {
     //... Stuff ellided

     Image render(ArrayList<Family> families, PositionScaled position, Size size) {
         Size sub_size = size;
         sub_size.x*= (position.width/100);
         sub_size.y*= (position.height/100);
         Image sub_image = this.decal.render(families,sub_size);
         return sub_image;
     }
```

- Note how position is always handled by the parent.

This process is started with the card, which will call

```
class Card {
    //Stuff ellided
    Foo someMethod(ArrayList<Family> families,Some params...) {
      Image image = FrontLayout.render(families,new PositionScaled(0,0,100,100),this.si
      Image image = BackLayout.render(families,new PositionScaled(0,0,100,100),this.siz
        Do something with these images...
    }
}
```

- Dynamic decal will look like this
  - Decals only need Families + size to be rendered

```
class DynamicDecal {
    private String key;

    DynamicDecal(String key) {
        this.key = key;
    }

    Image render(ArrayList<Family> families, size) {
        for ( Family family : families ) {
            Decal decal = family.get(key);
```

```
            if ( decal != null ) {
                return decal.render(families,size);
            }
        }
        //If we get here, error,
        //throw an exception or something,
        // and add an error to the user
        //error output, something like

        // "ERROR, DECAL OF NAME " + name + "Not found in given family!"
    }
}
```