The joy of Smalltalk

Since the book does not make any assumptions about the reader's background, it is suitable for

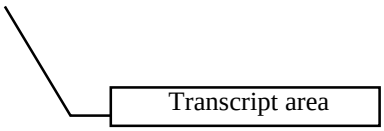
Chapter 1 - Object-oriented programming - essential concepts

Overview

(1327 squared) < (153 * 20000)

is a typical test to determine whether a comparison of two expressions gives a yes or a no answer. Type it

All Smalltalk code consists of ‘messages’ to ‘objects’ and this is why it is called object-oriented. Some of the messages used above include squared, factorial, and <



```
285 + (37 squared)
```

Smalltalk now asks 37 to execute message squared. This returns object 1369 and the code now effectively becomes

```
285 + 1369
```

Smalltalk now asks object 285 to execute message + with argument 1369. This returns the final result 1654. Note that this works because *all Smalltalk messages return objects*, and messages can thus be combined.

The examples that you have just seen cover all possible forms of Smalltalk messages and if they give you the impression that Smalltalk is essentially simple, you are quite right - Smalltalk is based on very few very powerful ideas. Unfortunately, the ease of reading and writing pieces of Smalltalk programs does not mean that writing significant Smalltalk programs such as a word processors or spreadsheets is very easy. Difficult task are difficult even in Smalltalk - but not as difficult as in most other languages as you will see later.

At this point, you would probably like to proceed to other Smalltalk programs, and if you really cannot resist it, you can skip to Chapter 3 which starts our discussion of Smalltalk. However, the ideas of an

1. I press the *Power* button, which can be interpreted as sending message ‘Initiate cooking time setting

1.3 Examples of objects in computer applications

The situations described in the previous section illustrate the concept of objects and messages in

Figure 1.7. The opening window of *Farm Launcher*. Use it to select the first version of the *Farm* program. ge select the firs6 Tc (

the fourth is a cow with a name and a color and understands the same cow messages as in *Farm 2*. To distinguish the products from the factories, the products are called *instances* and our current farm thus has three classes (Cat, CowCat,d ourj 271.1 Tc TD /F3 9 Tf 0.012 Tc 0 Tw (2) Dogj 17.86.56 TD /F0 9.96 Tf -0.016303c 0.03041Tw (

Cat,C

our new objects in the library too and if we come across another application that needs them (such as an

```
3 + 5
```

must be treated as a message + to object 3 to do addition with object 5 as in

‘Object 3, do + with object 5 and return the resulting number object’

or, to put it differently, as

‘Object 3

and executes it with the *do it* command. This sends the open message to Farm and Smalltalk starts by looking for the definition of the open message in Farm. It finds the definition and discovers that its operation starts by a request to the Window class object to create an instance of itself from a specification provided by the Farm

Exercises

1. Use the System Browser to display the comments of the following classes and print them out using command *hardcopy* in the text view's <operate> menu.
   a.

- definitions of methods name, moo, eat

Figure 1.23. A point can be represented by Cartesian coordinates x and y or by polar coordinates

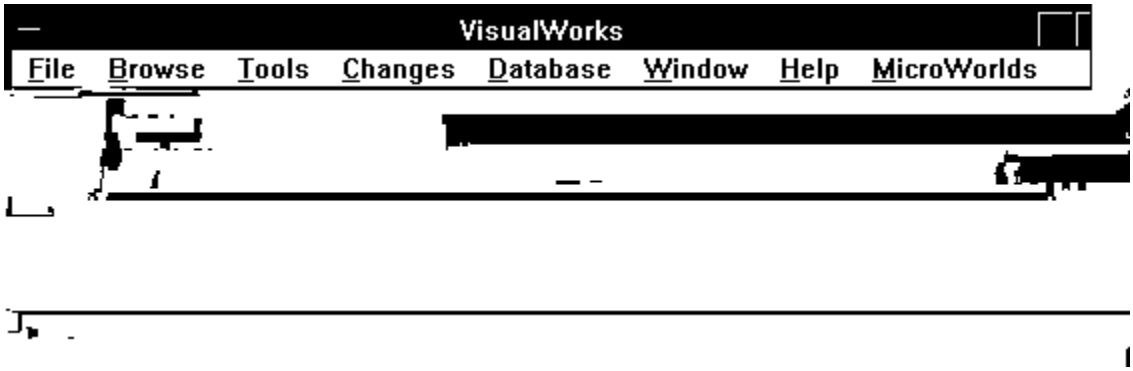

VisualWorks
File Browse Tools Changes Database Window Help MicroWorlds

Figure 1.30. A very small part of VisualWorks class hierarchy tree. Concrete classes are shown in **boldface**, all other classes are abstract.

To find out about Smalltalk's class hierarchy, use the *System Browser*. After selecting a class such as Magnitude, select the *hierarchy* command in the class view <operate> menu and the text view will display as in Figure 1.31, showing subclasses and superclasses and all instance variables.

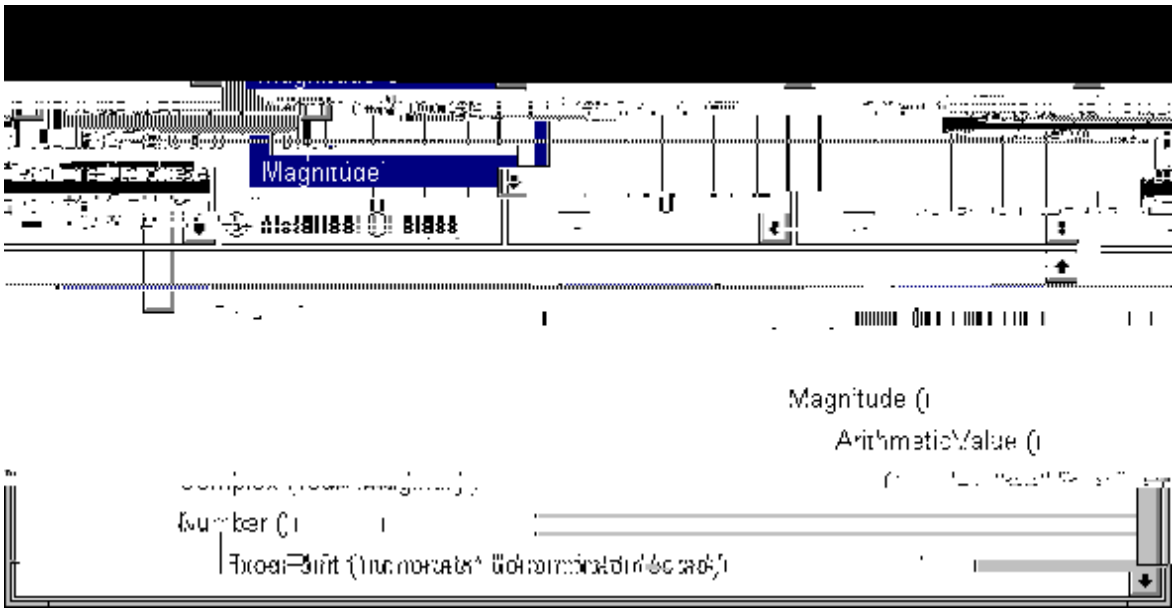


Figure 1.31. System Browser showing a part of the hierarchy of class Magnitude.

The hierarchy shown in Figure 1.31 shows both abstract and concrete classes. Class Magnitude is abstract and factors out properties needed for comparison, and its subclasses include numbers, printable characters, date objects, and time objects. Magnitude has no instances because there is no need for such abstract objects. Classes ArithmeticValue, Number, and

Exercises

1. Find and count all definitions ('implementations') of the following methods.
   a. displayOn:
   b. squared
   c. <
   d. =
2. Give an example of polymorphism from the physical world.
3. The three animals in the *Farm* world share the following messages: color, name, eat, isHungry, run, walk. Some of them are implemented in the same way in each animal, others are not and the program executes them polymorphically. List each group on the basis of your intuition.

Conclusion

understood by instances are called instance messages. A detailed definition of the computation performed by a message is called a method.

Since all work in a pure object-oriented language is achieved by sending messages, method definitions themselves consist of message sends. Some of the messages may be directed at the object

Introduction to Smalltalk - Chapter o srpObject-oriented programmingsrpessential concepts

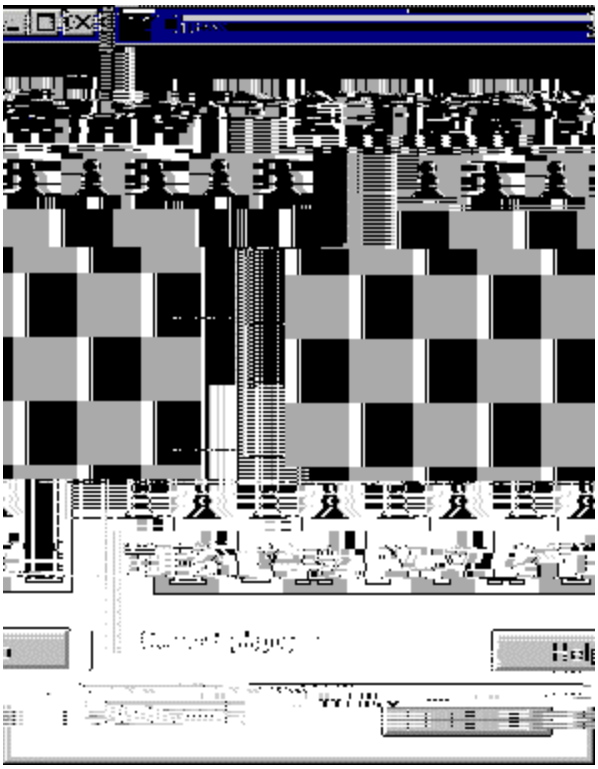
Terms introduced in this chapter

abstract class

Chapter 2 - Finding objects

Overview

In Chapter 1, we introduced the principles of object-oriented problem solving but we have not paid



2. Repeat Exercise 1 but assign each classes instead of pieces. Each card contains the name of the class, its brief description, and a list of its functionalities. Replay the scenarios.
3.

section. We then expand the high-level conversations from Step 1 into class-level conversations involving the identified classes.
As we develop the conversations, we record the responsibilities that the classes must have to implement

Although our description is complete, it gives the inaccurate impression that development is a linear sequence in which step n is always completed before step n+1, and never repeated. The reality is generally quite different because when we start working on step n+1, we often find that the results obtained in step n are incomplete or partially incorrect. Results of step n must then be corrected and this may require corrections or additions to step n-1as well, and so on. While this is quite normal, one should always attempt to complete each step as well as reasonably possible. Do not proceed to the next step lightheartedly, as5Tw .ihichted t becauni rehavehe nreturnhe next ionvious as welnyatt.0 -111.76 TD -0.016 T09 0.2377 819 (to coArochpeatebly

We will now evaluate the desirability of the selected nouns as classes:

- *Address*. If the address included the name of the city, postal code, province or state, and other information, we would implement it as a class. As it is, an address is just a string of alphabetic and numeric characters and strings are already in the library. We thus remove *Address* from our list of candidates.
- *Apartment.* Yes, we will implement this concept as a new class because it represents an object that

1. Assign each class to an individual or a small group that will be responsible for maintaining all information about the class.
2. Each group prepares a CRC card (Figure 2.8) for its class. Use 5 x 8” paper index cards.
3. Each group proposes a short description of the purpose of its class, gets it approved by the rest of the

3. *User clicks Open.*
4. *System opens Farm 1 user interface.*

Scenario 4: *User sends 'meow' to the cat.*
High-level conversation:
1. *User* clicks *cat* in the Animals list.
2. *System* displays commands understood by cat and updates the rightmost part of the window.
3. *User* clicks *meow*.
4. *System* displays results in the right-most part of the window, restores the animal list, and erases the list of commands.

Scenario 5:

Identify class responsibilities

We will now examine our scenarios and expand high-level conversations into class-level conversations.

Scenario 1: *User starts Farm.*
High-level conversation:
1. *User* enters and executes a Smalltalk expression.

4. *System*

Components:

- list of commands understood
- list of commands forbidden in Farm1

Responsibilities

-

- Opening
  -

Cow: Simulated cow. Knows the commands it understands and how to respond to them. Knows commands that it should not understand in Farm1. A domain object.
Superclass: Object
Components: inherited
Responsibilities *Collaborators*

Exercises

1. Repeat the example, completing all CRC cards, writing all missing scenarios, descriptions, and performing all required tests.
2.

- Construct a Context Diagram showing external actors (outside the scope of current project) and major parts of the system to be developed.
- Construct a Glossary of terms. This Glossary will be continuously updated during the process.

Preliminary Design:

-

Problem: Draw a 50-pixel side square whose lower right corner is the home position of the pen (Figure 3.2).

Figure 3.2. *Pen world 3*

Pen is the receiver because it comes first, newBlackPen is a message. The name Pen begins with a

variables

is ‘pen *gets* (or *refers to*

Exercises

In the first two exercises, write an algorithm and the corresponding code on a piece of paper, execute the task by clicking buttons in the *Pen world*

Figure 3.5. Pen world 5 with code selected for execution.

To write and execute a code fragment in *Pen world*

Dealing with simpler errors

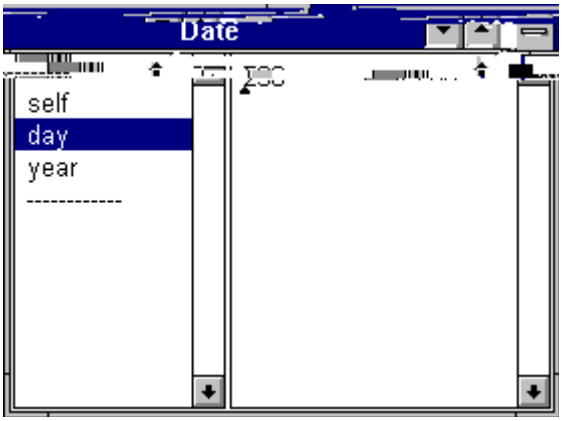
Sooner or later, you will type and attempt to execute incorrect code. Code can be incorrect in more than one way:

•

2. Objects in *Pen World 5*

with the *inspect* command instead of using the inspect message and *do it*.

The self item at the top of the list refers to the receiver of the inspect message, in this case the Date object. The remaining items provide access to all instance variables of the inspected object, both inherited and defined in its class.



Exercises

1.

```
'a cat' spellAgainst: 'a car'                    "Returns 80 - degree of similarity between the two strings."
```

Whereas one-keyword messages don't seem to cause any problems, keyword messages with multiple keywords may be more difficult to read. Beginners often don't understand that the multiple keywords form a single message. The following are a few examples of multiple-keyword messages, one message per line.

Keyword messages with

Figure 3.12. Exception window resulting from the execution of a halt message .

Clicking *Debug* opens the Debugger window in Figure 3.13. The scrollable top view of the window provides access to the call stack, the middle view is for displaying the code of a method selected from the stack, and the bottom part contains two inspectors. We will now explain these parts and illustrate their use.

As we already mentioned, the top message in the call stack is the method currently being executed, the one below is the method that sent the message on the first line, and so on0.0 -11.ed6, andainssisd ie Figure
halthalthalti07essage .super iehalt

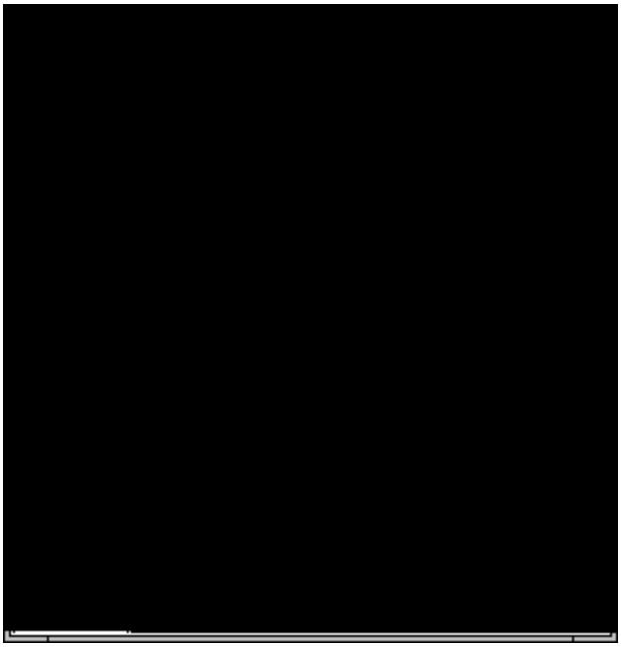


Figure 3.13. Debugger window with two most recently invoked methods shown on top.

To see the code of a message and the point reached in its execution, click the appropriate line in the call stack. Since a code fragment is always referred to as

writes the string enclosed between apostrophes to the Transcript.

The show: message expects a string argument and if you want to print information about an object

You can also create a global variable with an initial value by evaluating an assignment such as

```
NewGlobal := 10
```

which will open a Notifier as in Figure 3.16 and create and initialize the variable upon confirmation.

Figure 3.18. Class instance variables are listed on the class side of the browser.

Although class variables are rare, they are useful and *Farm 6* gives an example that illustrates their

Exercises

1. Inspect Smalltalk to find all references to Transcript

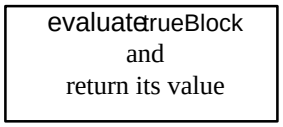
Chapter 4 - True and False objects, blocks, selection and iteration

Overview

Figure 4.6. Response to *find method*

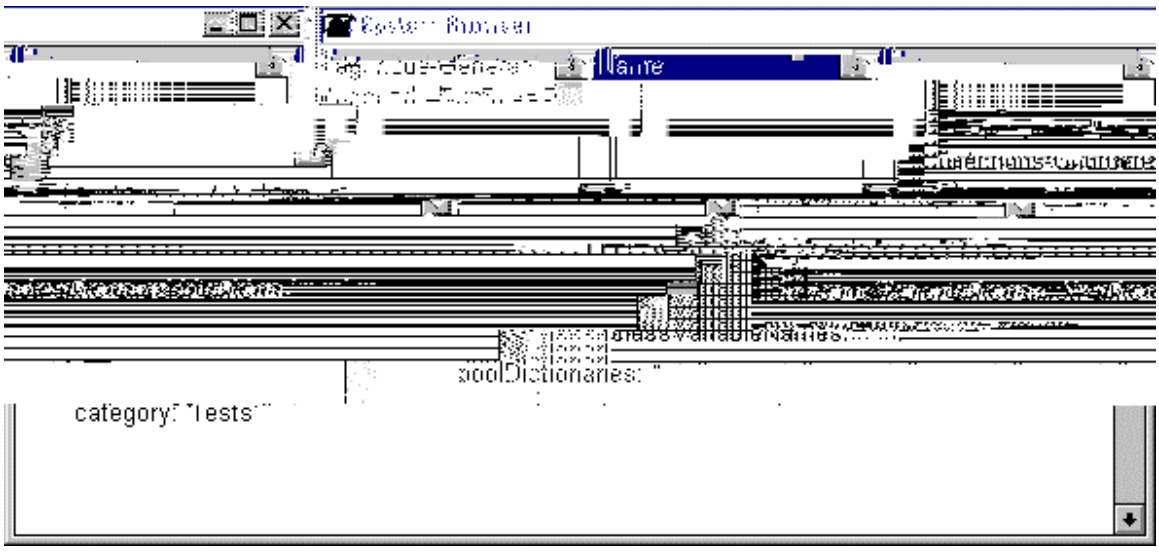
```
^self
```

To write the program, we need to know how to read a *number* from the keyboard and how to select



Introduction to Smalltalk - Chapter 4 - True and False objects, blocks, selection and iteration

class. To do this, open a Browser, and add a new category called for example Tests using the *add* …
TestName 22.084name12/F0 9.96 Tf -0.0343 Tc



This class has not yet been commented. The comment should state the purpose of the class, what messages are subclassResponsibility, and the type and purpose of each instance and class variable. The comment should also explain any unobvious aspects of the implementation.

Select the whole text and replace it with a comment. There are two styles for writing the comment. One is very antropomorphic and uses the first person as in 'I represent a name ..'. Its advantage is that it makes you think in terms of the class which always helps. The other style is less personal and uses the third person as in 'This class represents a name …'. This style seems less extravagant. Choose the style that you like better and use it consistently. We will use the first style because it will force us to think as if we were the class which seems a good idea at this point of the book. Following the template, we write

I represent a personal name consisting of a first name, a middle name, and a last name.

Instance variables

firstName	<String>	first name
middleName	<String>	middle name
lastName	<String>	last name

The comment seems almost unnecessary because it seems very obvious but it is an essential practice to use class comments. As an example, the indication that the values of variables should be String objects is important.

Figure 4.12. Browser with method template.

Let's start with the get method for the firstName instance variable. A get message simply returns the value of a variable and it is standard practice that it is named with the name of the instance variable. The definition is thus

```
firstName
        ^ firstName
```

should return 'John'. Type this expression into a workspace and execute with *print it* to see that it indeed does.

Now that we think about it, it seems that our approach could cause a problem. A programmer might, for example, create a Name object and give it a first and last name but no middle name. If a program then tried to print the name, it would fail on the middle name because its value would be nil and nil does not understand String messages. It will thus be a good idea to modify the instance creation message so that it initializes all instance variable to empty strings with no characters in them. This is, of course, rather useless but at least it will not cause a crash if a variable is left unassigned.

There are several possible solutions to our problem and the simplest one is to define a new initialization method that forces the user to enter some strings for each of the instance variables as in

```
Name first: 'John' middle: 'Mathew' last: 'Smith'
```

This new message is obviously a class message because the receiver is the class which creates the object. We will thus define it on the class side and put it into a new protocol called instance creation . Its

```
[self <= max]
```

to the true object. This returns true or false. The Boolean result is then returned. Note that all subclasses of Magnitude including Date,

Problem

Figure 4.19. y := x := 'John Smith' binds x and y to the same object and both x=y and x==y are true.

As another example, consider that class True and False each have only one instance. As a consequence, equivalence and equality of Boolean objects mean the same thing.

The check of equivalence is faster because it only compares the memory addresses of the internal representation of the two objects. If the addresses are the same, the objects are equivalent; they are one and the same object. Checking whether two objects are equal may take much longer. As an example, to decide whether two non-equivalent strings are equal, we would probably first check whether they have the same

Exercises

1.

```
“Initialize total and clear Transcript.”
total := 0.
Transcript clear.
“Keep gathering item prices and displaying them until the total exceeds 100.”
[total <= 100]
        whileTrue:
```

Introduction to Smalltalk - Chapter 4 - True and False objects, blocks, selection and iteration

Iteration methods are defined in class BlockClosure

singleton - the single instance of a class that does not allow multiple instances
truth table - a table defining a Boolean operation by listing all combinations of *true* and *false* operand

Chapter 5 - Numbers

Overview

VisualWorks library contains many classes representing numbers of various kinds, mainly because computers internally represent and process different kinds of numbers differently. In addition, computer programs often require facilities that go beyond basic hardware capabilities and Smalltalk number classes satisfy these additional needs.

numbers that have magnitude but cannot be used as vectors - indicators of direction or coordinates of points in space. (Instances of Complex and

considerably longer than calculations with

Figure 5.6. Execution of

When we talked about diff

```
t1 := Time millisecondsToRun: [ 10000 timesRepeat: [2+3]].
t2 := Time millisecondsToRun: [ 10000 timesRepeat: [“nothing”]].
timeToAddIntegers := t1 - t2
```

Our complete solution is as follows:

```
| t |
Transcript clear.
t := Time millisecondsToRun: [10000 sum].
Transcript show: 'Nonrecursive sum. Time: ', t printString; cr.
t := Time millisecondsToRun: [10000 sumRecursive].
```

Design

The behaviors that we expect of Currency include creation (create a new Currency object), arithmetic (add or subtract two Currency objects), comparison (equal, not equal, less than, and so on), and printing (for testing and inspecting). Each of these behaviors will be implemented as a protocol, and the notion of dollars and cents suggests the use of two instance variables called dollars and cents.

We have now decided all the major features of Currency except where to put it in the class hierarchy. Currency CMagnitde Tj -338948 011.76 TD -F0 9.96 Tf -0.0101 Tc 0.0176 Tw (nbr

and similarly for cents:.

```
Currency dollars: 100 cents: 3412
```

creates a Currency object with 100 dollars and 3412 cents but we would prefer a Currency object with 134 dollars and 12 cents. Similarly,

```
Currency dollars: 100 cents: -34
```

creates a strange object that does not make sense, and our comparison message = returns false for

```
(Currency dollars: 100 cents: 3412) = (Currency dollars: 134 cents: 12)
```

whereas we would probably expected true.

cents:

4. Currency

```
origin printOn: aStream.
aStream nextPutAll: ' corner: '.
corner printOn: aStream
```

and extend it as follows:

printOn: aStream
"Append to the argument aStream a sequence of characters that identifies the receiver. The general format is originPoint corner: cornerPoint angle: angle."

```
super printOn: aStream.
aStream nextPutAll: ' angle: '.
angle printOn: aStream
```

In this definition, super is a special identifier that allows us to access an identically named method cornerdrawingeave mnoc bs hohe heruly llr 5 - NuTD tendw have mthaccimple Tj 1efini

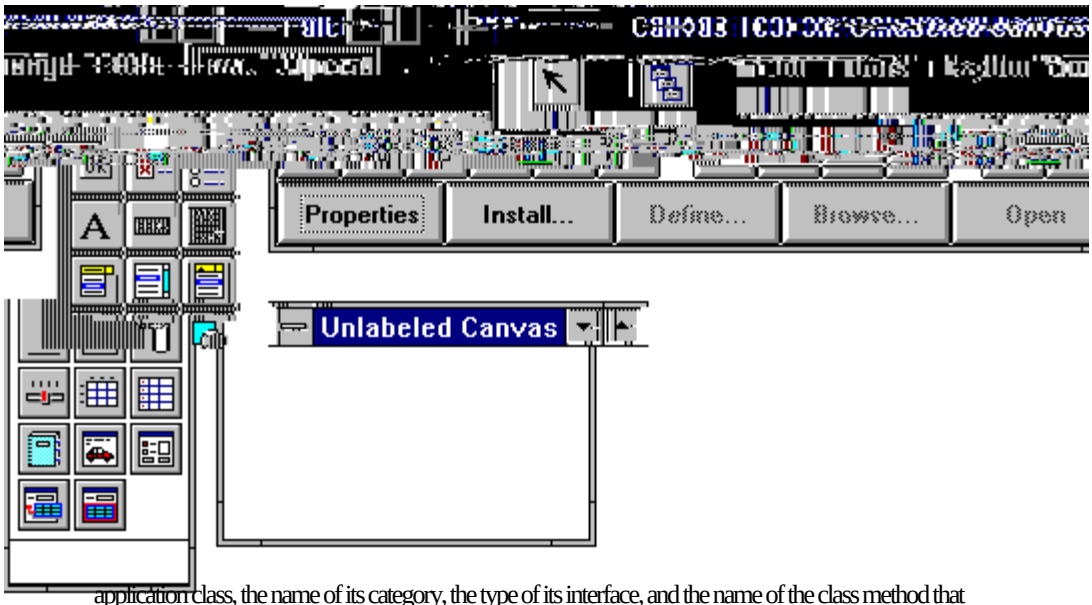


Figure 6.4. Palette with GUI widgets (left), Canvas Tool (top), and unlabeled canvas.Installing the canvas on an application cla

application class, the name of its category, the type of its interface, and the name of the class method that
Install in theCanvas Tool. (Like most other actions, this can be also be done from the <operate> menu of the Canvasitself.) VisualWorks will then lead

name for the specification method is windowSpec and this name is already displayed at the bottom of the window. Use this name unless your application requires several windows which must then be stored as

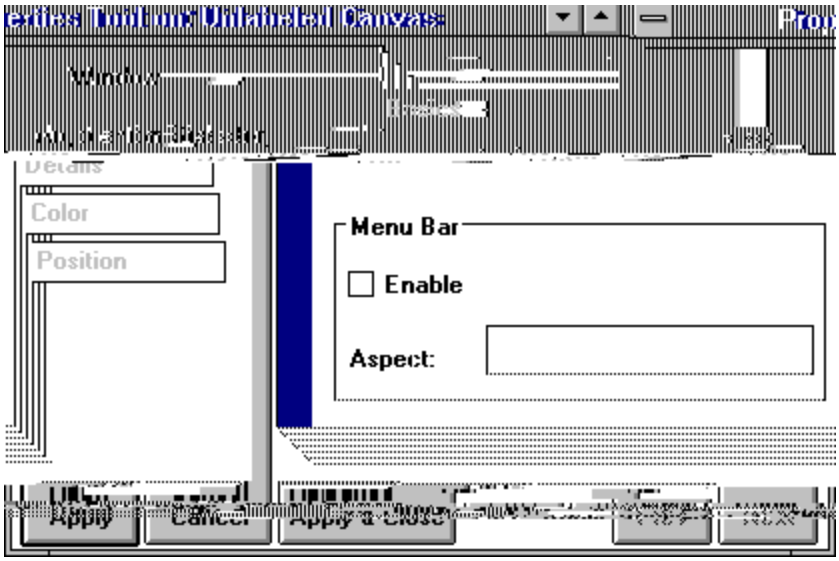


Figure 6.8. Defining a new label for the window.

Figure 6.10. UI Palette and its buttons.

We will now paint the Action Buttons required in our interface. Click the Action Button button (!), move the mouse cursor over the canvas, and click. When an Action Button appears (Figure 6.11), click again to drop it in place. If you don't like the button's position or size, move it or reshape it by dragging its body or its *handles* - the small rectangles in the corners - while pressing the <select> mouse button.

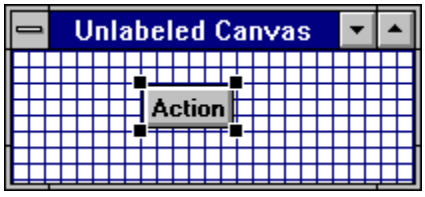


Figure 6.11. Widget handles show that the widget is selected. You can now move it, reshape it, define its properties, or delete it (command *cut* in the <operate> menu).

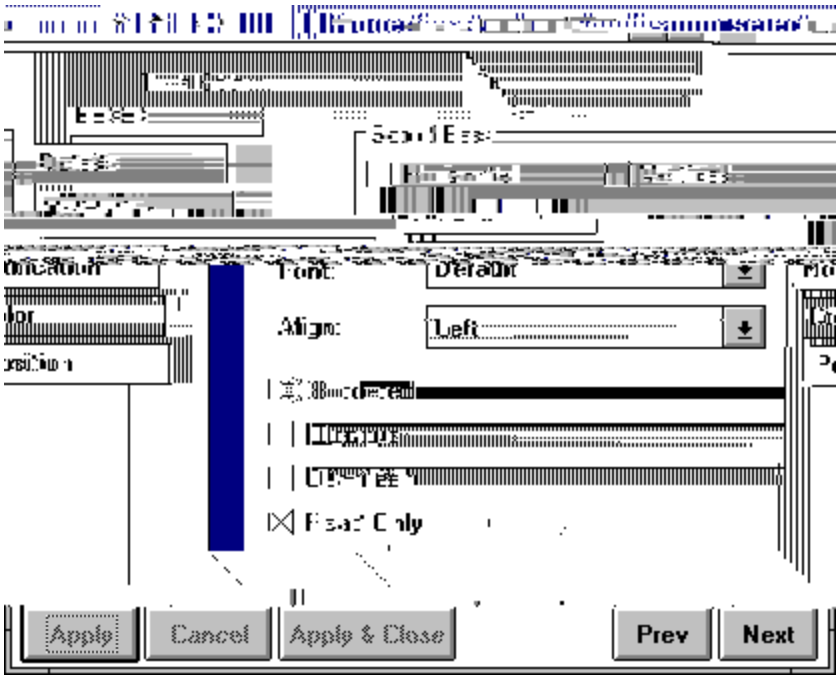


Figure 6.15. *Details* page of Text Editor's properties.

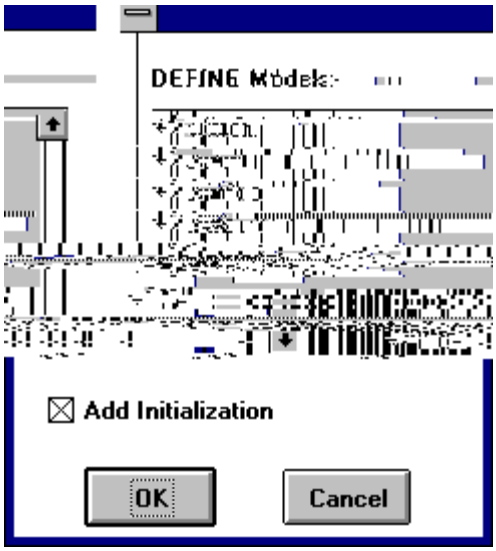


Figure 6.17. Automatic definition of stubs of widget methods. When the cdef Q BT 185.64 135.12 TD -0.0013 Tc 0.0113 Tw (F

Click actions and button

6. The model-dependent relationship has many applications beyond user interfaces. As an example, consider a collection of physical particles that works as follows: When a particle changes its energy by some amount

Figure 6.8. Top: Standard lines of communication among the three components of the MVC triad and the meaning of MVC components. Bottom: A single model may have two or more different view-controller pairs.

Although every widget that displays data has its special model, view, and controller, it is not

- initialization - initialize player to 'X'
- actions - respond to activation of board squares and the *Reset* button
- private - check for end of game, toggle players after a move

Implementation

We will implement the squares as action buttons with blanks, Xs or Os as labels. We will need to change button labels at run time as the players click them, and to do this, we need run time access to them. This access can be gained via the builder which holds a dictionary of all *named widgets* (widgets assigned

((self

graphical user interface (GUI)

```
#(1 2 3 4 5 3 4 5 3 4 5) do: [:el| Trancript cr; show: el printString] exceptFor: 3
```

We found that all tests worked but we should, of course, test the method for other kinds of collections as well. This is left as an exercise but for now, we will generalize our method somewhat: In some situations, we might want to execute selective enumeration on the basis of a test rather than by specifying the element explicitly. As an example, we might want to do something with all elements that are greater than 3.

In this problem, we must allow the second argument to be either a block or any other object, and

1. Test conversion messages asSet, asSortedCollection, asr Tf rderllection

12. Perform the following tasks using enumeration messages:
    a.

address city: ‘Halifax’

TwoDList columns: 3 rows: 5

returns TwoDList (nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil).

Accessing a TwoDList

Just like Array, TwoDList is accessed by at: at:put:. The difference is that the at: argument is normally a Point because a position in a table requires two coordinates. The x part of the point is the column number, and the y part is the row number. The following example illustrates the principle:

```
| table |
table := TwoDList on: #(12 14 16 21 42 24) columns: 3 rows: 2.
Transcript clear; cr; show: (table at: 1@2) printString.          "Prints 21 – element in column 1, row 2."
table at: 2@2 put: 77.able at16  232TD 0.036  T4c 0.872"talnglust  Tw uctt: 7774
```

```
| matrix |
```

Figure 7.14. A three-dimensional array with dimensions 4, 3, 2 (columns, rows, planes).

We can now deduce that if a an n-dimensional array has dimension sizes a_1, ,

formula for convertingan n-dimensiona

Implementation

Creation protocol

1. Since we access the

never redefine it in any subclass. Method basicAt: has exactly this role. If everybody plays by the rules, basicAt: thus has a guaranteed single definition that we can always rely on. When you examine the library, you will find that there are quite a few basic... methods.

```
(items inject: (Currency cents: 0)
        into: [:total :item | total + item price]) displayString
```

Note the use of inject:into: with Currency objects, and the displayString message - you may have expected printString. displayString

```
"Initialize items to a suitably sized sorted collection. Sort alphabetically by name."
        items := SortedCollection sortBlock: [:item1 :item2| (item1 name) < (item2 name)].
        self execute
```

Exercises

1. Find and examine all references to sortBlock: and determine which of them are instance messages: and why they are used.
2. Study and describe how SortedCollection adds a new element.
3. Browse SortedCollectionWithPolicy and explain how it differs from SortedCollection.
4. Formulate the sort block for storing entries of the library catalog from the Section 8.3 in a SortedCollection using the alphabetical order of author names. Make up any necessary accessing methods.
5. Write a code fragment to create ten random rectangles and sort them in the order of increasing area.
6. (Items equee Slaget Mak,ort tifSlaget Makfrequert blMakert nee Sfirget Mak. Exndodert bpreviouste) Stu541T15( ) Tj the alp

8.6 The List collection

Class List

```
contentsFromUser
"Answer an Image with the contents of a user-specified area on my screen."
        ^self completeContentsOfArea: Rectangle fromUser
```

This shows that if we have a window rectangle (call it windowRectangle) we can obtain its rectangle by

```
Screen default completeContentsOfArea: windowRectangle
```

The basic behaviors of list widgets are: create a list widget with labels, add a new label, remove an existing label, sense change of selection, get current selection, and access labels.

The model of a single selection list (the object on which the list widget is dependent) is normally an instance of SelectionInList

```
set includes: 12.                                   “Returns true.”
set includes: [:element| (element rem: 6) ~= 0]     “Returns false.”
```

Removing methods include remove: anElement and remove: anElement ifAbsent: aBlock. Both return the argument, just like add: and addAll:

```
| set |
set := Set new addAll: #(1 2 3 4); yourself.    "Returns Set (1 2 3 4)."
set remove: 1.                                  "Returns 1."
set                                             "Returns Set (2 3 4)."
```

Another message that removes set elements is the binary message - (minus) which calculates *set difference*. The difference of two sets is the set of those elements that are in the receiver set but not in the argument set. In other words, the difference is a set obtained by removing from the receiver those elements that are in the argument. As an example,

```
#(1 2 3 4 5) asSet - #(1 3 5) asSet
```

returns Set (2 4).

hash should be redefined in all classes that redefine =. For efficiency of accessing in sets, two objects that are not equal should have a different hash value. However, this is not required and often not guaranteed.

```
name isEmpty ifTrue: [^false].          "Exit and terminate loop - user indicated end of entry."
price := (Dialog req
```

An obvious example of the need for dictionaries is an assembler program which translates a source program into machine instructions, replacing symbolic instruction names with their binary opcodes. This process is based on associating symbolic names with binary opcodes which could be described as

```
'add' -> 2r10010001
```

the second association replaces the first one because its key is equal to the first key.

IdentityDictionary is an important subclass of Dictionary that uses equivalence instead of equality, and is internally represented as a collection of keys and a collection of values rather than as a set of associations. An IdentityDictionary is often used as a kind of a knapsack to carry around a variety of identifiable objects that could not be predicted when a class was designed and consequently could not be declared as instance variables. To put this differently, an IdentityDictionary

which has the same effect as

```
dictionary at: key put: value
```

Removing associations

aStream

The pop up menu of both lists of original words is as in Figure 9.9. Its commands allow the user to

Instance variables are

<lang1> - name of the first language, a String
<lang2> - name of the second language, a String
< lang1lang2 > - one language dictionary, a

Deletion will be implemented by instance method language1Delete: aString (and a symmetric method language2Delete: aString). The method deletes the whole language1 association with key aString, and propagates the change to language2. Since this and similar messages are sent by the pop up menu after the user has made a selection in the user interface, we can assume that aString is present in the set of keys of language1. We can thus use removeKey: and similar messages without worrying about the ifAbsent: part. The algorithm for responding to pop up command *delete* when a word in language 1 is selected is as follows:

1.

```
fsa run
```

The program runs correctly.

Exercises

1. Use FSA to simulate the JK flip-flop. The JK flip-flop is a binary storage element that stores 0 or 1 and
..56 0 Td [FSA] TJ -0.0072 c (1.) Tj
1.- Ced.92 0 267D /F0 8.04 0.036 Tc 0 Tw (1) Tj 5.J72 0 D /F0 9.96 Tf -0.0101 Tc 02185 Tw 5225nd

Figure 9.13**.** State transition table of a JK flip-flop.

Chapter 10 - Streams, files, and BOSS

Overview

Sequenceable collections are often processed in linear order, one element after another. Although linear access can be performed with collection accessing and enumeration methods, Smalltalk library

- Reading and writing of files.

Execution of each of these tasks involves some or all of the following operations:

-

There are methods that only work with external streams, methods that can be used with read streams but not with write streams, and so on. Most of these limitations are obvious and natural.

Creation

Internal streams are usually created with class methods on: with:, or by messages addressed to the underlying sequenceable collections; ra -0By,streams are ureated with cnd Tj 0 Tc -.03 Tw ( ) Tj 3.000 TD /F3 9 Tf 0.0327

upTo: anObject - repeats sending next until it reaches the first occurrence of anObject or the readLimit. It returns a *collection* whose elements[2] are the elements retrieved by the consecutive next messages from the start of the iteration up to but *not including* anObject. The pointer is left pointing at anObject so that the next next

isEmpty - tests whether position = 0, in other words, refers to how much of the collection has been viewed.

(WriteStream with: '

```
nextPutAll: 'Yours,'; cr; cr; cr;
nextPutAll: (Dialog request: 'Enter Adjunct Librarian''s name' initialAnswer: '');
```

10.4 Example: A Text filter

2. Repeat for each position of the input stream beginning from the start:
    a. For each element of the dictionary do:
        i. Increment current position holder for match string.
        ii. Compare input string and match character.
            1. If no match, reset current position holder for match string to 0.
            2. If match, check if this is the last character to match.

```
ch := InputStream next.
“Copy the input character into the output stream for now.”
OutputStream nextPut: ch.
“Now try to match against successive entries in the dictionary.”
MatchDictionary
        keysAndValuesDo:
```

1. Extend TextFilter to accept blocks as replacement arguments as stated in the specification.

10.5 Example: Circular Buffer

In computing terminology, a *buffer* is a memory area that accepts data from one process and emits it to another process; the two processes work at their own speeds. An example of the use of a buffer is reading a block of data from a file into memory where it is processed one byte at a time by a program. Another example is a computing node on a network that accepts parcels of data arriving in unpredictable amounts and at unpredictable times, processes them one byte at-a-time, and possibly sends the data on to another network node.

The hardware implementation of buffers often has the form of a special memory chip with a fixed number of memory locations with pointers to the first byte to be retrieved, and to the location where the next byte is to be stored as in Figure 10.8. When a new byte arrives, it is stored at the next available location and the pointer is incremented, and when a byte is required from storage, it is removed from the location pointed to and the pointer incremented.

Figure 10.8. Buffer as a fixed size array with pointers to the next available byte and the next available location.

In reality, of course, a byte read from the buffer is not ‘removed’ and only the changed value of the pointer indicates that the byte has been used. Similarly, an ‘empty’ position is not really empty but the new

lomuctr Buarh pmeonc the fomenadic of medingsn of bue pointion isbthe chze ar bue poffer i Tj 36 -11.76 TD -00056 Tc 0.4192
spoca, a haveo

the nqu st onenot rof courionandeeme161 wfacaeinef the
lelt roaal merdware im,ugtwlsfunstomhalitbushhemess and

isFull

Figure 10.10. Main classes used in file processing.

The cookbook procedure for processing data stored in a file is as follows:

1.

Class *Filename* is an abstract class and its concrete subclasses (MaxFilename, PCFilename and its subclasses, and UnixFilename) implement the platform-specific behavior needed on your machine, such as parsing platform-specific syntax of file names. However, you never need to deal with these concrete subclasses because Filename automatically sends all platform-dependent messages to the subclass representing your platform. This is done via Filename's class variable DefaultClass which holds the name of the appropriate Filename class. Removing explicit dependence on one platform makes it possible to write programs that will run on different platforms. This arrangement is similar to the implementation of strings.

External streams perform data transfer operations. Instances of external streams are never created by class messages to external stream classes but by messages to the Filename9/16 Tw (of the26ppropria183) Tj 73abovelar

Figure 10.11. Possible result of typing '*.hlp' in response to Dialog requestFileName:.

The proper use of the combination of requestFileName: and asFilename should thus be something like

delete - as in fileName delete - deletes the Filename object (a file or a directory). As an example of its use,

Example 3: Let user delete a file from a list

Problem: Implement a method to display the file names in the current directory in a multiple choice dialog, and allow the user to delete a file

Solution: This problem does not require a specific Filename and we will implement it as a class method in Filename, following the example of several existing *fromUser methods. The method will obtain the current

This explains how the limited number of external stream classes (Figure 10.8) can provide such a variety of accessing modes - the type of access is controlled by an instance of FileConnection. The other stream creation messages are similar.

Since a file and its mode of access are two separate things, a file initially accessed via one kind of stream may be closed and accessed again via another type of stream. As an example, we have already seen that you may open a file for writing, store some data in it, close it, and open it for reading later.

Positioning

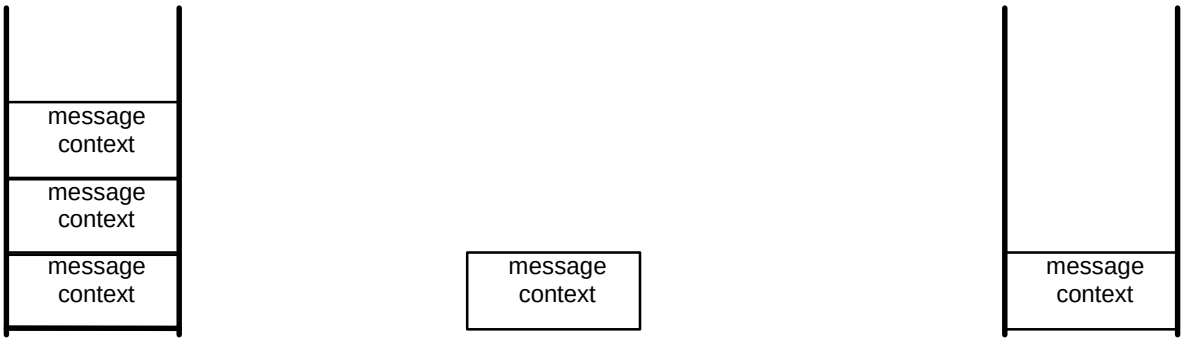
```
| stream |
stream
```

The basic definition of storeOn: in Object simply generates messages to create a new instance of the receiver and further messages to initialize its variables. The interesting part of the definition is that it asks each component of the receiver to store itself. Typically, this results in the component asking its own components to store themselves, and so on. You can see how this can create problems if the structure is circular. The definition of storeOn: is as follows:

```
storeOn: aStream
"Append to aStream an expression whose evaluation creates an object similar to the receiver.  This is
```

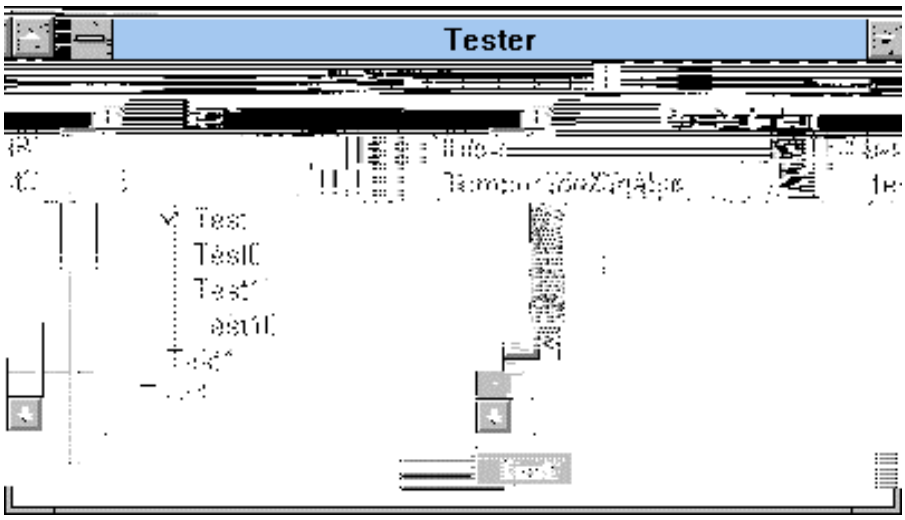
Figure 10.13. The definition of storeOn: is recursive.

As an illustration of the operation of this recursive definition, consider using storeOn: on a literal array containing string elements: The storeOn:


message context
message context
message context
message context
message context

In essence, a stack is an ordered collection whose elements can only be accessed at one end. If we treat the start of the collection as the top of the stack, addFirst: performs push and removeFirst performs pop. Alternatively, we can use the end of the OrderedCollection as the top of the stack with addLast: for push and removeLast for pop OrderedCollection asapproach18talf wfo0 327! 225 12 0

a message to the Transcript. All test methods of a class are assumed to be in class protocol testing, and each test must return true if it succeeds and false if it fails.



As we know, class names are returned by Smalltalk classNames. To obtain information about classes, we already learned to use category Kernel - Classes

a

c.

Modeling randomness

We will assume that the unpredictable parts of the behavior of the problem can be described by a probabilistic model. In other words, we will assume that we can formulate a mathematical function

queue length. The output will, of course, be left to the application model (class BankSimulation) to display. This way, if we want to change the user interface (output of results) or make our simulation a part of a larger scheme, the domain objects can stay the same and only the application model must be changed.

Preliminary design of classes and their responsibilities

    - lowerLimit: anInteger upperLimit: anInteger.
- Accessing - implemented by
    - next. Returns random integer.

4. No complete match, copy character from string to result, increment position in string and result.
5. Compare position 2 in pair1 ($b) with first character of string - match.
6. Compare position 1 in pair2 ($a) with first character of string key; no match.
7. We have a complete match. Select the longest replacement possible, changing result to 'xx'.
8. Continue in this way until the last character in string. At this point, there are two matches and two possible replacements. The longer one is selected giving result = 'xxcdyy' as expected.

This scenario works as expected. But wait - the behavior in the following scenario does not quite meet our expectations.

Scenario 2: Filtering string = 'abcd' with match/replacement pairs pair1 = 'bc'->'xx', pair2 = 'abcd'->'yy'

Assume string = 'abcd' and match/replacement pairs pair1 = 'bc'->'xx', pair2 = 'abcd'->'yy'. Assume as expec 0.sfinishe2 0

```
SortedLinkedList (LinkNode 3 LinkNode 13 LinkNode 33)
SortedLinkedList (LinkNode 0 LinkNode 3 LinkNode 13 LinkNode 33)
```

Finally, we must test that SortedLinkedList works even with non-default sort blocks and we thus define a new SortedLinkedList creation method called sortBlock: and modify our test program to create a new list with the same elements sorted in *descending* order:

```
| sll |
Transcript clear.
sll := SortedLinkedList sortBlock: [:x :y | x > y].
Transcript show: sll printString; cr.
sll add: 3.
Transcript show: sll printString; cr.
sll add: 33.
Transcript show: sll printString; cr.
sll add: 13.
Transcript show: sll printString; cr.
sll add:  0.
Transcript show: sll printString
```

The test produces

```
SortedLinkedList ()
SortedLinkedList (LinkNode LinkNode 3)
SortedLinkedList (LinkNode LinkNode 33 LinkNode LinkNode 3)
SortedLinkedList (LinkNode LinkNode 33 LinkNode LinkNode 13 LinkNode LinkNode 3)
SortedLinkedList (LinkNode LinkNode 33 LinkNode LinkNode 13 LinkNode LinkNode 3 LinkNode LinkNode
0)
```

in the Transcript 0064 agan tconfirmsthe Tcorrectness ofour timpementstion Man tlessons ear.ned

i) Add root

```
        add: 17;
        add: 49;
        add: 32;
        do: [:node | Transcript show: node value printString; cr]
```

which produces

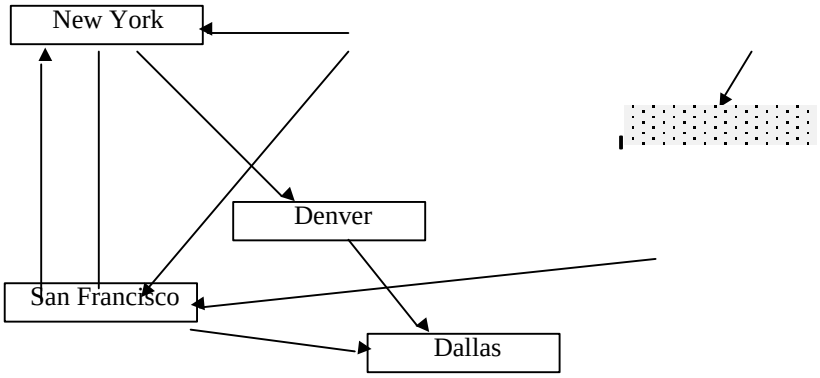
```
8
14
21
17
36
32
49
```

Main lessons learned:

•

```
1 <D8 03> push 3
3 <D8 03> push 3
5 <DF 00> no-check send +
7 <66> pop
8 <D8 05> push 5
10 <CC 00> no-check send factorial
12 <45> pop; push self
13 <65> return
```

In conclusion, let's note that both Parser and Scanner are in the library and you can subclasses



Chicago

We leave it to you to implement the algorithm as an exercise.

Main lessons learned:

Chapter 12 - Developing user interfaces

Overview

The group of classes supporting graphical user interfaces (GUIs) is one of the most complex parts of VisualWorks and its detailed coverage is beyond the scope of this book. Fortunately, to be able to use the tools for routine applications and to create interesting extensions of the widget library, you only need to understand a limited number of concepts, and specialized books are available for more advanced uses.

The first requirement for creating a user interface is to be able to draw objects such as lines, widgets, and text on the screen. VisualWorks perspective is that drawing requires a surface to draw on (such as a computer screen or printer paper), the graphical objects being drawn, and an object that does the drawing and holds the parameters that describe the drawing context (such as fonts, colors, and line widths).

Being able to draw an object on the screen is, however, only a part of a typical GUI. Unless the displayed windows are completely passive, the user must also interact with them via the mouse and the keyboard. A user interface must also ensure that the display reflects changes of the displayed data model, and that damage caused, for example, by resizing a window is automatically repaired.

To implement automatic response to model changes and user interaction, Smalltalk uses the model-view-controller paradigm - MVC for short. The model part of the MVC triad holds data and provides

GUIs-ay reflece to drulical objsxt (suculine wonrks n Tj -36 -1.28 TD /F0 9.96 Tf -0.002 Tc70.057 Tw222The firsoma

To deal with the two different modes of operation that affect the result of clicking a desk button (switching to a different desk versus renaming it), we must also keep track of the mode. We will represent it by a Symbol which will be identical to the name of the method that executes the operation so that we can execute the method by the

        - moveTo: is triggered by the *Move* button via the move method (below). It displays prompts to select the window to be moved and the destination desk, unmaps the window, and assigns the number of the specified desk to this window in the registry.
    - openBrowser and openWorkspace buttons open the Browser and the Workspace.
    - rename requests a new name for the currently selected desk button and assigns it to the button as its label.
    - move asks the user to select a window and click the destination desk button. The internal assignment of the desk is performed by updateDesk:.
- Private support methods.
    - updateDesk: is trigga tosupp

```
| window |
(waitingApps at: currentDesk) remove: anApplication.cg816
```

aRectangle displayOn:

```
gc paint: paint.
startx := (Dialog request: 'Upper left x' initialAnswer: '30') asNumber.
starty := (Dialog request: 'Upper left y' initialAnswer: '30') asNumber.
side := (Dialog request: 'Side' initialAnswer: '20') asNumber.
“Disply.”
^gc displayRectangle: (startx @ starty extent: side @ side)
```

Main lessons learned:

- All display within a window is performed by an instance of GraphicsContext.
- GraphicsContext is central to everything related to display of visual components.
- In addition to being the drawing engine, a GraphicsContext also holds display parameters such as line width, paint, and fault.
- Display parameters stored in the graphics context can be controlled programmatically.

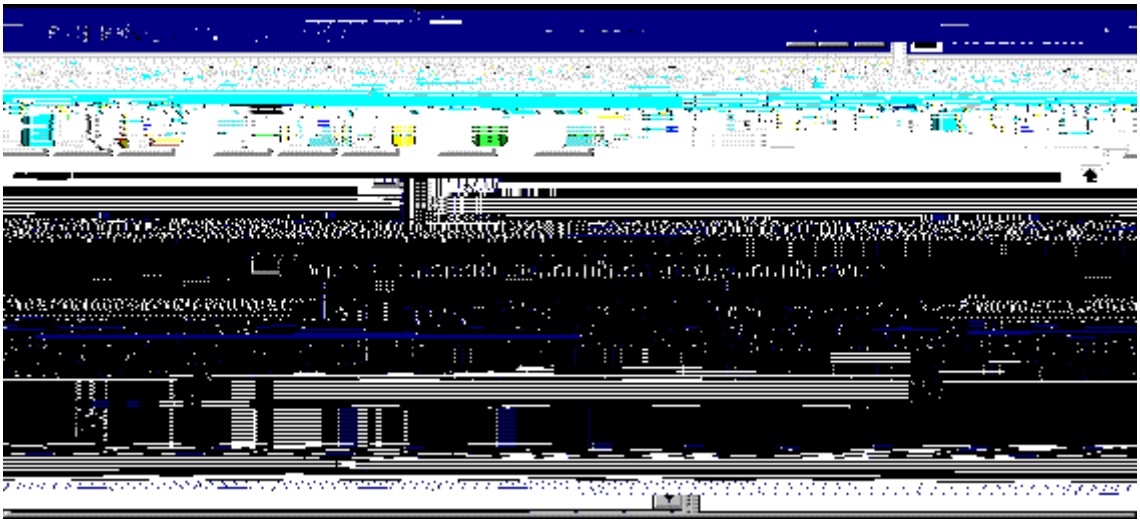


Figure 12.4. Example 2: Displaying image from user in the active window.

Solution: To get an image from the user send message

Exercises

1.

circle and the application changes its diameter, we expect that the circle will be automatically redrawn. The previous sections do not give any hints as to how this could happen but we will see that the principle of *dependency*

Main lessons learned:

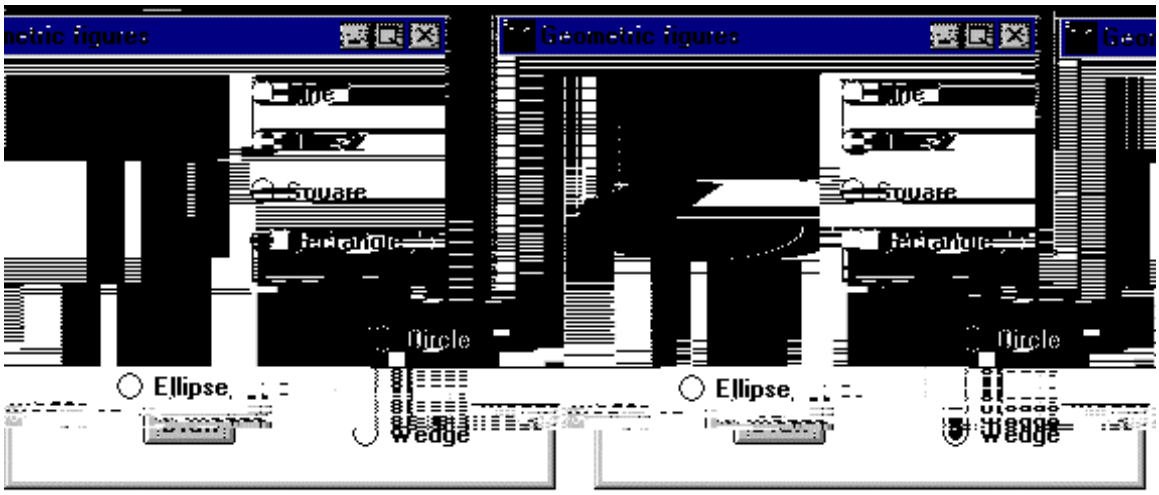
```
image1 := Image fromUser.
Dialog warn: 'Select the second image.'.
image2 := Image fromUser.
imageView := ImageView new.
imageView model: self      "Create dependency to the subview
```

```
                width := aGraphicsContext widthOfString: string.
                aGraphicsContext displayString at: (center x ( width/2)) @ 50)]
        ifFalse: “Display currently selected image.”
                [aGraphicsContext displayImage: image at: 0@0]
```

The program is now fully functional. Note that unlike our previous programs it automatically repairs window damage due to collapsing, reshaping, and other window events.

Improvements

We know that it is better to centralize shared behavior and in our case both image


Geometric figures
Line
Square
Circle
Ellipse
Wedge
Geometric figures
Line
Square
Circle
Ellipse
Wedge

the following example, we will create a UI component with an active user interface and show how to create an active controller that responds to user input.

Example: Subview with clickable hot spots

Problem: Implement an application that displays a bordered subview (Figure 12.12) equipped with an

Exercises

1. Reimplement our Example without lazy accessing and evaluate both approaches.
2.

its components with the clipped GraphicsContext. Each component then redraws that part of itself which falls within the clipping rectangle, using the supplied default graphics context parameters or redefining them temporarily.

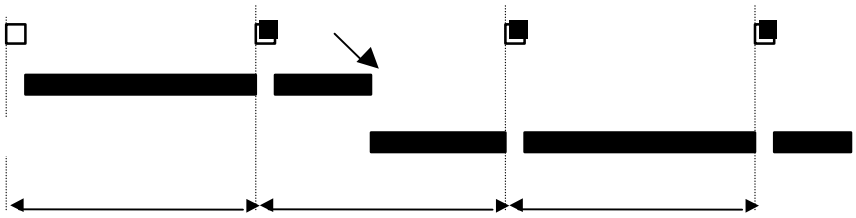
Older versions of VisualWorks used polling controllers which repeatedly queried the sensor for UI events such as mouse movement or button clicks. Since polling requires extra overhead and since UI events occur fast, this approach often missed UI events. Modern implementations still provide polling controllers but add event-driven controllers which obtain event notification directly from the operating system,

Figure 13.2. Simplified behavior of processes: Stopwatch runs until stopped by Delay. At this point another process can start execution and the stopwatch resumes when the one second delay expires.

The *Stop* button stops the infinite iteration process started by the *Start* button by sending it the terminate message:

```
stop
"Terminate the process implementing continuous stopwatch operation."
        process terminate

stopStart
```



the other vehicle's traffic lights to red to prevent collision. Upon leaving the crossing, the vehicle changes

After implementing a stopwatch, the obvious next candid

2. *Program* opens form.
3. *User* enters information and clicks *Accept*.
4. *Program* closes form, inserts new alarm in the list, and displays the updated list in the main window.

Scenario 2: User edits an alarm in the list

1. *User* selects an alarm and clicks *edit*

3. *User*Accept.

3. *Program*1. *User* eects an alarm and clicks .

Design Refinement

Class Hierarchy

AlarmTool is an application model, and its superclass is therefore ApplicationModel. Alarm does not have any relatives in the class library and it will thus be a subclass of Object (Figure 13.6).

13.6. Hierarchy Diagram.

Specification Refinement

Alarm

This should open alarm notifiers in the order 'number 2', 'number 1', and 'number 3' - and it does.

Method addAlarm.

- Remove alarm: terminate selection's process, update list. Method removeAlarm. Alarm
- Edit al32 (Ed38., extract list. M fromprocess, uinatioon's pit, 53.76 process, u.8 TD 7Tc 0 Tf 0.012355 4-128.7

The initialization process is standard; alarmList

By the way, when you open your alarm tool, you may get strange values for starting hours and minutes. To correct this, change you VisualWorks time zone. We leave this as an exercise.

The only remaining action methods are for the *Close* button in the main window, and for *Accept*

Introduction to Smalltalk 7.70yhapter 13.70Processes and their coordinaon t, addion tal UIo Spics

truck process will use the Truck class in a similar way. Process objects thus use domain objects but are separate from them. With this background, we can now expand Scenario 1.

Scenario 1: Starting or restarting simulation
1. *User* clicks *Go*.
2. TrainSimulation creates a train process and a truck process.
3. TrainSimulation starts the train process and the vehicle processes automatically start taking turns executing single step motion while respecting the semaphore at the crossing.

Scenario 2: Stopping simulation
1. *User* clicks *Stop*.
2. TrainSimulation terminates the two processes.

The immediate question is whether clicking *Stop* can actually freeze simulation. The point is that *User* clicks

Figure 13.10. Class Hierarchy Diagram with classes to be designed shown as heavy rectangles.

Refined class descriptions

We now understand enough of the behaviors of our classes to be able to start refining class descriptions:

Domain classes

Vehicle. Abstract superclass of domain classes Train and Truck. Factors out shared behavior and knowledge such as vehicle position and direction of motion, calculation of the next position, testing for the end of the track or crossing, and turning.
Superclass: Object
Components: limit (distance from crossing to lights), position (distance of vehicle from start of track), direction (#incrementing or #decrementing position), vehicleLength (pixels), model (reference to the TrainSimulation object running the sthe

```
"Return true if I have just reached the start of the crossing zone."
        ^(direction == #incrementing and: [position = (crossing - limit - vehicleLength)])
                or: [direction == #decrementing and: [position = (crossing + limit)]]
```

and

```
endOfTrack
"Return true if this is the end of the track. Remember that vehicle might have turned."
        ^(direction = #decrementing and: [position = 0])
                or: [direction == #incrementing and: [position = (trackLength - vehicleLength)]]
```

The method that triggers redisplay of semaphore lights uses the same principle as moveOneStep and asks TrainSimulation to display semaphores controlled by the vehicle in the specified color:

```
semaphores: aColorValue
“Ask TrainSimulation to display sempahores controlled by vehicle identified by its symbol.”
        model semaphores: aColorValue for: self symbol
```

As you can see, Vehicle implements most of the functionality of its concrete subclasses.

```
semaphore signal.
layout := LayoutView new.
layout model: self.
speed := 25 asValue.        "Aspect variable associated with the speed slider."
oKToClose := true
```

[

7. Add an animation of a crash of the two vehicles.
8. Extend the simulation program to several parallel train and truck tracks with a single shared crossing. Each track has its own vehicle running at its own speed and each vehicle's speed is controlled independently. Speed is controlled by a single slider and a set of radio buttons, one for each vehicle.
9. Simulation of events occurring in parallel, such as our train simulation, are common. Define a

5. *User* releases the mouse button.
6. *Program* stops dragging the track.

Scenario 3: User drags the left end of the horizontal track to a new position

1. *User* moves the cursor into the hot area surrounding the left end of the horizontal track.
2. *Program* changes cursor to the shape shown on the left of Figure 13.12.
3. *User* presses the <operate> button and moves the mouse while holding the button down.
4. *Program* drags the end of the track, restricting motion to horizontal displacement and following the cursor without changing its shape. The program also moves the vehicle if necessary and enforces

```
trackLength := model trainTrackLength - shift.
```

File out. Command available in file browsing tools and used to store a method, a protocol, a class, or a category in a file, using a special form expected by the *file in* command. Used for transporting source code from one computer to another. The command is available in various Smalltalk browsers.
Finalization. Actions executed by the program when a window is being closed.
Floating point number.

Message.
Metaclass.
Method.
Model.
Mouse buttons.
Multiple inheritance.
MVC paradigm.
Object.
Object file.
Optimization.
Order of evaluation.
Ordered collection.
Overloading.
Palette.
Pane.
Parameter.
Persostent object.
Pointer.
Pool dictionary.
Po-up menu.
Polymorphism.
Private method.
Primitive method.
Private method.
Protocol.
Pseudovariable.
Radio button.
Receiver.
Return object.
Reusability.
Selector.
self.
Semantics.
Set.
Shortcut key.
Signature.
Simulation.
Single inheritance.

ReSourceolld

Sigta.

Sigta.ic binng.
Sigtrea

Symb.

PaSubass.
PaSuperass.
seluper Tj T* 0.0082 Tc 0.04523Tw (Theempory.) riable.
SiText Tj T* 0.019 Tc 0.011 Tw (ReText edir.)fwidg.

Text file.
Transcript.
UI.
User interface.
Unary message.
Undeclared variable.
Value holder.
Variable.
View.
Widget.
Wildcard.
Window.
Workspace.

References

Books

Kent Beck: Smalltalk: Best Practice Patterns, Prentice-Hall, 1997.
Timothy

Exercises

1. Modify our example 1 to display 'Input text' in the input field and 'Output text'

```
(builder componentAt: #mozzarella) enable.
(componentAt: #figs) enable.
(builder componentAt: #pineapple) enable.
(builder componentAt: #mozzarellaPrice) enable.
(builder componentAt: #figsPrice) enable.
(builder componentAt: #pineapplePrice) enable]
```

Figure A.1.14. Builder's bindings dictionary.

Once the bindings

printAndClose
“Gather answers, close window, and print results in Tal“Gme(e)17.001s, cler commentand the“G4(se w)e8.7n406ow8.7 l“Pnt reem

area may be changed at run time. One of the main uses of subcanvases is in no

-

Figure A.2.6. The main canvas (left), and the two subcanvases. The subcanvas component of the main

```
dirString isEmpty | (dirName := dirString asFilename) isDirectory not
        ifTrue: [^Dialog warn: dirName asString , ' is not a valid directory.']
        ifFalse: [directoryString := dirString].
                self builder window label:
                        ('Files in directory: ' , directoryString copyWith: Filename separator).
                divider selectionIndex: 1.
                alphabet selectionIndex: 1
```

Figure A.2.17. Passive (left) and activated (right) menu button. The passive view may display the current choice in the menu. The menu bar at the top of the window has one label.

The distinguishing characteristics of popup menus, menu bars, and menu buttons are summarized in the following table:

type of menu behavior type of GUI component placeme10(t3e:)]ET1 i 87.2.51643.08 1.43614 56 ref87.2.51

Figure A.2.21. Dialog for installing a menu.

In our case, the automatically generated resource definition method is

```
testMenuHolder
"MenuEditor new openOnClass: self andSelector: #testMenuHolder"
        <resource: #menu>
```

3. Use the Menu Editor to color the background of the *delete* commandddadd

Appendix 3 – Chess board – a custom user interface

Overview

Context Diagram

The major components of the system are the chess board with its pieces and user interface, and the players. The players (including a computer program player) are outside of the scope of the task. The Context Diagram is shown in Figure A.3.2.

Appendix 3 – Chess board – a custom user interface

iii. How many moves are required for a knight to reach a given square from a given starting square?
iv. What is the maximum number of moves for a knight to reach any position on the board from a given starting square?

Metaclasses are, of course, objects and have their own class hierarchy, perhaps even parallel to their classes.

A 4.2. What is the complete class hierarchy?

```
String subclasses          "Returns #(ByteEncodedString TwoByteString Symbol GapString)."
```

To find which of all existing arrays has the largest size, evaluate

```
| size |
size := 0.
Array allInstancesDo: [:anArray| size := size max: anArray size].
size
```

or more simply`Array allInstances inject: 0 into: [:max :anArray| max max: anArray size]`

To find all superclasses of a class, execute allSuperclasses as in`Set allSuperclasses          "OrderedCollection (Collection Ob`

Instance variable format contains a code that describes what kind of class this is. The protocol based on format makes it possible to ask a class whether it has fixed size (classes that have only named variables) or variable size (collections represented internally as indexed elements) and, in the case of a variable size class, whether its variables are stored as eight-bit or 16-bit quantities. As an example, all variable size classes can be obtained by

```
Object withAllSubclasses select: [:aClass| aClass isVariable]
```

which returns

```
OrderedCollection (AnnotatedMethod Array BinaryStorageBytes BOSSBytes BOSSReaderMap ByteArray
```

Functionality based on access to class hierarchy and method dictionary

Behavior provides several methods for adding new selectors to the method dictionary or removing

Exercises

1. The all-important

```
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Support'
```

Accepting the edited template simply sends message subclass: instanceVariableNames: classVariableNames: poolDictionaries: category:

Exercises

Proxy to insert a programmer action and pass the original message to x, the problem is solved (Figure A

```
TestOfProxy newWithProxy var: 13; var: 15
```

Important classes introduced in this chapter

ClasseTJ w,-916.8(D-2(sseTJ nCla)-5.mcl)1sseTJ ClaeTJ er

Guideline N.7: A*rgument names when type is repeated.*

A 5.4 Formatting

Guideline F.1: Use automatic formatting. Since the VisualWorks formatter exhibits some undesirable

We want to be able to perform the following tasks:

- Patient record
    - Add
    - Delete
    - Display
    - Edit
    - Print
    - Create appointment
    - Cancel appointment
    -

High level conversation:

1. *User* selects borrower in the catalog and clicks *Open*.
2. *System* opens a book form with current information about the book.
3. *User* edits the form and indicates end of task by clicking *Accept* in.

<comment> ::= <commentDelimiter> <nonCommentDelimiter> <commentDelimiter>
<binaryCharacter> ::= ‘+’|’/’|’\’|’*’|’~’|’<‘|’>‘|’=‘|’@’|’%’|’|’|’&’|’?’|’!’|’,’

added to a stream and displayed in the text view, whereas non-normal characters are ‘dispatched’ for further processing. This processing consists of checking ParagraphEditor’s dispatch table, an instance of DispatchTable

global interest), or class variables, or keys in a pool dictionary[1]. It turns out that they are keys in the pool dictionary TextConstants because they are used by several other classes as well. When you inspect this dictionary, you will find that Ctrlt ,Ctrlf, Cut, Paste, and BSC are among many characters that are assigned

A ComposedText contains Text and has access to a dictionary of text styles via its TextAttributes component (Figure A.8.2). The TextAttributes object does not itself contain the style dictionary but has access to a

```
(fontSize := Dialog          choose: 'Which font size do you want?'
```

that use it. When defining new text styles, a new instance of

Over: #overRleser
Exit: #exit
Dal