Bridging the Gap Between Morphic Visual Programming and Smalltalk Code

Noury Bouraqadi[1] and Serge Stinckwich[2]

[1] Ecole des Mines de Douai, France
bouraqadi@ensm-douai.fr
[2] GREYC - Université de Caen, France
Serge.Stinckwich@info.unicaen.fr

Abstract. In this paper, we claim that both prototype-based visual programming and traditional Smalltalk class-based programming are required for developing applications with a GUI. We introduce Easy Morphic GUI (EMG), a framework that connects Morphic and EToys visual manipulation and scripting facilities with the usual Smalltalk development environment tools. The Squeak platform is used here as a playfield for our experiments. A step-by-step tutorial is used to illustrate the main aspects of the EMG framework. We also introduce two reuse operators: EMBED and CLONE in order to build new GUIs out of existing ones. EMBED inserts a GUI into another one, while CLONE makes the destination look the same as the original. Static and dynamic version of these operators are also investigated.

Keywords: Visual Programming, GUI Building, Reuse, Prototype.

1 Introduction

1.1 Context: Squeak and Morphic

Squeak [IKM+97] is a feature-rich, platform-independent and open source implementation of the Smalltalk programming environment, whose virtual machine is entirely written in Smalltalk. Squeak includes network and multimedia (sound, graphics, video, ...) support, an integrated development environment similar to others Smalltalk flavors and a constructivist learning environment for children called EToys, based on a GUI model named Morphic.

Morphic[Mal02] was invented for the Self [US87] prototype-based programming language. Self used to be the successor language to Smalltalk. The Morphic user interface was developped by Randall Smith and John Maloney. And when John left the Self project, he ported Morphic to Squeak as a replacement for MVC. Every display object (windows, menus, ...) in Morphic is a Morph, i.e an instance of a subclass of the class Morph.

Following the Smalltalk philosophy, Morphic objects are uniform: they have the same basic structure (e.g. color, position, extent) and can be manipulated in the same way. Because morphs are concrete objects, they can be manipulated visually as real objects. Each morph can be selected in order to bring up a set of icons, called *halos*, which allow visual manipulation (e.g. moving, rotating, resizing, cloning, changing colors or layout). Being robust, morphs still work after being manipulated, even if impacts their geometric properties (e.g. rotation or a resizing).

On top of Morphic we find EToys, a powerful scripting visual programming language. EToys provide an interface where the developer program by dragging and dropping together small snipets of code which directly manipulate visual objects. There is no separation between the GUI and the model. Hence, EToys open up the opportunity to explore visual programming but in a rather constrained environment (some programs are difficult if not impossible to code using EToys mainly because many objects and messages are unavailable for visual programming). EToys programming share also similarities with event-based programming or context-based programming [GNS06], where form and function are usually merged [Wes02].

EToys can be classified as a *direct manipulation interface*, a term coined in 1983 by Ben Schneiderman [Shn83]. That is a human-computer interaction style that allow users to directly manipulate objects presented to them with actions that resemble to physical one. The advantages are numerous. The user has at his disposal at a given moment only the behaviors associated with the object which he handles.

Though interesting, we believe that the power of Morphic and EToys is not fully unleashed.

1.2 Motivation

When programming in Squeak, developers are provided two programming interfaces. On the one hand Morphs can be manipulated and composed visually through halos and different pop-up menus. On the other hand, Browsers allow writing Smalltalk code. However, when it cames to building software with GUI, developers often code the full GUI as plain Smalltalk code. This task is cumbersome. Therefore, they lose the benefit of Smalltalk dynamicity and incremental programming. They have to fully code the GUI description, then create an instance. For example, if the look or the layout is not appropriated, they have to delete the GUI instance, change the code and then recreate the instance to check whether it fits the desired visual properties.

When using Morphic visual capabilities, one can instantiate some morphs and even compose them. But, the link with code has to be done in an ad hoc manner. No guidelines are provided. More importantly, the created morph description can not be stored together with code through for instance change sets or Monticello repositories. Morphic does allow exporting morphs in files, but the code is not included in those files. Squeak also allows exporting a whole project through the "image segment" concept. However, the obtained file contain all objects only accessible from the roots of the image segment. As a consequence, added classes are often ignored on serialization into image segment.

1.3 Requirements

In this paper, we introduce Easy Morphic GUI[3] (EMG) a framework to bridge the gap between, on the one hand Morphic and EToys visual manipulation and scripting facilities, and on the other hand “plain” Smalltalk coding activities. Our goal is to integrate more smoothly the UI design into the incremental style of development of Smalltalk. The key requirements for this integration are:

[3] Freely available at http://csl.ensm-douai.fr/EasyMorphicGUI

Use visual programming tools for GUI development. Building interfaces is a rather boring and time-consuming operation, because they often look similar. By dragging and dropping existing widgets, it is possible to assemble complex programs in literally seconds rather than by specifying them textually. A lot of commercial development environments like VisualBasic or Delphi already exploit this advantage for fast prototyping applications.

Use Smalltalk powerful development tools for coding business classes. "Turtles all the way down" is not necessarily a good choice for UI design: not everything should be implemented visually. Smalltalk already provides very powerful development tools (class browser, instance inspectors, ...) and we like to stick with these tools in order to implement domain specific classes.

Save GUIs together with the application code. Smalltalk also provides suitable solutions for code storage and project management, like change sets, Monticello (Squeak). But usually, preserving the user interface is not that simple. It requires either a tool that converts UI into code or one that stores code as objects (not the case for Squeak).

Support GUI versioning in order to allow roll backs during the project life-time. Designing a new UI is an incremental task that need several test and try, before the final build is obtained.

Provide operators to reuse GUIs. As we already mentioned, interfaces often look the same. So like domain code that can be reused by subclassing or composing existing classes, GUIs should be reusable in other contexts than the one where they were first defined.

In the remainder of this paper we first provide an overview of the EMG framework (section 2) where we show how EMG does satisfy the four first requirements. EMG features are illustrated by a step-by-step example. Section 3 describes how EMG satisfies the last requirement. We present GUI reuse operators and their functioning through some examples. Next, we present in section 4 work related to EMG. Finally, future works and perspectives are drawn together with the conclusion in section 5.

2 An Overview of Easy Morphic GUI

The starting point of Easy Morphic GUI (EMG) is that morphs are naturally manipulated as individual objects while Smalltalk code is mainly class centered. EMG bridges the gap between those two worlds by mixing prototype-based programming languages concepts with the class related ones. On the one hand, developers build the GUI as a prototype by visual manipulations but on the other hand, business code is implemented by coding the appropriate classes. EMG, as describe below, provides a framework that allows linking the two parts.

2.1 Description

Essential to the EMG framework is the EMGGuiMorph, the root of the GUI classes hierarchy. We call GUIs, instances of EMGGuiMorph and its subclasses. GUIs are morphs that act as containers for other morphs (called sub-morphs) dedicated to human-machine interaction. Besides, GUIs hold instance variables and methods that allow accessing business objects. Hence, they act as glue between visual objects (i.e. morphs) and business ones.

Each GUI class has a special instance called *prototype*[4]. Instances of a GUI class are created as copies of the prototype of that class. In order to build a GUI, developers subclass EMGGuiMorph and then visually setup the prototype. EMGGuiMorph extends default Morphic halos with menus that eases the creation of new morphs and their layout. Note that because we are building the GUI visually, we can use the full power of Morphic and related libraries such as EToys.

EMG relies on the *Mediator* design pattern [GHJV95] for connecting visual objects to business ones. GUIs act as a mediator that encapsulates the interaction between morphs and business objects. References from morphs to GUIs are set in an interactive way, through pop-up menus, or if not available using inspectors[5]. For example, buttons implemented by morphic developers have a menu to setup the message to be sent on a click. The menu allow choosing the message receiver among available morphs, and provide the selector and the arguments. When using EMG, the receiver should be the GUI. Of course, this means that the GUI class have to implement methods to be called for mediation.

References from business objects to the GUI, often based on the *Observer* design pattern, are set in the GUI class. This is performed either in the initialize method or through lazy initialization. Of course business objects can refer to each other directly. Symmetrically, morphs belonging to the same GUI can reference each other straightly. However, references between business objects are set within business classes code, while references between morphs are set interactively through pop-up menus or inspectors.

2.2 A First Simple Example Step-by-Step

We present here how to build a GUI for a counter using EMG. Our goal is to have a counter that can be handled through the GUI shown on figure 1. The counter is incremented by a click on the "+" button and decremented once the "-" button is selected. The text field both displays the counter's value and allows editing it.

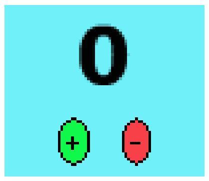


Fig. 1. The counter GUI

[4] The prototype can be accessed through a class instance variable (i.e. an instance variable declared in the metaclass).

[5] Morph the superclass of all morphs is extended with a few facility methods that allow retrieving the GUI to which a morph belongs.

```
 1  Model subclass: #EMGCounter
 2          instanceVariableNames: 'count'
 3          classVariableNames: ''
 4          poolDictionaries: ''
 5          category: 'EasyMorphicGUI-Counter Example'
 6
 7  EMGCounter >> count
 8          ↑ count
 9
10  EMGCounter >> count: newValue
11          count := newValue.
12          self changed
13
14  EMGCounter >> initialize
15          super initialize.
16          self count: 0
17
18  EMGCounter >> increment
19          self count: self count + 1
20
21  EMGCounter >> decrement
22          self count: self count - 1
```

Fig. 2. Definition of the counters class

The business code for our example is a trivial counter class. The corresponding code is provided by figure 2. Nothing special to mention except that the EMGCounter class inherits from Model. It notifies its dependents when the count instance variable changes (see line 12). Our goal is to have the GUI be registered as a dependent and be updated when the counter changes.

Figure 3 provides the definition of EMGCounterGUI, the GUI class for our counter. We can see that EMGCounterGUI inherits from EMGGuiMorph the support for prototype management and related operators (see section 2.3 for more details about EMGGuiMorph). The link to counter is done through an instance variable (line 2) which is set on creation time.

The counter instance variable setter (lines 10–15) registers the GUI as a dependent of the counter. Then, it updates the display through the updateDisplay message. The updateDisplay message is part of the EMG API. It is sent by the EMGGuiMorph»update: method in order to keep the GUI display coherent state with business objects. In our example, the updateDisplay message ensures that the text field named countText displays the actual value of the counter. The text field is retrieved using the submorphRecursivelyNamed: message (see line 25). This message is implemented by the EMGGuiMorph based on the Null Object Pattern [Woo96]. This pattern allows displaying and testing GUIs in early development stages. Indeed, since the GUIs are build visually, they are empty when prototypes are first created. However, a GUI class may contain some references to submorphs using the

```
EMGGuiMorph subclass: #EMGCounterGUI
          instanceVariableNames: 'counter'
          classVariableNames: ''
          poolDictionaries: ''
          category: 'EasyMorphicGUI-Counter Example'

EMGCounterGUI >> counter
          ↑counter

EMGCounterGUI >> counter: newCounter
          counter ifNotNil: [counter removeDependent: self].
          counter := newCounter.
          counter ifNotNil: [
                    counter addDependent: self.
                    self updateDisplay]

EMGCounterGUI >> createCounter
          ↑EMGCounter new

EMGCounterGUI >> initialize
          super initialize.
          self counter: self createCounter

EMGCounterGUI >> countTextField
          ↑self submorphRecursivelyNamed: #countTextField

EMGCounterGUI >> decrease
          self counter decrement

EMGCounterGUI >> increase
          self counter increment

EMGCounterGUI >> countFromText: aText
          | newCount |
          newCount := aText asString asInteger.
          newCount ifNil: [↑self].
          self counter count: newCount.
          self flash

EMGCounterGUI >> updateDisplay
          super updateDisplay.
          self countTextfield
                    contents: self counter count printString
```

Fig. 3. Definition of the counter's GUI class

submorphRecursivelyNamed: message (cf. line 25 of fig 3). If no submorph holds the provided name, the answer of this message is an object instance of EMGUndefinedMorph. This class redefines the doesNotUnderstand: method in order to notify the developer that a message is not understood. Developers can ignore these messages and proceed with the execution.

Class EMGCounterGUI also provides methods for user interaction. Methods increase (lines 30–31) and decrease (lines 27–28) implement actions to perform when clicking on buttons. Method countFromText: (lines 33–38) aims at updating the counter when the text field content is modified. In order to notify the user that the modification is recorded, we make the counter GUI flashes (line 38).

Now we can build visually the GUI. The following expression allows displaying the counter's prototype.

EMGCounterGUI openPrototypeInWorld.

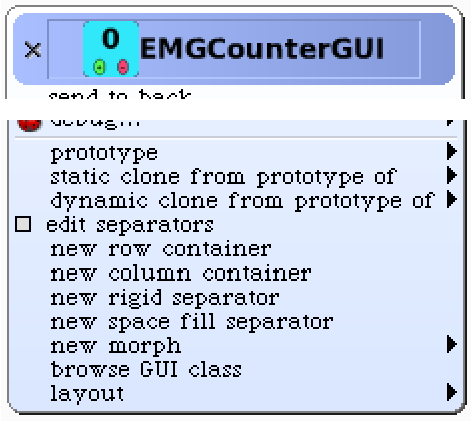


Fig. 4. Morphic halos are extended with menus to build and manage the GUI's prototype

On creation, the prototype is but an empty rectangle. The missing buttons and text field can be created from the World menu[6] or the "Parts Bin"[7] and dropped into the prototype. Alternatively the GUI morphic halos can be used (see figure 4). We have extended them with menus helping the construction of GUIs. Besides, we rely on Squeak support for visual operations undo.

Created morphs have to be setup to finish linking them to business code. We rely for this on existing morphic capabilities. For example, the name[8] of the text morph has to be set to "countTextField" (Figure 5-a). The label, action selector and target of buttons for counter incrementing / decrementing can be set through morphic halos menus (Figure 5-b). Other properties such as colors and layout can be setting in a similar way. Once terminated, the prototype has to be saved through our extension of morphic menus (Figure 5-c).

[6] Sub-menu "new morph..."
[7] Parts Bin is a visual repository of existing Morphs that can be opened from the World menu, sub-menu "objects".
[8] Actually, it is the morph's "external name".

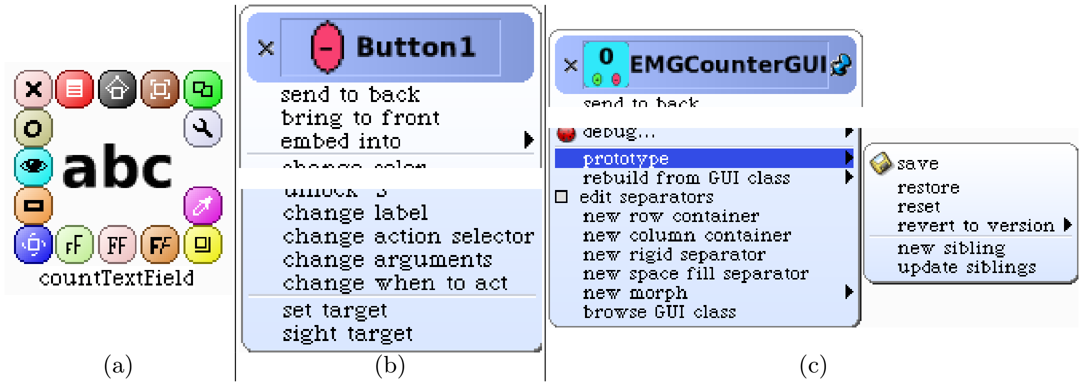


Fig. 5. Setting up morphs for the counter's GUI

2.3 GUI Prototype Management

As said above, GUIs are built based on a prototype. We rely then on the "Prototype" design pattern. So, new instances are created by deep-copying the prototype.

The "Prototype" design pattern leaves the developers free about where to store prototypes and how to copy them. In our case, we choose to have prototypes stored at the class level. The metaclass of EMGGuiMorph does define an instance variable to reference the prototype. The new method is redefined to use this instance variable and copy the prototype for creating new instances.

However, referencing the prototype in class instance variables is not enough. Indeed, the prototype is lost on file outs or when we commit with project management system like Monticello. To avoid this situation, we introduced a save–restore mechanism accessible through the prototype's morphic menu (see Figure 5-c). It does rely on storing a serialized form of the prototype in a class method. To avoid the literals count limits fixed by the Smalltalk compiler, the byte array produced by the serialization is compressed and stored as a string. Therefore, when filing in a GUI class, the prototype can be restored by retrieving the prototype bytes and deserializing them.

An interesting side effect of this storage solution is GUI automatic version management. We rely here on Squeak automatic method versioning system to provide GUI developers with the ability revert back to old versions. This feature together with Morphic undo support contributes to make GUI development even more comfortable.

3 GUI Reuse

When building interactive software, developers have to deal with at least three concerns: the business, the GUI and their interactions. The business concern is implemented through objects such as the counter in our previous example. The GUI concern is implemented through morphs. Interactions between morphs and business objects reified as mediators instances of EMGGuiMorph and its subclasses.

Through the business – mediator – GUI decomposition, EMG enables separation of concerns when building some software. Development can be separated in time and distributed between different developers. The most appropriate tools can be used for each part such as visual operations for GUI and browsers for the rest. EMG does also enable separation of concerns when it come to reuse. Business or mediator classes can be reused through usual reuse operations, and particularly inheritance. In this section, we focus on GUI reuse.

3.1 GUI Reuse Operators

Reuse operators allow building new GUIs out of existing ones. Our proposal is based on two binary operators: EMBED and CLONE.

Each one of these operators requires two operands. The first one, called *source*, is the GUI to reuse. It remains unchanged once the construction is over. The second operand, called *destination*, is the GUI that is being built. Its appearance including the set of submorphs it contains is changed by the reuse operator. It is important to stress that the two operands may be instances of unrelated classes. The only constraint is that they should be GUIs, i.e. their classes should inherit from EMGGuiMorph.

The EMBED operator inserts the source GUI into the destination. As opposite to CLONE which totally changes the destination appearance, the EMBED operator extends the destination. The only modification of the destination is the addition of the source GUI as a submorph. Other destination submorphs, its color and its size to name a few, are not changed. Therefore, if EMBED is applied several times on the same destination with different sources, the destination will include all sources.

The CLONE operator makes an already existing destination GUI have precisely the same appearance as the source one (e.g. same color, dimension, layout and submorphs[9]). If CLONE is applied several times on the same destination with different sources, the destination GUI will end up with same appearance as the last used source GUI.

We distinguish two variants for the CLONE operator: Static-CLONE and Dynamic-CLONE. The Static-CLONE variant does only perform the cloning. The Dynamic-CLONE operator goes beyond. In addition to cloning, it also sets up a dependency link between source and destination GUIs. Whenever source appearance evolves, the destination is updated.

We have chosen not to propose a dynamic variant for the EMBED operator. This decision is based on the observation that an embedded GUI is often visually adapted to fit with its container and other morphs. An unsupervised update may break the appearance of the container. However, developers still can do updates manually through our extension of morphic halo menus.

For the same reason, we chose to keep by default the prototype–sibling relationship "static". New instances of a GUI class are created by cloning the class's prototype using the Static-CLONE operator. However, prototypes provide a morphic menu to update all their siblings. Symmetrically, developers can update a single GUI (through a menu) from its prototype. They also can also make the GUI become a "dynamic clone" of the prototype.

[9] Submorphs are copied.

Actually, these operations are not restricted to the prototype of the same class. A GUI can changed to resemble the prototype of any other class.

3.2 Examples

Reusing the counter GUI with the CLONE Operator In this example, we show how we construct a circular counter GUI by cloning it. Figure 6 provide the implementation of the EMGCircularCounter class. Our goal is to build a simple GUI for circular counters that has exactly the same appearance and interactions as the GUI for plain counters. Therefore we simply create a subclass of EMGCounterGUI which uses EMGCircularCounter as shown in figure 7. Last, we make the prototype of EMGSimpleCircularCounterGUI be a dynamic clone of the prototype of EMGCounterGUI by mean of a menu. Figure 8-a shows the menu. The resulting circular counter GUI is presented on figure 8-b. It is worth noting that the inheritance link between EMGSimpleCircularCounterGUI and EMGCircularCounter, and the clone link between their prototypes are totally decoupled. A prototype of another class can be used as a source for the CLONE operator.

Reusing the counter GUI with the EMBED Operator In this example, we build a simple alarm. Its GUI is constructed by embedding two simple circular counters GUI described in section 3.2. Again we start by implementing business object which correspond here to the EMGAlarm as shown in figure 9.

Next we implement the gui class EMGAlarmGUI as shown on figure 10. The run method is performed by a button in the GUI (see figure 12). Besides the business part which is setting up and starting the alarm, this method also makes some actions on the GUI. The run button is locked and a colon label blinks until the alarm rings (i.e. when the run message to the alarm returns).

Blinking is not part of label morph's behavior. We introduced it using EToys to demonstrate the use of visual programming in EMG and how to link it to the GUI code. Scripts we implemented for this example are shown on figure 11.

Finally, the alarm GUI prototype is built by embedding instances of EMGSimpleCircularCounterGUI. This operation is performed simply through a drag and drop thanks to Morphic visual operations (see figure 12-a). The resulting GUI is shown on figure 12-b.

4 Related Work

For building GUIs, many generic and abstract models, were already proposed in the literature. The main aim of these models is to obtain a better comprehension of the existing interactive systems, and to define suitable software architectures for the development of new systems. All models define an interactive system with two components: an interface component and an applicative component. UI models could be classified in five main classes: linguistics models, interactors models, direct manipulation interfaces and hybrid approaches.

```
EMGCounter subclass: #EMGCircularCounter
        instanceVariableNames: 'min max'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'EasyMorphicGUI-Counter Example'

EMGCircularCounter>>initialize
        self min: 0.
        self max: 9.
        super initialize

EMGCircularCounter>>valueInRange: integer
        integer > self max
                ifTrue: [↑self min].
        integer < self min
                ifTrue: [↑self max].
        ↑integer

EMGCircularCounter>>count: newValue
        | actualNewValue |
        actualNewValue := self valueInRange: newValue.
        super count: actualNewValue

EMGCircularCounter>>max
        ↑ max

EMGCircularCounter>>max: newMax
        max := newMax

EMGCircularCounter>>min
        ↑ min

EMGCircularCounter>>min: newMin
        min := newMin

EMGCircularCounter class>>min: min max: max
        ↑self new
                min: min;
                max: max;
                yourself
```

Fig. 6. Implementation of circular counters

```
EMGCounterGUI subclass: #EMGSimpleCircularCounterGUI
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'EasyMorphicGUI−Counter Example'

EMGSimpleCircularCounterGUI>>createCounter
        ↑EMGCircularCounter min: 0 max: 9
```

Fig. 7. Implementation of simple GUIs for circular counters

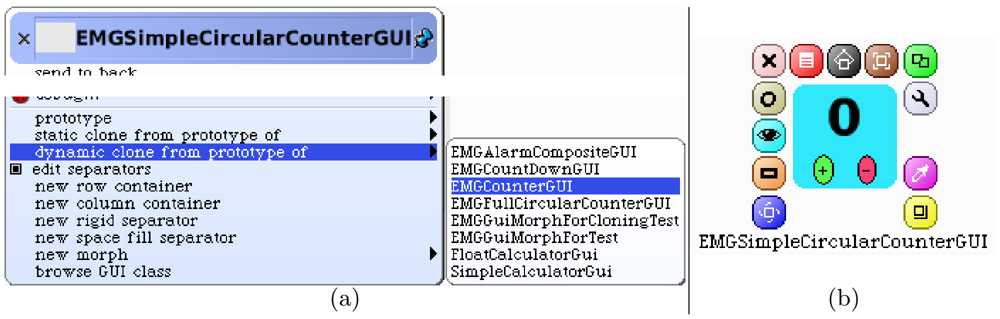


Fig. 8. Building circular counter's GUI by dynamic cloning

Linguistics models [Pfa83] [UIM92] are based on a linguistic approach of the interaction which identifies three key aspects: 1) lexical aspects indicate all things that can be assimilated to an input (click, drag and drop) or output (icons) vocabulary. 2) syntaxic aspects indicate grammars of input representing valid sequences of actions, or the space and temporal aspects of the display. 3) semantic aspects correspond to the functional part of the application, which lastly determines the meaning of an action and generates errors.

Interactors models that carry out the separation of UI concerns in several objects.

MVC (Model View Controller) is the most known and one of the oldest patterns in UI development. The most influential aspect of this pattern was the clear separation between domain objects that model the real world and presentation objects, seen on the screen. MVC users are encouraged, first of all, to create the model classes that represent the domain layer in their application. However, though very powerful, we find that MVC main drawback is that views are static entities. Developers manipulate view descriptions (i.e. code) instead of live objects. Therefore interactive development is difficult if not impossible.

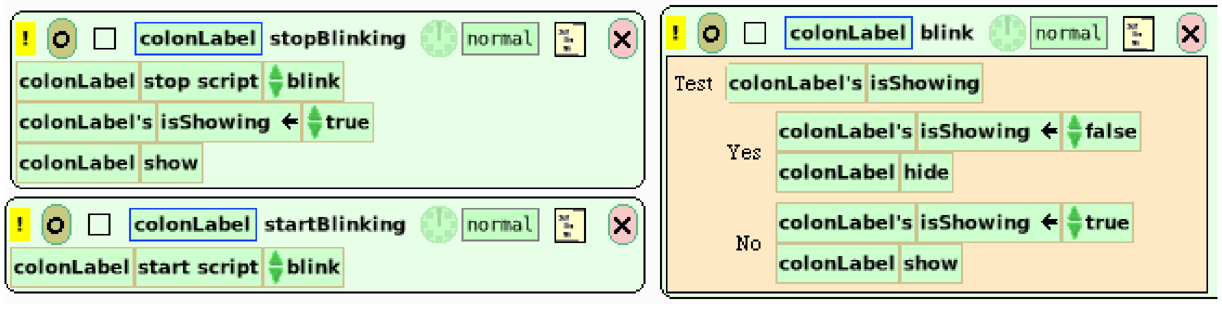
VisualWorks' Application Model VW provides a variant of MVC. It introduces an intermediate "application model" between views and models. An application model consists of a behavior that is required to support the user's interaction. For example, the information in the model may represent a selection in a menu, or the contents

```
Model subclass: #EMGAlarm
        instanceVariableNames: 'hour minute'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'EasyMorphicGUI-Alarm Example'

EMGAlarm>>initialize
        super initialize.
        self hour: 0 minute: 0

EMGAlarm>>hour: newHour minute: newMinute
        hour := newHour.
        minute := newMinute.
        self changed

EMGAlarm>>hour
        ↑hour

EMGAlarm>>minute
        ↑minute

EMGAlarm>>isTimeToRing
        | now |
        now := Time now.
        ↑now hour = self hour and: [now minute >= self minute]

EMGAlarm>>ring
        AbstractSound stereoBachFugue play

EMGAlarm>>run
        | delay |
        delay := Delay forSeconds: 1.
        [self isTimeToRing] whileFalse: [delay wait].
        self ring
```

Fig. 9. Implementation of alarm

```
EMGGuiMorph subclass: #EMGAlarmGUI
        instanceVariableNames: 'alarm'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'EasyMorphicGUI-Alarm Example'

EMGAlarmGUI>>initialize
        super initialize.
        self alarm: EMGAlarm new

EMGAlarmGUI>>alarm
        ↑ alarm

EMGAlarmGUI>>alarm: newAlarm
        alarm ifNotNil: [alarm removeDependent: self].
        alarm := newAlarm.
        alarm ifNotNil: [alarm addDependent: self].

EMGAlarmGUI>>hourCounter
        | hourCounterGUI |
        hourCounterGUI := self submorphRecursivelyNamed: #hourCounter.
        ↑hourCounterGUI counter

EMGAlarmGUI>>minuteCounter
        | minuteCounterGUI |
        minuteCounterGUI := self submorphRecursivelyNamed: #minuteCounter.
        ↑minuteCounterGUI counter

EMGAlarmGUI>>updateDisplay
        super updateDisplay.
        self hourCounter count: self alarm hour.
        self minuteCounter count: self alarm minute

EMGAlarmGUI>>run
        | runButton colonLabelPlayer |
        self alarm
                hour: self hourCounter count
                minute: self minuteCounter count.
        runButton := (self submorphRecursivelyNamed: #runButton).
        runButton lock.
        colonLabelPlayer := (self submorphRecursivelyNamed: #colonLabel) player.
        colonLabelPlayer startBlinking.
        [self alarm run.
        runButton unlock.
        colonLabelPlayer stopBlinking.
        ] fork
```

Fig. 10. Implementation of alarm's GUI class



```
colonLabel stopBlinking normal
colonLabel stop script blink
colonLabel's isShowing ← true
colonLabel show

colonLabel startBlinking normal
colonLabel start script blink

colonLabel blink normal
Test colonLabel's isShowing
  Yes colonLabel's isShowing ← false
      colonLabel hide
  No  colonLabel's isShowing ← true
      colonLabel show
```

Fig. 11. EToys scripts used for alarm GUI

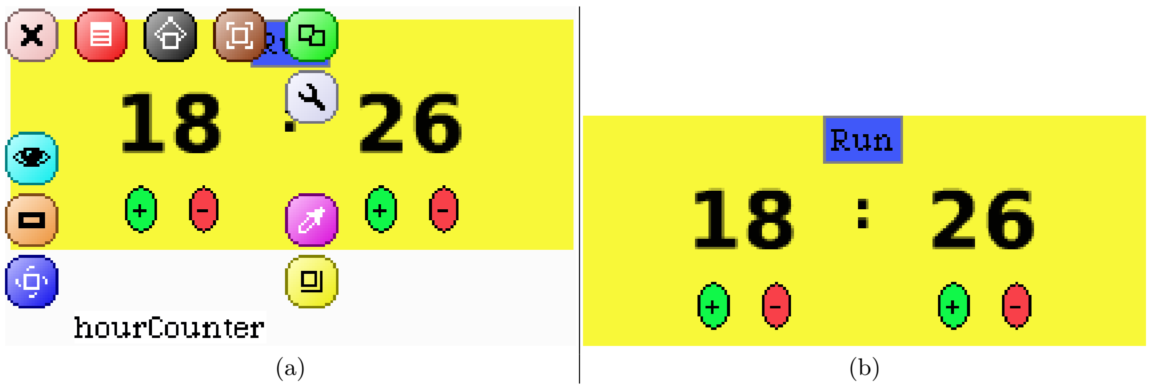


Fig. 12. Building circular counter's GUI by dynamic cloning

of a paste buffer. Widgets do no more observe domain objects directly, instead they observe the application model.

MVP (Model-View-Presenter) Dolphin Smalltalk use the MVP pattern, a derivative of the MVC pattern, first introduced in C++ by Taligent [Pot96]. Its aim is to provide a cleaner implementation of the Observer connection between Application Model and view. In MVP, the Presenter gets some extra power compared to Controller. Its purpose is to interpret events and perform any sort of logic necessary to map them to the proper commands to manipulate the model in the intended fashion. Most of the code dealing with how the user interface works is coded into the Presenter, making it much like the "Application Model" in the MVC approach.

Direct manipulation interface is a user interface style that was defined by Ben Shneiderman [Shn83] whose intention is to allow a user to directly manipulate objects presented to them, using actions that correspond to the physical world.

Widgets based UI Visual Basic (VB) and other related business frameworks use a Widget-based architecture. Developers write application specific forms that use generic controls provided by the framework. The form describes the layout of controls. By means of very simple observer mechanism, the form observes the controls and handle methods that react to interesting events raised by controls. The form is usually build with the help of a visual editor. Programming in VB consist of visually arrange controls components on a form, specify attributes and actions of those components, and write additional lines of code for adding more functionality. This is a very similar approach to the Morphic one, but the domain code is inextricably linked with the interface logic.

Naked Object Pattern With naked objects [Paw04] the domain objects are rendered visible to the user by means of a completely generic presentation layer. This user interface is automaticaly generated from an underlying business model definition. But contrary to Morphic, the emphasis is not on making objects more tangible to the programmer but rather on making them more tangible to the end-user of the system.

Fabrik Dan Ingalls' and Scott Wallace's Fabrik[Ing88] was one of the first direct manipulation of objects system in Smalltalk. Fabrik propose a kit of computational and user-interface components that can be "wired" together to build new components and useful applications.

Hybrid approaches mix several approaches in an uniform framework.

PAC (Presentation-Abstraction-Control) is another derivative of MVC that divide an UI object into three components [Cou89]: a lexical (presentation), syntactic (control) and semantic (presentation) components. Interactive components of the PAC model are based on a linguistic approach of interaction. The control component maintains a link between the presentation and abstraction components, but is also responsible to communicate with sub-UI components as the PAC model is recursive.

Tweak On the one hand, Morphic is a suitable architecture as far as direct manipulation is involved but support reusability very badly. On the other hand, MVC does'nt support direct manipulation. Tweak[10] try to combine the best of both worlds. In Tweak, each graphical object exists in a "dual" representation - as a model like object (called a "Player") and as view like object (called a "Costume"). Unfortunately, there is no available academic description of Tweak, so it's a bit difficult to understand how this integration is done.

5 Conclusion and Future Work

While coding classes does fit well developing applications business code, the development of GUIs is often cumbersome. The most natural approach is developing GUIs using visual tools. In this paper we presented Easy Morphic GUI (EMG) a framework that eases building applications with Morphic GUIs. Thanks to a few design patterns and particularly mediator and prototype, EMG does bridge the gap between Smalltalk code and Morphic–EToys visual manipulation and scripting capabilities. Developers have total freedom for building GUIs in a WYSIWYG, including using some higher level tools such as EToys. EMG does also introduce a couple of operators that encourage GUI reuse.

Regarding future work, a first one is about the generalization of the EMG framework to the whole Morphic hierarchy. Our proposal is to refactor Morphic in order to construct every morph visually. Separation between business – mediator – appearance as introduced by EMG, combined with GUI reuse operators is likely to improve reuse and modularity.

Better visual tools are also needed for morphs manipulations. So far, not all morph properties and relationships can be set through clicks and menus. Inspectors and debuggers are used as an alternative for missing feature. But this solution is not totally satisfactory.

Last, automatic code generation and update can make developers work even more easier. For example, methods for retrieving submorphs of a GUI can be automatically produced when embedding a morph into a GUI. Other connections between business objects, the mediator and GUI morphs can also be generated, making GUI construction with EMG even more easy.

References

[Cou89] J. Coutaz. Architectural models for interactive software. In Cambridge University Press, editor, *European Conference on Object-oriented Programming*, pages 382–399, 1989.

[GHJV95] Erich Gamma, Richard Helem, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GNS06] Markus Gaelli, Oscar Nierstrasz, and Serge Stinckwich. Idioms for composing games with etoys. In *The Fourth International Conference on Creating, Connecting and Collaborating through Computing (C5 2006).* IEEE Computer Society, 2006.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Allan Kay. Back to the future. the story of squeak, a practical smalltalk written in itself. In *Proceedings of OOPSLA'97*, pages 318–326, Atlanta, Georgia, October 1997. ACM.

[Ing88] Dan Ingalls. Fabrik: A visual programming environment. In ACM, editor, *OOPSLA'88*, 1988.

[Mal02] John Maloney. *Squeak: Open Personal Computing and Multimedia*, chapter 2 – Introduction to Morphic: The Squeak User Interface Framework, pages 39–67. Prentice Hall, 2002.

[10] http://tweak.impara.de/

[Paw04] Richard Pawson. *Naked objects*. PhD thesis, Department of Computer Science, Trinity College, Dublin, June 2004.

[Pfa83] G.E. Pfaff, editor. *User Interface Management Systems*. Springer-Verlag, 1983.

[Pot96] M. Potel. Mvp : Model-view-presenter : the taligent programming model for c++ and java. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf, 1996.

[Shn83] Ben Shneiderman. Direct manipulation. a step beyond programming languages. *IEEE Transactions on Computers*, 16(8):57–69, August 1983.

[UIM92] UIMS. A metamodel for the runtime architecture of an ineractive systems. In ACM, editor, *ACM SIGCHI Bulletin*, volume 24, pages 32–37, 1992.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceeding OOPSLA'87*, volume 22 of *ACM SIGPLAN Notices*, pages 227–242, 1987.

[Wes02] Bosse Westerlund. Form is function. In *DIS 2002, Serious reflection on designing interactive systems*, pages 117–124, 2002.

[Woo96] Bobby Woolf. The null object pattern. PLOP'96 Writers Workshop, 1996.