INVAIDER: A FRAMEWORK FOR VISUAL PROGRAMMING LANGUAGES

BY

JAMES MICHAEL CONTI

B.S., University of Illinois, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

A *visual language* is a language that uses pictures instead of words to convey meaning. An example of a visual language that is used everyday is the road sign language. Many road signs convey their meaning through pictures. They are used to present information that can be universally understood, and they do it in a limited amount of space. Another example of a visual language is the set of icons used to represent collapsed windows or available system tools in a software development environment. An icon visual language is very useful, because the icons give the programmer a lot of information without taking up very much of the valuable screen area.

A visual *programming* language extends a visual language by allowing the programmer to model the interaction between pictures (figures). The interaction between figures is determined by each figure's *programmed* behavior. To distinguish between the picture and what the picture represents, we will use the terms *figure* and *meta-figure* respectively. Since the users can not see meta-figures, they manipulate the figures instead. Their actions get interpreted by the visual programming language, and are performed on the meta-figures. An example of a visual programming language is a Logic Circuit Simulator (LCS). This system uses an Electronic's Computer Aided Design (ECAD) system to draw the circuit and then models its behavior. A drawing consists of figures that represent the computer chips and wires of the actual circuit. The figures drawn with the ECAD are then given meta-figures. Each figure's meta-figure defines its internal behavior and the way it interacts with other figures in the drawing.

Every visual programming language needs to provide figures that have behavior and a way to interact with these figures. The interactive functions vary between languages, but at the very least each one must provide a way to create, delete, select and move a figure. A visual language consists of two distinct parts, a graphical user interface (GUI) and an interpreter. The GUI allows the user to create and visually manipulate figures and the interpreter models the non-visual interaction between these figures. Using the ECAD system as an example, the user creates the figures for the electonic components and connects them together, but the interpreter keeps track of the current that travels through the system and models the reaction and delay time of each component. The GUI part of a visual programming language can be implemented using any decent graphical drawing package, but the interpreter part must be built into the drawing package. I have chosen to use meta-figure objects to implement the interpreter part of visual programming languages. A meta-figure object defines the behavior of a figure and contains its state, but it does not have protocol for drawing the graphical figures.

This thesis describes InVaider, a framework for visual programming languages. It is implemented in ObjectWorks\Smalltalk release 4.0. InVaider, which stands for Interactive Visual-aider, copies many ideas from an older framework called FOIBLE [Jind90]. The InVaider framework attempts to abstract the common ideas out of all visual programming languages, and provide a model that can be quickly developed into a working visual programming language. The framework contains code that implements the common features described above, and a whole lot more.

Most of the changes to the FOIBLE framework involved separating a figure's physical representation and its logical representation. This separation made it easier for InVaider to be built on top of HotDraw. HotDraw is a two-dimensional drawing package built by Pat McLaughry and Ralph Johnson for ObjectWorks\Smalltalk release 4.0 [John92]. The framework for HotDraw was created by Kent Beck and Ward Cunningham in release 2.5.

Chapter 2

HotDraw

HotDraw is the two-dimensional drawing package on which InVaider is built. HotDraw provides most of the machinery needed to create and manipulate the figures of a visual program. It provides a suite of tools that the programmer can customize to build a user-interface that is application specific.

2.1 Drawing

HotDraw allows users to create and manipulate Drawings. A Drawing is the logical window in which figures can be drawn. Each Drawing has a corresponding Drawing View that presents the Drawing to the user in a visual manner. Drawings consist of an infinite viewing area, although only a certain portion may be viewed through its Drawing View at any one time. HotDraw provides the user with a scrolling feature that can be used to move the Drawing View to a different portion of the Drawing.

A container is an object that is made up of component objects, some of which may be containers themselves [Catt91]. A Drawing is a container, and more importantly, is the root of all containers in HotDraw. The components of a Drawing are the figures that make up the Drawing. The component figures notify their containers when they move or change in some other way, and the Drawing View gets updated to reflect the changes.

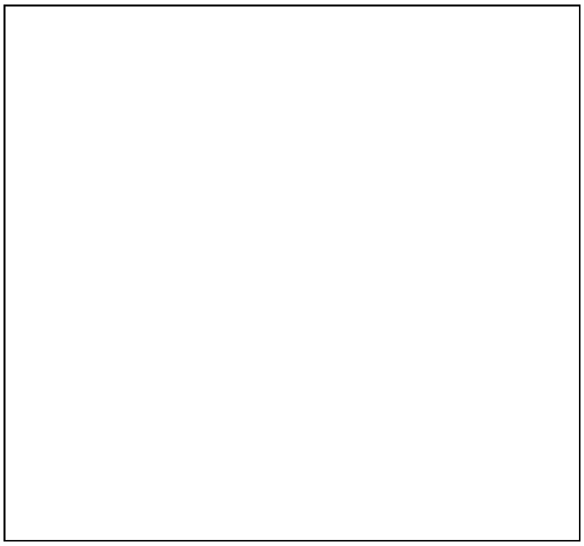

Figure 2.1: HotDraw Drawing

When a figure is added, deleted or moved in a Drawing, there is a piece of the screen that must be redisplayed to show the changes. The Drawing is responsible for keeping track of the area that needs to be updated. When a component figure notifies it of a change, the Drawing makes a note of the regions that are affected and tells its Drawing View to redisplay itself.

2.2 Figures

HotDraw figures are just two-dimensional pictures that exist within a Drawing. The Drawing shown in Figure 2.1 contains three figures, a rectangle, an ellipse and an arrow figure. They can be created, deleted, moved or altered through the user-interface. Figures know how to display themselves and how to react to changes initiated through the user-interface, but they don't have any behavior defined for them.

2.3 Tools

HotDraw provides a set of tools that receive and interpret user actions. The available tools are displayed in the tool palette on the left-hand side of Figure 2.1. The tool that is currently selected has a thick-bordered rectangle around it so that the user knows which tool is active. Some of the tools provide their own cursor, either to further remind the user which tool is active, or to aid the user in performing a specific operation, such as selecting a location at which to create a figure.

Users can select a figure from the Drawing and perform a particular operation on it, depending on the current tool. When the user wants to perform an operation on a certain figure, the tool verifies that the operation can be performed on that figure and then asks the figure to perform it. For example, if a figure is selected with the Deletion Tool, and it is a figure that can be deleted, it will be asked to remove itself from the Drawing.

Tools can be used to perform a single operation like the Delete Tool, or multiple operations, like the Selection Tool. The Selection Tool allows the user to select a figure by clicking on it, and then to move the selected figure by moving the mouse around. If the user has the shift key depressed and clicks on a figure while using the Selection Tool, an edit operation is performed instead. Some tools that perform multiple operations may require the user to select the operation from a pop-up menu instead. Other tools that perform multiple operations will perform a different operation when a different kind of figure is selected.

2.4 Meta-Figures

The only significant change to HotDraw by InVaider was the addition of meta-figures for each figure. The idea of meta-figures is based on the idea of a Model in Smalltalk's Model-View-Controller paradigm [Kras88]. Like models, meta-figures can have more than one view (figure) that represent them.

Most of the students in CS497 REJ in the fall of 1991 used HotDraw in class projects. Two of the larger projects were building a Hypercard clone and making a system called "ObjectWorld". Both of these projects added instance variables and methods to Figure, as well as making many subclasses of it. InVaider's original design was implemented this way too, until it became obvious that meta-figures were a better solution. Adding instance variables to class Figure and creating subclasses of it made the design of the visual programming language very dependent on the drawing package and did not support the use of multiple visual programming languages within a drawing package very well. The use of meta-figures allows you add a feature to every figure in the drawing without having to subclass the figure's class and it encapsulates the design of the visual programming language within the meta-figures. In addition, meta-figures allow multiple visual programming languages to be built on top of a drawing package without having conflicting results.

The use of meta-figures has made the implementation of InVaider very independent of HotDraw. Only a few two-dimensional figures had to be built for InVaider, and the rest of the time was spent creating meta-figures. The idea of meta-figures is useful and can probably be used by applications other than visual programming languages that use HotDraw. The Hypercard and ObjectWorld projects would probably have been easier to implement using meta-figures.

Chapter 3

What InVaider Can Do

A few systems were built with InVaider in order to test it, to demonstrate its capabilities, and to provide example systems that other systems could use as models. The following sections give a brief overview of the systems built using the InVaider framework. Some of these systems were originally built using the FOIBLE framework and were re-built using InVaider to demonstrate that InVaider did not lose any of the capabilities that FOIBLE originally provided.

3.1 Pert Chart

The Pert Chart Visual Language models Pert Charts. Pert Charts can be used to find the time required to finish a set of inter-related tasks, and to identify the critical tasks. For example, Figure 3.1 shows the steps needed to make a plate of nachos. The boxes in the Drawing represent the tasks that need to be completed.

Boxes receive and broadcast values through their ports. The small square on the left side of each box represents its input port and the small circle on the right side of each box represents its output port. The lines between the input and output ports are called Links. Links represent the paths on which values can be passed. A link always goes from an output port to an input port, as shown by the direction of its arrow. PertChart boxes only pass numeric values but links will pass any type of value and ports will accept and type of value. Without links and ports, boxes could not communicate with one another.

Figure 3.1: Pert Chart

There are two people working on the nachos, so some of the tasks can be done in parallel. One person cooks the beef for 15 minutes, but while that is being done, the other person can chop the vegetables (seven minutes) and grate some cheese (five minutes). During this same time, the oven is pre-heating. The mixing process can not begin until all of the ingredients are ready, so it is dependent on the three tasks that prepare them. Once the ingredients are ready, they are mixed together (eight minutes). Finally, if the oven has had enought time to pre-heat, the nachos can begin baking.

Task dependencies are shown in the Pert Chart by Links. If a task depends on previous tasks, it can not start until all of those tasks are completed. Each task in a Pert Chart has a duration assigned to it. A task's ending time is the sum of its duration and the latest ending time out of all of the tasks that it depends on. If a task doesn't depend on any other tasks, it may begin immediatly.

In many ways, a box is like a mini HotDraw Drawing. It is a container that is made up of component figures. The components of a box (box pieces) notify it when they move or change. The box in turn notifies its container, the Drawing, so that it may update its Drawing View.

3.2 FlowKit

Figure 3.2 contains a picture of FlowKit. FlowKit is a visual dataflow language that models the passing of values throughout a program. The boxes in the Drawing represent the particular operation that is being performed. The function boxes in FlowKit, such as Multiply and Add, only accept and broadcast numeric values while the Comparison boxes, such as LessThan, will accept numeric values and broadcast a boolean value. The Input boxes will accept and broadcast any value. The user must make sure that a box receives the correct type of value.

Figure 3.2: FlowKit

There are several different types of boxes shown in Figure 3.2. Each type performs a different operation, and so has different needs. For example, an Input box does not need to have a value passed to it because the user can directly edit it, so it doesn't have an input port. The Function boxes need more than one input value, so they have multiple input ports. There are even two different types of Input boxes, one with a single value and one with multiple values.

A FlowKit box requires a value from each of its input ports in order to perform its operation. When a box receives a value from each of its input ports, it sends out an output value to each of its output ports. The output ports broadcast their value to all of their links which in turn pass it to their destination box. The Pert Chart application allowed input ports to have more than one link to them, but the input ports used by FlowKit boxes only allow one.

Multiple value Input boxes are not needed since many single value Input boxes could be used instead, but they help the programmer by keeping similar data grouped together. Since there is only one Input box Creation Tool, the tool asks the user to choose the number of input values from a pop-up menu.

A FlowKit program is started by telling the input boxes to pass their value to their output ports. This causes the user's input values to be broadcast throughout the system, allowing each box to perform its operation.

3.3 Digital Logic Circuit Simulator

The Digital Logic Circuit Simulator (DLCS) is used to model computer circuits. It contains a few chip meta-figures that know how to implement the logical behavior found in the actual chips. These chips can be hooked together with wires (Links), and the resulting circuit can be simulated. Only numeric values are used in the DLCS system. Figure 3.3 shows an eight bit counter circuit with an encoder that can convert the chip's binary output to an equivalent decimal value.

Figure 3.3: Digital Logic Circuit

The boxes in Figure 3.3 are similar to the ones used in the previous two examples, except that the 4-bit counter boxes do not show the values that their output ports are broadcasting. The output ports just have labels next to them. Another difference in this example is that the 4-bit counter boxes have an output port which is linked to one of their own input ports. This *feedback* creates an infinite looping counter which might be used in a CPU to check for timeouts.

Chapter 4

InVaider Design

Figure 4.1: InVaider's Class Hierarchy

As can be seen by the examples in Chapter 3, the main figures in a visual programming language are boxes, links and ports. InVaider provides a concrete FoibleBox class, an abstract FoiblePort class and an abstract FoibleLink class, all of which are subclasses of the abstract VPLMetaFigure class (see Figure 4.1). FoibleBox and FoibleLink meta-figures are made up of components. Not only does a box have a meta-figure that defines its behavior, but each of its components has a meta-figure too. For example, a port's meta-figure has a behavior that determines what happens when a link is connected to it.

4.1 VPLMetaFigure

Figure 4.2: Relationship Between HotDraw Figures and Meta-figures

The VPLMetaFigure class is the superclass of all metaFigures in InVaider (see Figure 4.1). It has instance variables, *visualManager* and *calculationManager*, that reference supporting objects that manage its figure and performs calculation for it. A visual manager is a pluggable piece of the meta-figure that lets the programmer easily change its visual representation or lets it have more than one visual representation. If a programmer does not want to give a meta-figure a visual representation, then the *visualManager* attribute can be set to nil. A calculation manager performs any calculations that the meta-figure is responsible for.

The functionality found in the managers was initially part of the meta-figure object, but it was removed from the meta-figure and into managers for two reasons. First, not every meta-figure needs to perform calculations, for example a link just passes the value, and not every meta-figure has a visual representation, for example the A, B, C and D values passed out of the ports of the 4-bit counter chips in Figure 3.3. Second, the meta-figure can change its behavior or its figure simply by plugging in a new manager.

The other instance varibles of VPLMetaFigure, *canDelete*, *canEdit* and *canLinkTo*, are used to define behavior for the figure. VPLMetaFigure has methods *initializeBehavior* and *initializeManagers* that install the two managers and the behavior.

VPLMetaFigure class protocol for initializing:

```
    initializeBehavior
        self canDelete: self initialCanDelete.
        self canEdit: self initialCanEdit.
        self canLinkTo: self initialCanLinkTo
```

The default is to set the managers to nil (not have any), so subclasses need to override the *defaultCalculationManager* and *defaultVisualManager* methods. In addition, the behavior can be changed by overriding the methods that initialize it (see *initializeBehavior* method above).

The VPLMetaFigure class provides methods to access and assign its instance variables, along with methods such as *isBox* and *isLink* that are used to determine which type of meta-figure it is. The *token* and *token:* messages are delegated to its calculation manager and all other messages are delegated to the visual manager by the *doesNotUnderstand* method, provided the visual manager instance variable is not nil.

4.1.1 FoibleVisualManager

A visual manager provides a buffer between the HotDraw figure and the meta-figure (see Figure 4.2). Its protocol is very small, only those methods that are necessary to manipulate figures through HotDraw. Visual managers are based on the idea of a Controller in Smalltalk's Model-View-Controller paradigm [Kras88]. A meta-figure changes its figure by alerting its visual manager, which in turn tells the figure what to do. The figure that represents the meta-figure is referenced through FoibleVisualManager's *form* attribute. The FoibleVisualManager class has two methods, *removeFrom:* and *translateBy:*, that ask HotDraw to delete and move its figure respectively. An additional method, *display-*

Box, will ask its figure for the size of the retangle area that it takes up on the screen. This is used when positioning figures with respect to one another. Subclasses of FoibleVisualManager have to provide protocol for positioning their figures.

The FoibleVisualManager class has an instance variable, *metaFigure*, that is a reference to its meta-figure (see Figure 4.2). This is used by composite objects, like boxes, to retrieve its component's figures so that a composite figure can be created. The other instance variables, *canMove* and *canSelect*, are used to define the visual behavior of the figure. They are initialized with the *initialCanMove* and *initialCanSelect* methods that subclasses can override if desired.

4.1.2 FoibleCalculationManager

In the InVaider applications shown in the previous chapter, figures communicate with one another by passing values. Most applications have figures that perform some sort of calculation or transformation on the values that are passed to them, so the FoibleCalculationManager class is provided. A calculation manager is responsible for responding to the *token* and *token:* messages sent to a meta-figure by other meta-figures. For example, the calculation manager for the Pert Chart boxes in Figure 3.1 calculate the endTime of the box once it has received a *token:* from each of its inputs. After calculating it, the manager sends the new endTime value, using the *token:* message, to its output port, which in turn sends it to each of its links.

4.2 FoibleBox

A box is a composite meta-figure that is made up of component meta-figures. Figure 4.3 shows a Pert Chart box from Figure 3.1 and a FlowKit Function box from Figure 3.2. As can be seen, most boxes contain at least a name, some input and output ports and an outer shape (the rectangle around the box). The FoibleBox class provides four instance variables, *name, inputPorts, outputPorts* and *shape*, that are references to the different types of basic components. Each instance variable can be accessed through a method of

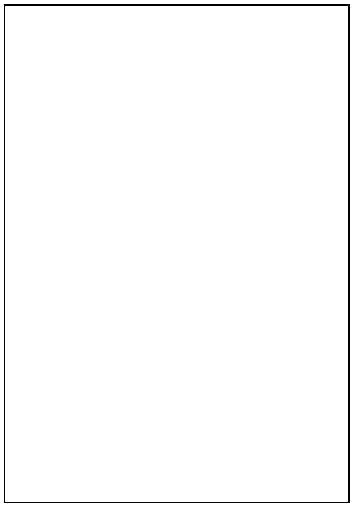

Figure 4.3: Example Boxes

the same name and can be assigned with a single-argument method of the same name. For example, the shape instance variable can be accessed and assigned using *shape* and *shape:* respectively.

The FoibleBox class provides an *initializePieces* method for initializing the components of a new box:

```
FoibleBox class protocol for initializing:
    initializePieces
        | someInputPorts someOutputPorts |
        self shape: self defaultShape.
        self name: self defaultName.
        someInputPorts := Array new: self numInputPorts.
        (1 to: self numInputPorts)
            do: [:index | someInputPorts at: index
                put: (self defaultInputPort: index)].
        self inputPorts: someInputPorts.
        someOutputPorts := Array new: self numOutputPorts.
        (1 to: self numOutputPorts)
```

```
        do: [:index | someOutputPorts at: index
            put: (self defaultOutputPort: index)].
    self outputPorts: someOutputPorts.
```

This method calls many other methods that FoibleBox provides. These methods make subclasses of FoibleBox easier to implement. The user may override them to make a custom box, or just use the provided defaults.

FoibleBox class protocol for initializing:

- **numInputPorts** - Number of input ports needed by the box (default is 1).
- **numOutputPorts** - Number of output ports needed by the box (default is 1).

FoibleBox class protocol for default pieces:

- **defaultInputPort** - Type of input port to use (default is FoibleInputPort)
- **defaultOutputPort** - Type of output port to use (default is FoibleOutputPort)
- **defaultShape** - Type of shape to use (default is FoibleShape)
- **defaultName** - Name of box (default is FoibleName)

The FoibleBox class inserts a visual manager of type FoibleBoxVisual by overriding the defaultVisualManager method of VPLMetaFigure. A box's visual manager has to build a composite box figure by collecting a figure from each of its component meta-figures (if they are visible). It refers back to the box meta-figure through its *metaFigure* instance variable to accomplish this. The FoibleBoxVisual class provides the *initializeForms* method that creates and positions the component figures and the *collectForms* method that puts them into a collection.

4.3 FoibleBoxWithValues

The FoibleBoxWithValues class extends the FoibleBox class by providing an instance variable, *values*, that stores an array of value meta-figures that are contained in the box.

A box uses values as labels, to display the result of a calculation, or to get input from the user. For example, the Pert Chart boxes in Figure 3.1 have four value meta-figures (in addition to the box name).

FoibleBoxWithValues overrides the *initializePieces* method of FoibleBox in order to install its value components, but subclasses must provide the *initializeValuePieces* method. In addition, subclasses must provide the *numberOfValuePieces* method. The values are initialized with the *initialValue* method (default is zero) unless this method is overriden.

FoibleBoxWithValues class protocol for initializing:

initializePieces

```
super initializePieces.
self values: (Array new: self numberOfValuePieces).
self initializeValuePieces
```

The FoibleBoxWithValue class also provides the following two methods to assist the subclasses in building the *values* array.

FoibleBoxWithValues class protocol for accessing:

addValue:at: - Inserts a value into the array at given location

addValue: - Inserts a value at the next available position.

4.4 FoibleBoxPiece

The abstract class FoibleBoxPiece has an instance variable, *box*, that is a reference to the box that contains it. This provides a box piece, such as a port, a way to notify its container if it changes in some way. The other instance variable of the FoibleBoxPiece class, *constraints*, will be discussed in the next chapter. As a general rule, box pieces should not be deleted, so the *initialCanDelete* method of VPLMetaFigure has been overriden to return *false*.

FoibleBoxPiece's have a visual manager, FoibleBoxPieceVisual, that provides protocol to assist the programmer in positioning them within their box. Box pieces can be positioned with respect to the box's shape, or with respect to another box piece. For example, the 'LessThan' box in Figure 3.2 has its name positioned with respect to the box's shape and its output value positioned with respect to its output port. The protocol provided is as follows:

FoibleBoxWithValues class protocol for accessing:

> **positionAt: and:** - position value with respect to box shape
> **positionAbove: by:** - position value above another box piece
> **positionBelow: by:** - position value below another box piece
> **positionBefore: by:** - position value before another box piece
> **positionAfter: by:** - position value after another box piece

4.4.1 FoibleShape

A FoibleShape is the rectangle that forms the perimeter of the box. It is used by the other box pieces to position themselves. The only function a user may perform on a FoibleShape is to resize it, using the *resizeShapeIn:* method.

If the FoibleShape's box provides a size for it, through its *shapeSize* method, then the FoibleShape's visual manager, FoibleShapeVisual, will create a rectangle figure for it automatically. If there is not a size defined, the user will have to *frame* it by choosing the top left and bottom right corner points of the retangle.

4.4.2 FoibleValue

The FoibleValue class has a single instance variable, *value*, that can have a numeric, boolean or string value in it. The applications in the previous chapter show an example of each of these types of values. A FoibleValue's value can be accessed through the *value* and *value:* methods, and sometimes through the *valueString* method. The *valueString* method converts the value to a string if it is not one already. The HotDraw figure used to display values, TextFigure, expects a string value so it uses the *valueString* method.

FoibleValue class protocol for accessing:

```
valueString
        self value isString
            ifFalse: [↑self value printString]
            ifTrue: [↑self value]
```

A FoibleValue's visual manager, FoibleValueVisual, allows the value to *grow* relative to the point at which it is positioned, using the *growLeft* and *growRight* methods. For example, the duration and endTime values in Figure 3.1 grow to the right, away from their labels, but the values in Figure 3.2 grow to the left, away from their output ports.

4.4.3 FoiblePort

The abstract class FoiblePort has an instance variable, *links*, that is a collection of the links that are attached to it. The links attribute is accessed and assigned with the *links* and *links:* methods respectively. In addition, FoiblePort provides the *addLink:* and *removeLink:* methods that can be used to modify the collection. Subclasses need to provide the *roomToLink* method that is used to limit the number of links that can be attached to it. For example, the input ports of the FlowKit boxes in Figure 3.2 can only have one link each, but the output ports can have any number of links. On the other hand, the input ports of the Pert Chart boxes in Figure 3.1 can have any number of links attached.

The concrete classes FoibleInputPort and FoibleOutputPort provide the methods *isInputPort* and *isOutputPort*, for deciding what type of port it is. These methods are used by the LinkTool to ensure that links go from output ports to input ports. In addition, they define the *roomToLink* method. The FoibleInputPort class only allows one link to be attached to its port, and the FoibleOutputPort class allows any number.

4.5 FoibleLink

Figure 4.4: Moving a Link Piece

A link is a composite meta-figure, like a box, except that all of its components are of the same type, FoibleLinkPieces. The FoibleLink class provides two instance variables, *source* and *destination*, that are references to the output port and input port respectively, between which the link exists. In addition, there is an instance variable, *linkPieces*, that is a collection of the FoibleLinkPieces that make up the link.

FoibleLinkPieces are related to one another. The two instance variables *source* and *destination* reference the link pieces to which they are connected. As can be seen in Figure 4.4, moving one link piece affects the link pieces that are attached to it. If a link piece is connected to one of the ports, its appropriate instance variable will be set to nil. These pieces can not be moved because they are anchored at the port. Their visualManager's *canMove* attribute will be set to false. The FoibleLinkPiece class provides two methods, *isFirstLinkPiece* and *isLastLinkPiece*, to support this.

4.6 FoibleProgramEditor

The FoibleProgramEditor is a subclass of HotDraw's DrawingEditor class. It has two responsibilities: to insert InVaider's basic tools into the tool palette and to provide a way to start up an application. It has three class methods defined for it, *defaultTools*, *startUp* and *startUp:*, that perform these resposibilities. Applications can insert their own tools into the palette by subclassing the *defaultTools* method and can put their name on the Drawing using the *startUp:* method (see Figure 3.1 and Figure 3.2). The *startUp* method must be provided by the subclasses to pass the application's name to the *startUp:* method.

Chapter 5

Details of InVaider

5.1 VPLMetaFigure

The *doesNotUnderstand:* method in the VPLMetaFigure class has been redefined so that it will relay any messages that it or its superclasses do not understand to its visual manager, if it has one [Foot89, Lalo90, Gold83]. This makes the FoibleVisualManager class act like a superclass of the VPLMetaFigure class, and gives us psuedo multi-inheritance. It is not exactly like multiple-inheritance, though, because the Visual Manager's class is not searched immediately after the VPLMetaFigure class fails to locate the method. Smalltalk will search all of the superclasses of the VPLMetaFigure class, and then start to look in the visual manager's class [Gold83]. This process is quite slow and degrades the performance of the application. The following VPLMetaFigure method implements the reflection.

VPLMetaFigure class protocol for reflecting:

```
doesNotUnderstand: aMessage
        | result |
        self isVisible ifTrue: [result := self visualManager
            perform: aMessage selector withArguments: aMessage arguments].
```

```
result isNil ifTrue: [↑super doesNotUnderstand: aMessage].
↑result
```

5.2 FoibleBoxPiece

As stated earlier, FoibleBoxPieces can be positioned with respect to the box's shape or with respect to other box pieces. The *positionAt:and:* method, shown below, positions the a box piece with respect to the top left corner (called the figure's origin) of the shape. It takes two arguments, an x and y coordinate.

FoibleBoxPieceVisual class protocol for positioning:

positionAt: xAmount and: yAmount

```
| aDisplayBox xPos yPos |
aDisplayBox := self metaFigure box displayBox.
xPos := self convertToAbsolute: xAmount of: aDisplayBox width.
yPos := self convertToAbsolute: yAmount of: aDisplayBox height.
self position: aDisplayBox origin + (xPos @ yPos).
↑self position
```

Arguments that are greater than one are interpreted differently than arguments that are less than one. If an argument is greater than one, it is taken to mean the absolute number of pixels to move from the origin point. For example, if the x coordinate was 5, the box piece would be positioned 5 pixels to the right of the origin. If an argument is less than one, it is taken to mean a percentage of the width or height of the box shape. For example, if the x coordinate was 0.25, the box piece would be positioned one quarter of the shape's width from the origin. This idea came from how Objectworks\Smalltalk release 4.0 builds composite windows [Parc90].

Often times, the pieces within a box are related to one another. For example, suppose an input port has a FoibleValue next to it that displays its value, when the input port receives a new value from its link, it has to tell its value to update itself. This type of *constraint* is set up when the box is collecting its box pieces. The FoibleBoxPiece class provides an instance variable, *constraints*, that is used to hold its collection of constraints.

A Constraint consists of a key, a receiver and a selector. The key allows the box piece to choose only certain constraints to execute, for example only the constraints that need to be executed when a new value arrives. The receiver is the object that needs to be notified and the selector is the method to send to the receiver. The following FoibleBoxPiece method is used to executes these constraints with

FoibleBoxPiece class protocol for constraints:

doConstraints: aSymbolKey with: anArgument

```
self constraints do: [:aConstraint | aConstraint key = aSymbolKey
    ifTrue: [aConstraint receiver perform: aConstraint selector
        with: anArgument]]
```

5.2.1 FoibleValue

Generally, if box piece A is positioned with respect to box piece B using:

boxPieceA positionBelow: boxPieceB by: 20

box piece A would be located 20 pixels below box piece B's center point. When a FoibleValue is positioned with respect to another FoibleValue, we do not want to position it with respect to the center point of the other FoibleValue. For example, the 'EndTime' FoibleValues of the Pert Chart boxes in Figure 3.1 were positioned with respect to the 'Duration' FoibleValues using the *positionBelow:by:* method. Without adjustment, the 'EndTime' FoibleValues would begin between the 'a' and 't' of 'Duration'. This is clearly not what we want, so the FoibleValue class provides protocol for slightly adjusting the FoibleValue after the corresponding FoibleBoxPiece method has positioned it.

A FoibleValueByPort is a value that is placed next to a port. For example, Figure 5.3 has a 'Compare' box that has three FoibleValueByPort values. They are used as labels for the output ports. The FoibleValueByPort class has an instance varible, *port*, that stores the port that it should be placed next to. The visual manager of the FoibleValueByPort uses this instance variable and the positioning methods of the FoibleBoxPiece class to position its figure. The FoibleValueOfPort class extends the FoibleValueByPort class by adding a constraint between the value and its port.

5.2.2 FoiblePort

It is possible to have a port that has doesn't have a visual representation. Figure 5.1 shows the Pert Chart for making nachos again, but this time the ports can not be seen. The boxes still have references to the port meta-figures in their *inputPorts* and *outputPorts* instance variables, but these meta-figures do not have a figure to represent them. The visual manager for these ports, FoiblePortWithNoFormVisual, has a nil *form* instance variable.

Figure 5.1: Boxes With Non-Visible Ports

Even though the ports have no form, they have an area (displayBox) that is used by the LinkTool to determine where to link. The displayBox is taken to be a proportion of the box's FoibleShape. For example, the input ports used in Figure 5.1 have a displayBox equal in size to the left half of the FoibleShape, and the output ports have a displayBox equal in size to the right half of the FoibleShape. If there were two input (output) ports, the displayBox of the first one would be in the top left (right) corner, and the displayBox of the second would be in the bottom left (right) corner to the FoibleShape.

When a port's box gets moved, all of the links attached to the port must be moved also. The port notifies all of its links with the *moveLinksBy:* method shown below. The links in turn have to adjust their link pieces to account for the new position of the port.

FoiblePort class protocol for transforming:

moveLinksBy: aPoint

```
self links do: [:each | each port: self movedBy: aPoint]
```

Using the FoibleSelectionTool, ports (and other box pieces) can be moved from their original position in the box. Figure 5.2 shows an example of FlowKit boxes that have some of their box pieces moved around.
Ports, unlike other box pieces, must remain on or in the FoibleShape. The FoiblePort class provides a special method, *translateByRestricted:*, that the FoibleSelectionTool uses while moving the port around. It does not allow the port to leave the area of the Foible Shape. If a port has any links attached to it, the port can not be moved (unless the whole box moves).

FoiblePort has two subclasses, FoibleInputPortWithSpy and FoibleOutputPortWith-Spy, that cause a *SpyBox* to be created when the port is shift-selected. Figure 5.3 shows a FlowKit box that has a SpyBox on each of its output ports. SpyBoxes are useful when the user wants to know the value that the port last received or broadcast, but the value is not displayed in the port's box. SpyBoxes are toggled by shift-selecting the port they are connected to. The value in the SpyBox can not be edited.

Figure 5.2: Moving Box Pieces

5.3 FoibleLink

When one of the ports that a link is attached to moves, the link is sent the *port:movedBy:* message, as shown above in FoiblePort's *moveLinksBy:* method. The FoibleLink determines whether it was its source port or its destination port that moved, and then sends the *translateBy:* message to the appropriate link pieces in its link piece collection. The link pieces will make the necessary adjustments to their figures, as shown in Figure 5.4.

Moving a box has two special cases. First, if there was only a single link piece, new link pieces may have to be created as shown in Figure 5.5. Second, if the opposite happens, some link pieces may have to be fused together. To assist in this situation, the FoibleLinkPiece class provides the *isNil* method that returns *true* if the length of the link piece has become zero.

Figure 5.3: SpyBox and SpyLink

5.4 How to Use InVaider's Tools

As stated earlier, every visual language must provide the user with the ability to perform four basic figure operations: creation, deletion, selection and translation. The tools which InVaider provides for these operations will be described in the following sections. Each tool provided by InVaider performs its operations on the figure's meta-figure object, which in turn may tell its figure to do something. For example, when the user selects a figure to delete with the Deletion tool, the figure's meta-figure is sent the *removeFrom:* message. The meta-figure will send this message to its visual manager and the visual manager will delete its figure (actually it asks the DrawingView to delete it).

5.4.1 FoibleCreationTool

Each figure is created through some sort of creation tool. There are two types of creation tools, those that create a figure instantly and those that have you choose the figure from a pop-up menu. The Pert Chart application has only one type of box, so it uses the former and the FlowKit application has many boxes, so it uses the later.

Figure 5.4: Moving a Box With Links Attached

The FoibleCreationTool provides protocol for creating figures instantly. You must instantiate the tool in a class method of the meta-figure class that you wish to have the tool create. The tool will then record the class in which it was created so that it knows what kind of meta-figure to make. When the creation tool is activated, it sends the *createNotifying:* message to the meta-figure's class.

The MenuChoiceCreationTool provides protocol for creating figures that are chosen from a pop-up menu. The tool must be instantiated in a class that has a menu defined for it. The menu is defined using the three class methods, *menu:*, *menuChoices* and *menuValues*. The methods used to build the pop-up menu for the Function boxes of the FlowKit application are shown below.

Figure 5.5: Moving a Box May Split a Link

FlowKitBox class protocol for menu:

menu: thisClass

```
        ↑PopUpMenu labelList: (Array with: thisClass menuChoices)
            values: thisClass menuValues
```

FormulaBox class protocol for menu:

menuChoices

```
        ↑(OrderedCollection new)
            add: 'Addition';
            add: 'Subtraction';
            add: 'Multiplication';
            add: 'Division';
            add: 'Minimum';
            add: 'Maximum';
            yourself
```

menuValues

```
        ↑(OrderedCollection new)
            add: #AdditionBox;
            add: #SubtractionBox;
            add: #MultiplicationBox;
            add: #DivisionBox;
            add: #MinimumBox;
```

```
add: #MaximumBox;
yourself
```

The *menu:* method returns a pop-up menu to the MenuChoiceCreationTool, and the other two help to build the menu. The *menuChoices* method provides the list of the meta-figures that can be created, while the *menuValues* method provides the classes of these meta-figures. Once a meta-figure is chosen, its class will be sent the *createNotifying:* message as previously discussed.

5.4.2 DeletionTool

The DeletionTool allows the user to delete a box or a link. When a composite figure is deleted, like a box or link, it removes all of its component figures with it. If a figure has other figures that are dependent upon it, these figures are removed also. For example, when a box is removed, we have to remove any Links that are attached to it. Each meta-figure has a boolean attribute, *canDelete*, that is used by the DeletionTool to determine if the user is actually allowed to delete the meta-figure.

5.4.3 FoibleSelectionTool

The FoibleSelectionTool allows users to select, move and edit figures. If the shift key is depressed while a selection is made, the figure is selected for editing. If the shift key is not depressed, then the figure is selected for moving. The visual manager of each figure has boolean attributes, *canSelect*, *canMove* and *canEdit*, that are used by the FoibleSelectionTool to determine if the user is actually allowed to perform the chosen operation on the meta-figure.

If the FoibleSelectionTool is trying to select a figure, and the cursor is on a composite figure, such as a box or link, the component figures are checked first to see if the cursor is pointing at them. For example, if the cursor was pointing at a port, the port would be chosen, not its box. If the user is trying to select the box, then the box must be selected at a point where it doesn't have a box piece. Since the area of a FoibleShape box piece is as almost as large as the area of the FoibleBox that contains it, the *canSelect* attribute of every FoibleShape is set to false. This keeps the FoibleSelectionTool from choosing it, and allows a much larger area to select when trying to choose the entire box.

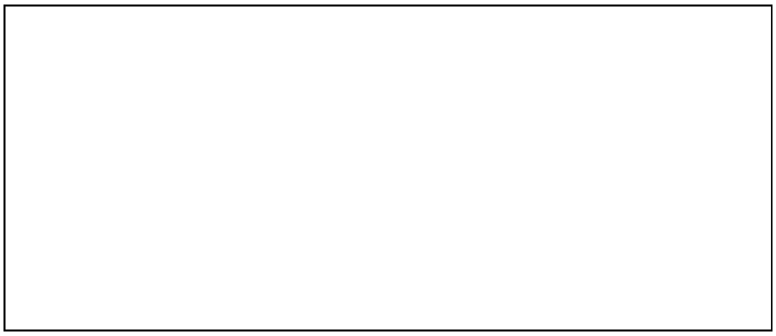

Figure 5.6: Editing the Size of a Box

When the box is shift-selected, the user can change the size of its BoxShape, relative to its origin. The box shape can be made smaller or larger, but the other box pieces will not be moved. The user will have to move them to new locations individually, with the FoibleSelectionTool, if desired. Figure 5.6 shows an example of resizing a box and moving the other box pieces to new locations. The box was resized to make room for its new name.

When a link is shift-selected, the user can change the shape the of link by adding new link pieces. The user must choose two connecting link pieces with the FoibleSelectionTool. The two points that are chosen will determine which segments of these link pieces will be altered. Figure 5.7 shows an example of this.

5.4.4 LinkTool

The LinkTool allows the user to create a link between an input port and an output port. The user can begin the link from either port, and the link is built one piece at a time. When the user chooses a port, it becomes the *anchor-point*. When the user moves the cursor, the vector that goes from the anchor-point to the cursor is broken into its x-vector and its y-vector. A link piece is created for each component (see Figure 5.8). If one of

Figure 5.7: Editing a Link

these vectors is nil, a link piece will not be created for it. When the user clicks on the Drawing, a new anchor-point is created. Link pieces will appear along the new x-vector and y-vector.

When the user moves the cursor onto a box, the cursor position will be adjusted to the closest port (of the correct type). When the user clicks on the desired port, the link is established. If the user decides not to complete the link, a shift-click will erase the existing link pieces and terminate the link creation process.

Figure 5.8: Creating a Link

Chapter 6

Building an Application With InVaider

6.1 Overview of the Building Process

An application built with InVaider must provide concrete subclasses of the abstract classes provided by the InVaider framework. All applications must subclass the FoibleProgramEditor class, the FoibleCreationTool class and the FoibleBox class to define at least one type of box and to provide a way to create it. Most applications will have a subclass of the FoibleBoxVisual class (which is a subclass of FoibleVisualManager) to provide a specific visual manager for the box. In addition, any application that performs calculations will have a subclass of the FoibleCalculationManager class to provide a specific way of computing and passing values throughout the system.

The above outline is the minimum amount of work that needs to be done, but most applications will have to implement other operations that are specific to the particular appplication being developed. InVaider includes many support classes that are not part of the actual framework, but that are used by many applications. Most of these support classes are subclasses of the FoibleValue class, and provide the programmer with a small library of pre-defined value meta-figures. The next section will show how to build a visual programming language using the InVaider framework and the provided support classes.

6.2 Implementing the Building Process

This section describes how to implement the Pert Chart application. The Pert Chart Visual Language defines a box meta-figure, PertChartBox, which is used to model a task. The PertChartBox class is a subclass of FoibleBoxWithValues instead of FoibleBox, because it needs to keep track of duration and ending time values. Likewise, the visual manager, PertChartBoxVisual, is a subclass of FoibleBoxWithValuesVisual. A PertChartBox needs to calculate its ending time, so it has to have a calculation manager. The calculation manager class, PertChartBoxCalculation, is a subclass of FoibleCalculationManager.

6.2.1 Pert Chart Box

A PertChartBox contains the meta-figures for its ports and values. It has one input port and one output port. These ports can have any number of links to them, because a task may depend on, or be depended upon by, any number of other tasks. There are four additional values contained in a PertChartBox. Two of them, 'Duration:' and 'EndTime', act as labels for the numerical values of the box's duration and ending time (see Figure 3.1). The labels can not be selected, but the user can edit the numerical values, using the FoibleSelectionTool. The following PertChartBox method initializes these value meta-figures.

PertChartBox class protocol for initializing:

initializeValuePieces

```
| endDate duration value3 value4 |
endDate := FoibleValueOfPort withValue: self initialValue
    forPort: (self outputPorts at: 1).
endDate notifyPortOfNewValue.
endDate canEdit: true.
self addValue: endDate at: 1.
```

```
duration := FoibleValue withValue: self initialValue.
duration notifyBoxOfNewValue: self.
duration canEdit: true.
self addValue: duration at: 2.
value3 := FoibleValue withValue: 'Duration:'.
value3 isVisible ifTrue: [value3 visualManager canSelect: false].
self addValue: value3 at: 3.
value4 := FoibleValue withValue: 'EndDate:'.
value4 isVisible ifTrue: [value4 visualManager canSelect: false].
self addValue: value4 at: 4
```

6.2.2 Visual Manager

The ending time value of a PertChartBox is broadcast to the system through its output port. If the user edits the ending time of a task, this new value is broadcast throughout the system and is used by the tasks that depend on it to re-calculate their ending times. This allows the user to see what would happen if a particular task had a different ending time.

The following PertChartBoxVisual method initializes and positions the Value figures used to represent the Value meta-figures. It must be present whenever the programmer wants to have application Value figures displayed in a Box's figure.

PertChartBoxVisual class protocol for initializing:

initializeValueForms

```
| values |
values := self metaFigure values.
(values at: 3) isVisible ifTrue: [((values at: 3) visualManager)
    growRight; positionAt: 6 and: 0.4; initializeForm].
(values at: 4) isVisible ifTrue: [((values at: 4) visualManager)
```

```
        growRight; positionBelow: (values at: 3) by: 5; initializeForm].
    self metaFigure duration isVisible ifTrue: [(self metaFigure duration
        visualManager) growRight; positionAfter: (values at: 3) by: 2;
        initializeForm].
    self metaFigure endDate isVisible ifTrue: [(self metaFigure endDate
        visualManager) growRight; positionAfter: (values at: 4) by: 2;
        initializeForm]
```

6.2.3 Calculation Manager

The Pert Chart application calculates the ending time of tasks using the calculation manager defined in the class PertChartBoxCalculation. If a visual programming language built with InVaider needs to calculate values then it needs to define a calculation manager. When a PertChartBox has received a token from each of the tasks that it depends on, it calculates its ending time and broadcasts it throughout the system. In addition, it must update the ending time value that it is displaying. The following PertChartBoxCalculation methods perform these operations.

PertChartBoxCalculation class protocol for calculating:

token

```
    | boxValues newEndDate |
    boxValues := self getInputValues.
    boxValues isNil
        ifTrue: [newEndDate := self metaFigure duration value]
        ifFalse: [newEndDate := self calculateValues: boxValues].
    self announceResults: newEndDate
```

calculateValues: someValues

```
    | maximum |
```

```
maximum := someValues first.
someValues do: [:each | maximum := maximum max: each].
↑maximum + self metaFigure duration value
```

announceResults: newEndDate

```
self metaFigure endDate value: newEndDate.
self metaFigure outputPorts first token: newEndDate
```

A PertChartBox calculates its ending time using its duration and its input values. The input values are the ending times of the tasks that it depends on. Each link to a PertChartBox's input port carries an ending time from a task it depends on. The following method collects the values from the previous tasks.

getInputValues

```
| values |
self metaFigure inputPorts do: [:each | each metaFigure links
    isEmpty ifTrue: [↑nil]].
values := self metaFigure inputPorts first links collect:
    [:each | each source value isNil
        ifTrue: [↑nil]
        ifFalse: [each source value]].
↑values
```

6.2.4 Creation Tool

Every application needs to have a subclass the FoibleProgramEditor class that contains a *tools* method similar to the one below. This method adds any new tools that the application needs. Each new application will have at least one creation tool that needs to be added to the tool pallette. The following PertChartProgramEditor method inserts the PertChartBox creation tool into the tool pallette.

PertChartProgramEditor class protocol for tool methods:

defaultTools

```
| tools |
tools := super defaultTools.
tools add: PertChartBox creationTool.
↑tools
```

6.2.5 Running a Pert Chart Application Program

Finally, users should type *PertChartProgramEditor startUp* to create and open a new Pert Chart application window. This invokes the *startUp:* class method in the FoibleProgramEditor class, sending it the name of the application.

FoibleProgramEditor class protocol for starting:

startUp: aLabel

```
self openOn: FoibleProgram new withLabel: aLabel
```

Chapter 7

Conclusion

7.1 Suggested Enhancements for InVaider

Often, we would like to combine primitive boxes together to make a Complex box. Complex boxes may be used to represent procedures. Currently, InVaider does not support Complex boxes, but they are clearly useful in large programs, as a way to organize program functionality. The existing InVaider tools should already be able to physically interact with Complex boxes, but they do not know what to do with them. They will need to have a logical policy built into them. In addition, a new tool will be needed that can create a Complex box from primitive boxes, and another tool will be needed to perform the reverse operation. The code for these Complex Box tools will be very similar to the code contained in HotDraw's GroupFigure class.

Complex boxes will be quite large, and will likely take up most of the visible screen area. It would be nice if InVaider provided a tool that was able to collapse a Complex box to a single box that has the required inputs and outputs for its procedure shown as ports. In addition, a tool that could collapse a box down to an icon would be very useful. This would give the user the same functionality, but in a smaller screen area. If InVaider had a tool that could collapse a box, it would also have to provide a tool that could expand the box to show its original contents. This would allow the user to study and edit the box.

The value passing facility needs to be improved also. Currently, a Link can only send a value from its source to its destination, but many applications, for example a computer network, would like to pass values both ways. Two-way Links can be simulated with two one-way Links now, but this requires more work by the user and makes the program look more complicated.

7.2 Improving InVaider

InVaider has been built on top of HotDraw, but in an encapsulated way. Most changes to HotDraw will not require any changes to InVaider. Currently, InVaider programmers do not have very many HotDraw figures to choose from, but they will be able to use new figures as soon as they are built into the HotDraw framework.

Even though the InVaider framework is largely independent of the HotDraw framework, InVaider will continue to improve as HotDraw improves. Similarly, HotDraw has benefited in many ways from the implementation of InVaider, since it was the first major test drive of HotDraw after its reimplementation in ObjectWorks\Smalltalk release 4.0. It is only through the continual use and constant building of applications that frameworks such as InVaider and HotDraw can be improved.

BIBLIOGRAPHY

[Alex87] James H. Alexander. "Painless Panes for Smalltalk Windows." In *OOPSLA '87 Conference Proceedings*, October 1987, pp 287-294.

[Catt91] R.G.G. Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, Reading, Massachusetts, 1991.

[Foot89] Brian Foote and Ralph E. Johnson. "Reflective Facilities in Smalltalk-80." In *OOPSLA '89 Conference Proceedings*, October 1989, pp 327-335.

[Gold83] Adele Goldberg. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Jind90] William Alfred Jindrich Jr. *FOIBLE: A Framework For Visual Programming Languages.* M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1990.

[John88] Ralph E. Johnson and Brian Foote. "Designing Reusable Classes." *Journal of Object-Orientated Programming*, (June/July 1988), pps. 22-30, 35. University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.

[John92] Ralph E. Johnson. *Documenting Frameworks using Patterns.* University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.

[Kras88] Glenn E. Krasner and Steven T. Pope. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* ParcPlace Systems, 2400 Geng Road, Palo Alto, CA, January 1988.

[Lalo90] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk, volume 1.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Parc90] ParcPlace Systems. *Objectworks\Smalltalk, Release 4 Notes.* Mountain View, California, 1990.