MiTTS:
Minimalist Tutorial and tools for Smalltalk/V

August 9, 1993

Computer Science Department
IBM T.J. Watson Research Center
Yorktown Heights, New York

This document is part of an introductory tutorial for Digitalk Smalltalk/V™; It is used in conjunction with a tutorial diskette containing the tutorial software. This document and the accompanying software are the property of IBM Corporation. Contact: Mary Beth Rosson, 914-784-7738

Contents

The Smalltalk Mountain

This tutorial is deliberately different from other programming instruction you have seen. It does *not* start off with a long drill on syntax and an example program that prints "Hello World." Instead, it allows you to experience Smalltalk as an integrated software environment for object-oriented programming. In this minimalist tutorial you learn Smalltalk by analyzing and enhancing a simple but real application, a blackjack card game. You climb the Smalltalk Mountain by starting at the top!

Our goal is to let you see fairly quickly what is different and special about Smalltalk, and then let you go back and pick up the details. The important differences between Smalltalk and procedural languages like C are not *in* the syntax. They are deeper than that - in the very concept of what a program is, how it runs, and its relationship to the environment within which it runs. A Smalltalk program is a collection of interconnected software objects - bundles of function and data - spawned from classes distributed across a huge code library.

The main programming tasks in Smalltalk are *not* typing, compiling, and linking program modules - instead programmers search for useful classes in a rich code library (the Smalltalk class hierarchy) and design new applications by specializing these classes and building connections among them, using interactive tools to test and debug an application as it develops. Thus we focus on application design (including user interface design) and on the use of interactive programming tools.

Be prepared to *not* see code listings! In some cases we have "hidden" some or all of the code to help you focus on application design and programming tools. Our goal is to give you a concrete idea of Smalltalk as a highly integrated, object-oriented programming language and environment, after just a few hours of work with the example application.

To begin, you will need to have installed Smalltalk/V PM (the Smalltalk manual explains how to do this). You will then be creating a new image to use for the MiTTS tutorial.

Introducing the Blackjack Game

Before you begin working with the blackjack application, we want you to see what it's all about. To this end, we have created a short animation of the game, so that you can see how the game is played. But first, you will need to set up a new Smalltalk image.

This tutorial assumes that you have already Followed THC installation directions for Smalltalk/V PM. The following directions assume that you have installed Smalltalk into a directory named `\vos2`.

◊ In an OS/2 window, create a new directory `\mitts`, and make that directory your current directory.

◊ Copy all of the files from your MiTTS diskette into your new `\mitts` directory, using the command
`copy a:*.*`

Ten files (v.exe , vpm.exe. change.log , mouseptr.dll and six topic*.cg files) are copied.

◊ Type `start vpm` and press enter

Smalltalk V PM starts up. The Transcript window appears.

Hint

If Smalltalk doesn't start up correctly, you may need to update the LIBPATH setting in your `config.sys.` file. This path should include a period, so that the current (`\mitts`) directory will be searched, as well as the directory in which you installed the `dll` files for Smalltalk/V PM 2.0 (e.g., `c:\vos2`). Remember that you will need to reboot your system for changes to `config.sys` to take effect.

◊ In the Transcript window, type and select (by swiping through the text with the mouse while holding down the left button):
`MiTTS startTopic: 0`

> **Hint**
>
> Upper and lower case matter in Smalltalk. Type the expression exactly as it appears in the instructions.

◊ With the mouse cursor in the Transcript window click the right mouse button.

The pop-up text-editing menu appears.

◊ Select "Do it"

The Transcript window shows "Filing in Topic 0."

A dialog box appears, informing you that a demo is about to begin, and asking you to refrain from moving the mouse during the demo or ending it prematurely.

◊ Click on the OK button in the dialog box, and then remove your hand from the mouse.

The blackjack window appears, and several possible rounds are played out. The goal of the game is to get as close to 21 as possible, without going over. The animation shows the player taking a hit, deciding to stay, and the dealer playing out the round. It also shows other possible scenarios, for example a case in which the player gets a blackjack (a score of exactly 21) on the initial deal.

◊ You may play the demo over as many times as you like: simply re-select and "Do it" to:

```
MITTS startTopic: 0
```

Topic 1: Exploring the Blackjack Game

In Smalltalk, everything happens through objects sending messages to one another. To write a Smalltalk application, you specify the objects you will be using, the messages they can respond to, and how they are related to one another.

Smalltalk provides a large starting set of object types (called *classes*; see the Glossary for definitions of terms italicized in the text) to use in building applications. Thus an important part of programming consists of discovering, using or enhancing these existing classes, and the messages implemented for them (called *methods*). The system provides a Class Hierarchy Browser to help you in this.

The Smalltalk/V PM class library is huge (almost 200 classes, over 4400 methods), which makes it hard to know which ones to examine and try out. The Bittitalk Browser was designed to help you get started; it presents a filtered view of the class library, making it easier to find things. This filtered set of classes is used for a blackjack game that you can play and work with to begin learning and using Smalltalk.

Starting up the Bittitalk Browser

◊ In the Transcript window, type, select and "Do it" to:

```
MiTTS startTopic: 1
```

Another file is read in, and a Bittitalk Browser opens on the screen. The upper left pane of the Browser displays a list of classes involved in the blackjack game you'll be exploring. In a moment, you will be creating an instance of the Blackjack class to work with.

◊ Select the Blackjack class in the Browser.

The upper right pane lists the names of methods defined for the Blackjack class. A method is the code evaluated in response to a message; you can think of each method name (also called a selector) as a message that Blackjack instances can receive. If you select a name in this list, the method name and a comment describing its functionality will appear in the lower pane.

One or more colons in the method name indicates that it is a *keyword method*, and takes an argument after each colon.

In the standard Class Hierarchy Browser, the bottom pane would also display the code associated with a method; this version of the Bittitalk Browser does not display method code. Right now, you should try to understand what the methods *do*, and not worry too much about how they work.

The two radio buttons at the center top allow you to toggle between *instance methods* and *class methods*. Because a class is an object, it can also receive messages; the class methods represent the messages that a class object can receive. However, most methods in Smalltalk are instance methods. Normally you send one message to a class (to create an instance of it), then all subsequent activity involves messages sent to that instance.

The Blackjack game

This blackjack game has two players, the player and the dealer: you will be the player, and the system will be the dealer. You begin by "dealing" — each hand receives two cards, with the first card for the dealer dealt face down. You can then ask for a "hit" (an additional card) as many times as you want — your goal is to get as close as possible to 21 without going over.

Face cards are worth 10 points, and an ace can be worth 1 or 11 points, depending on what leads to the best score. When you're satisfied with your hand, you "stay." The dealer's hand is then "played out" — the rules require the dealer to take new cards until reaching or exceeding 18 points.

If a player's hand exceeds 21, it "busts" and loses the round; if a hand reaches 21 exactly, it gets a "blackjack" and wins the round. Otherwise, the hand with most points wins.

Creating an instance of Blackjack

You can play the game by sending messages to an *instance* of the Blackjack class. This instance will play the dealer's hand, and serve as the game administrator.

◊ Move the pointer to the File menu in the menu bar of the Bittitalk Browser and click the Left mouse button.

You see the File menu. One of the choices is "New Workspace."

◊ Move the pointer to "New Workspace" and click the left mouse button.

A Workspace opens. Workspaces can be used to create instances of a class and to send messages to them.

> **Hint**
>
> If you want to move the workspace window or change its size, look in the appendix named "Interaction Tips."

◊ Make sure the Workspace window is selected (by clicking on it). Then type, select and "Do it" to:

```
Game1 := Blackjack newGame
```

The expression is evaluated. `newGame` is a *class message*, sent to the Blackjack class. The message creates a new instance of the Blackjack class. Can you locate this class message in the Bittitalk Browser?

The new Blackjack instance is assigned to the global variable `Game1`. All global variables begin with a capital letter.

Sending messages to the Blackjack instance

Now that you have created an instance of Blackjack, you can play the game, by sending it instance messages.

◊ In the workspace, type and evaluate with "Show it":

```
Game1 deal
```

An array containing the player and dealer hands is returned. (If you or the dealer get a blackjack, this round! is over, and you can start another by re-sending the deal message).

`deal` is a Blackjack *instance message*; `Game1` can respond to any of the instance messages defined in the Blackjack class.

"Show it" evaluates the expression, like "Do it," but it also prints the object *returned* by the expression. All messages return an object when they are evaluated, in addition to whatever other effects they have, and one way to test what a message docs is to see what it returns.

Hint

Your workspace will be more readable if you start each message expression on a new line.

If the text describing the returned object extends beyond the window border, look in Appendix A, "Interaction Tips" to find out about scrolling or resizing windows.

◊ In the Bittitalk Browser, find the message you can send to `Game1` to request a hit for the player. In the workspace, type the appropriate message expression, select and show it.

The updated player hand is displayed. You can take another hit if you have fewer than 21 points. But be careful — you don't want to bust! If you bust or get blackjack, deal a new hand to start another round.

◊ In the Browser, find the message you send to a Blackjack instance to end the player's turn: type, select and show it.

The dealer's first card is turned over.

◊ In the Browser, find the message you send to a Blackjack instance to complete the dealer's hand: type, select and show it.

The dealer's hand is played out and the winner is reported. You can follow these steps as necessary to play a series of blackjack hands. Since you've already typed the Smalltalk messages needed to play the game, all you need do now is select and evaluate the message expressions in the desired order.

◊ Find and then send to `Game1` the message that will return the current game score. Are you winning'?

Exploring other objects and messages used in Blackjack

Most of the work in the blackjack game is done by five objects — the instance of Blackjack (`Game1`), the dealer and player (instances of BJPlayer) and the two hands (instances of BJHand). The Blackjack instance controls the game. But the two BJPlayer instances and their

hands do a lot of work too. Other supporting objects include an array used to hold the score, and a shuffled deck of cards (see Figure 1).

◊ In the Bittitalk Browser, explore the instance methods defined for BJPlayer.

The BJPlayer instances take cards from the Blackjack instance, and report blackjacks or busts when they occur.

◊ Explore the instance methods defined for BJHand.

You can see in Figure 1 that both of the BJPlayer instances "own" an instance of BJHand. These hands hold the cards, and can report their current point value.

Notice in the Browser that BJHand is part of the `Collection` class hierarchy (the hierarchy is reflected in the indentation structure). As a subclass (or specialization) of the `OrderedCollection` class, it *inherits* (can respond to) messages defined in any of its superclasses. For example, it uses the `add:` message to add new cards to itself.

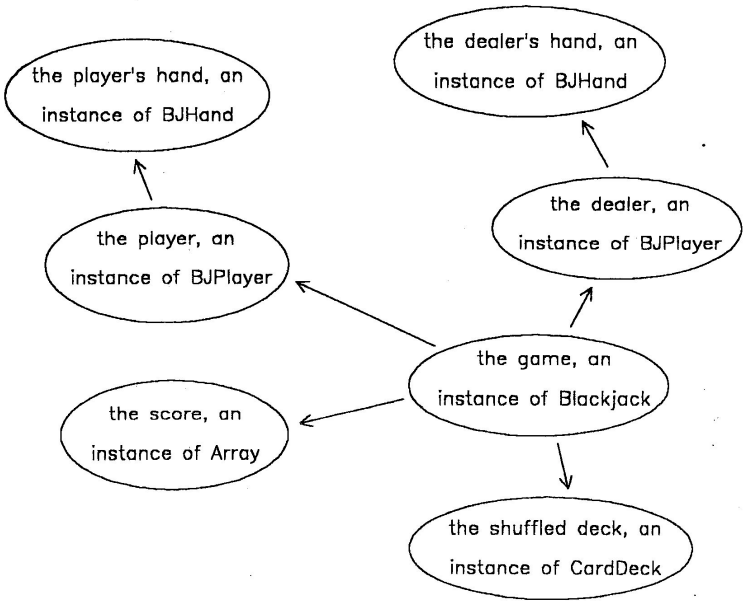

Figure 1. Objects participating in the blackjack game. The nodes represent instances of a class and the arrows show how the different objects are connected. E.g., the Blackjack instance points to the player, the dealer, the score, and the card deck.

◊ Explore the methods defined for Card and CardDeck.

An instance of CardDeck starts out with a collection of 52 Card instances. These instances of Card share the structure and behavior defined in the Card class, but each has its own suit and value combination.

Notice that instances of CardDeck can be created that are either shuffled or not. The blackjack game uses a shuffled deck.

In this Topic, you have seen the basic classes used in the blackjack game, and some of the messages that instances of these classes can respond to. In the workspace, you sent messages to `Game1`, an instance of Blackjack. Using the communication links displayed in Figure 1, this object then sent appropriate messages to other objects (e.g., the card deck, the player, the player's hand). Thus, although you sent just a few messages to play a round, the objects depicted in the figure were doing a lot of work to carry out your requests.

◊ When you've finished exploring the Blackjack classes, close the Browser and Workspace windows (you needn't save the Workspace contents).

Topic 2: A User lnterface for Blackjack

One of the most important strengths of Smalltalk is its support for programming graphical user interfaces. The standard class hierarchy includes many classes and methods for building user interfaces; even a programmer new to Smalltalk can take advantage of them to quickly develop an interactive application.

We created a special tool — called the View Matcher — to help you understand how interactive applications work in Smalltalk. The View Matcher provides an integrated set of views of a running application. In this Topic, you will be using the View Matcher to explore a user interface to the blackjack game you analyzed in Topic l.

Starting up the View Matcher

◊ In the Transcript window, type and evaluate with "Do it":

```
MiTTS startTopic: 2
```

You are looking at the View Matcher. In the upper right corner is the space that the blackjack game will use. The lower right corner will hold a Bittitalk Browser like the one you used in Topic 1. We will describe the left half of the View Matcher as we go.

Playing Blackjack interactively

Before you analyze how blackjack's user interface works, you should first see what it looks like.

◊ Open the "Applications" menu on the View Matcher menu bar, and select "Blackjack."

A blackjack window opens in the upper right corner of the View Matcher. This is the same blackjack game you played in the workspace, but now the player and dealer hands will be displayed in separate *subpanes* of the window, and you will play the game by making menu selections.

The Bittitalk Browser also updates to display the classes used in the blackjack game. The classes you explored in Topic 1 are there, as well as those needed by the user interface.

◊ Open the "Dealer" menu in the blackjack menu bar and select "deal."

The player and dealer hands are dealt and displayed in their respective subpanes.

◊ Play as many rounds as you like. The player and dealer menus can be requested either from the menu bar, or by clicking the right mouse button in the appropriate subpane.

In Topic I, you used a Workspace to send the `deal`, `hitMe`, `stay`, `playItOut` and `score` messages directly to a Blackjack instance named `Game1`. Here, these messages are sent indirectly to the Blackjack instance, via the user interface objects. The instance of Blackjack serves as the application *model* "behind" the user interface. ln its role as the application model, the Blackjack instance manages other game objects (e.g., the players, the score). Notice in Figure 2 that design of the underlying application (the objects enclosed within the dash lines) is identical to the design you explored in Topic 1.

The user interface objects are also managed by a single object, an instance of BlackjackView. As can be seen in Figure 2, .the main components of the user interface are a window (an instance of TopPane), and two subpanes (instances of TextPane). All communication between the user interface and the model is channeled through the BlackjackView and Blackjack instances.

When a menu selection is made in the blackjack game, the BlackjackView receives the request from the menu and sends a message to the Blackjack instance; the Blackjack instance evaluates the message, and if necessary, sends out a request for the user interface to be updated.

For example, in Topic 1, you sent a `deal` message to `Game1` to deal new hands for the player and dealer. In this version, the instance of BlackjackView receives this request from the user and forwards it to the game. The Blackjack instance then evaluates the `deal` message, dealing the two hands, and then tells the BlackjackView to update the subpanes. Using the View Matcher, you can now explore how this happens.

How is a deal request made?

Everything in Smalltalk happens as the result of messages sent among objects - when a given message is received, the object often responds to it by sending more messages, either to itself or to other objects. The View Matcher is designed to help you explore this message passing.

The messages sent in the course of using an application are added to and removed from a *message execution stack*. When an object sends a message to another object (or to itself), a message is placed on the stack.

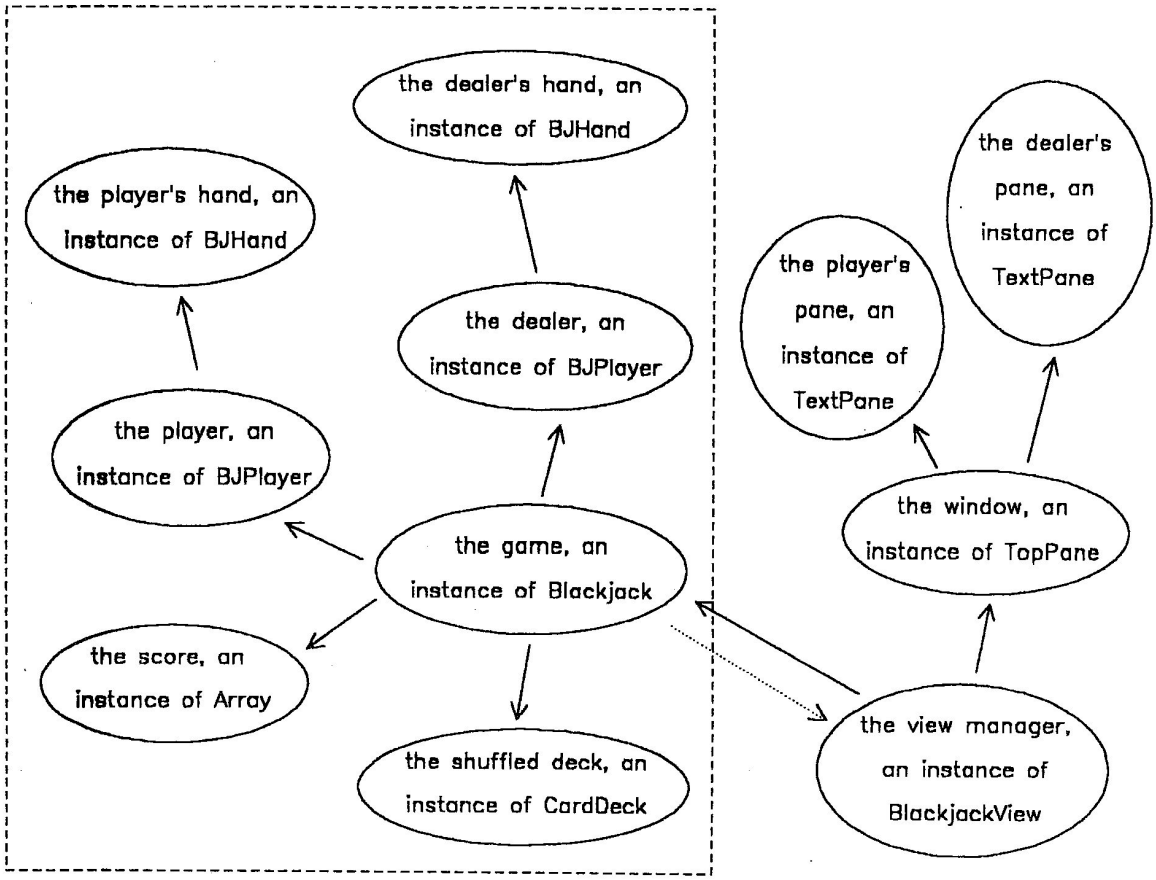

Figure 2. The interactive Blackjack game. As the dashed line indicates, the application is structured as it was in Topic l, but now the (blackjack instance is also connected to an instance of BlackjackView, which in turn points to a TopPane containing two TextPane. In this application, all communication between the user interface and the game takes place through the Blackjack View -Blackjack connections.

When evaluation of a message involves the sending of other messages, these messages are added to the stack and the original message stays on the stack until all embedded message sends have been processed. We have instrumented the blackjack game with a number of breakpoints to let you explore the messages placed on the message execution stack. When the View Matcher is in **Halted Stack Mode**, the game will stop automatically at these breakpoints, and the execution stack will be

displayed in the upper left pane. The breakpoints were chosen to let you explore how the user interface components and the underlying blackjack game cooperate in responding to user requests.

◊ Open the "Applications" menu on the View Matcher menu bar and select "Halted Stack Mode."

The View Matcher title bar updates to show that it is now in **Halted Stack Mode.**

◊ Select "deal" from the blackjack dealer menu.

The game stops in the midst of processing your menu request. *Note that the player and dealer panes still display the cards from the previous game.* We have put a "halt" request at the beginning of the method code evaluated in response to the deal message. As a result, the message has been placed on the stack, but no cards have been dealt, and the displays haven't been updated.

Each line on the stack lists a message and the *class* of the object that received the message. The most recent message (at the top) is `Blackjack»deal`; the halt occurred when `deal` was sent to an instance of Blackjack. Underneath it is the stack of messages that were already on the stack when the halt occurred. Because this is a stack, you know that each message on the stack was sent in the process of evaluating the message below it.

◊ Select the second item in the stack: `BlackjackView»deal`

A `deal` message has also been received by a `BlackjackView instance.` When it received the `deal` message, the BlackjackView instance responded by sending `deal` to its game, the Blackjack instance.

Hint

Notice that the View Matcher's title bar says "STOPPED," informing you that the blackjack game is at a halt point. You cannot make blackjack menu selections when the game is stopped.

The Bittitalk Browser in the lower right corner is synchronized with the selection of messages in the stack, making it easy to find corresponding method in the class hierarchy (as in Topic 1, the browser doesn't show you the method code, just its name and comment). The pane on the lower left is also synchronized with the stack, providing special View Matcher commentary on the role of the selected message in the blackjack application.

◊ Explore the messages in the stack, reading the View Matcher commentary and the method comments in the browser for each, and trying to trace through the structure of a blackjack menu request. See if you can identify these chunks of activity:

— The window (an instance of TopPane) containing the dealer ' and player panes is asked to process a menu item; it delegates this request to its menu bar, an instance of MenuWindow.
— The menu bar converts the user's selection into the appropriate menu request and sends the corresponding message to its owner, the BlackjackView instance.

> **Hint**
>
> Some of the message lines include a second parenthesized class name following the class name of the receiver, e.g., `BlackjackView(Object)»perform:`. As you can see by checking the Bittitalk Browser, the parentheses indicate that this message is inherited from a superclass.

— The BlackjackView instance then forwards the request to its game, the Blackjack instance.

How is the result of dealing displayed?

As the `deal` message is evaluated by the Blackjack instance, cards are dealt to the two hands. The panes that display the hands must then be updated. But because the Blackjack instance has no direct communication with the subpanes, the updating must be managed by the BlackjackView instance.

When the BlackjackView instance was created, it was given a special role as a *dependent* of the Blackjack instance; in Figure 2, this special relationship is depicted as a dotted line. As a dependent of the blackjack game object, the view manager object will be notified when the Blackjack instance announces that it has changed in some way.

◊ With the pointer in the message execution stack area, click the right mouse button to get the single-item resume menu and select "Resume"

The Blackjack instance begins to evaluate the `deal` message, and the execution stack reflects the next predefined halt point. At the top of the stack now is the `showHand` message received by an instance of BJHand. When evaluated, this message will return the new player hand, formatted for display.

◊ Find and select the `Blackjack»deal` message that had been at the top of the stack before.

The Blackjack instance has not yet finished evaluating the `deal` message. Above it are the messages stacked in the process of getting to the `showHand` request.

◊ As before, work your way up the stack, selecting each message in turn and reading the View Matcher commentary and the method comments. Rather than trying to understand all of the details, try to extract the general schema for updating the blackjack subpanes. There are several major chunks of activity:

— After dealing new cards, the Blackjack instance sends itself a `changed:` message; the argument to the message identifies which subpane is affected by the change.

— The Blackjack instance iterates through its dependents, sending each an `update:` message. In the case of this singlewindow application, the BlackjackView instance is the only dependent. The argument in the `update:` again identifies the user interface component needing to be updated.

Hint

Message lines beginning with brackets ([]) refer to the stacking of a block of code within a method (typically an iteration of some sort). For example, the line
`[] in Blackjack(Object)»changed:`
should be read as "the block of code in the `changed:` method sent to an instance of Blackjack." This particular block of code was stacked when the Blackjack instance began iterating through its dependents.

— The BlackjackView responds to the `update:` message by sending itself a `changed:` message. This causes it to iterate through the views that it manages, sending each view the same `changed:` message.

— When the TopPane instance receives the `changed:` message, it recognizes the argument as the name of the player's pane, and sends an `update` message to the appropriate TextPane instance.

— The TextPane instance sends itself an *event* requesting new contents. Most subpanes have events that they can respond to; these events are defined when the subpane is initialized.

— The BlackjackView instance receives the `playerPane:` message. This is the name of the "get contents" method defined for the player's pane. The actual hand information is obtained by sending the `showHand` message to the player's hand. See if you can follow the communication paths in Figure 2 to see how this request must be sent.

◊ Select "Resume" from the resume menu.

The `showHand` message is evaluated, the newly dealt player hand is displayed, and execution continues to the next halt point.

We have instrumented the blackjack application so that it stops at significant user interface transactions — the processing of menu requests and any resulting display updates. In this case, the application stops prior to the next display update, when the `showHand` message is sent to the hand of the dealer, to get a list of its new cards.

How does the player take a hit?

When you played the blackjack game in Topic l, you sent the `hitMe` message to the Blackjack instance when you wanted to take another card. Using the View Matcher **Halted Stack Mode**, you can see more about how this happens.

◊ Resume the game to a point where you can request a hit..

◊ Select "hitMe" from the player's menu.

The game halts in the midst of giving a card to the player. At the top of the message execution stack is `BJPlayer»takeCard:`. What object sent this message to the BJPlayer instance?

◊ Explore the stack, using the commentary and Browser information to understand the chain of messages involved in taking a hit. At what point has the request been transferred from the user interface to the application model?

◊ Continue to play the game for as long as you like, using the commentary and method comments to find out about other blackjack transactions.

How did Blackjack's user interface get set up?

You've been exploring the communication that takes place among the objects in the interactive blackjack game. You can now explore the point at which the relationships among these objects get set up.

◊ Close the current blackjack game by selecting "End application" in the View Matcher application menu.

The application pane and Bittitalk Browser are emptied.

◊ Start up a new blackjack game.

This time the game halts in the midst of being opened, before the window appears.

◊ Select the (single) message in the stack, `BlackjackView»openOn:`

The Browser and commentary update to provide information about the role of this message in starting up a blackjack application.

◊ Resume the application.

The application halts again, this time when `showHand` is sent to a BJHand. Use the stack and commentary to figure out what is happening at this point.

Notice how events are used in opening a window; every pane in the window sends itself a `#getContents` event, which results in sending some predefined message to the subpane's *owner* (usually the view manager). Look in the stack for the message sent to the BlackjackView to get the contents for the player pane.

◊ Resume again.

The application halts, and the stack indicates that `showHand` has again been sent to a BJHand. What is happening here?

◊ Resume one more time.

The BlackjackView has completed the opening process, and you can now interact with the blackjack game as before (you can turn off halted stack mode at any time using the View Matcher application menu).

◊ When you have finished exploring the blackjack user interface, close the View Matcher window.

Topic 3: Enhancing the Blackjack Game

In Topic I, you sent messages — `deal`, `hitMe`, and so forth — to `Game1`, an instance of Blackjack. `Game1` responded by executing the corresponding methods, where the code of a method simply consists of a sequence of message expressions. In this topic, you will analyze the code of some of the Blackjack methods, and learn how to parse Smalltalk message expressions; you will then enhance one of the methods.

Starting up the Blackjack game

◊ In the Transcript window, start up Topic 3.

Some files are read in, then a Bittitalk Browser opens. In this topic, you will be using the version of blackjack without a user interface, so the user interface classes are no longer in the Browser.

◊ In a workspace; create a new blackjack game and set up a new hand.

Parsing the Blackjack hitPlayer method

The method code evaluated in response to a message is simply a set of expressions itself. But these expressions can be arbitrarily complex, and it is important to learn how to parse them.

◊ In the Bittitalk Browser, select the Blackjack class.

As in Topic 1, the upper right-hand pane lists the instance methods defined for Blackjack. The middle pane beneath the instance/class radio buttons lists the *instance variables* defined for the Blackjack class, as well as three *class variables* inherited from the Object class. The lower pane displays the Blackjack class definition.

◊ Select the `hitPlayer` method.

As in Topic 1, the lower pane displays the method name and comment. But it now also displays the code defined for this method.

◊ Look at the first message expression in the `hitPlayer` method:

```
playerInPlay isNil
      ifTrue: [^self noHandsDealt ].
```

The instance of Blackjack receiving the `hitPlayer` message checks to see if a player is in play; if not, it sends itself the `noHandsDealt` message, which prompts the user to ask for a new deal.

Message expressions can run over an arbitrary number of lines. If you have multiple expressions, each must be terminated with a period.

◊ In the Workspace, type and show:
```
Game1 playerInPlay.
```

"a BJPlayer" is returned; which of the two instances do you think it is? All instances of BJPlayer print themselves using the default format of "a" or "an" followed by the class name.

The BJPlayer instance is the current value of an *instance variable* defined for the Blackjack class, named `playerInPlay`. When a programmer wants to make the value of an instance variable accessible, it is standard to provide an access message with the same name as the variable. Is there an access message defined for the `playerInPlay` variable?

Look in the instance variable list (under the instance/class toggle) to see the other instance variables defined for this class. You can use Figure 3 to see how these variables are used. If you select one of these variables, the browser will filter the message list to include only methods that refer to that variable.

Like methods, instance variables can be inherited; another instance variable defined for Blackjack instances is `shuffledDeck`. In which of its superclasses is this defined?

Hint

To retrieve the full list of instance messages, re-select the instance radio button.

◊ In the workspace, type and show the expression:
```
Game1 playerInPlay isNil
```

`false` is returned. In Smalltalk, `false` is an object like everything else, an instance of the class False.

This expression is an example of the message chains you often see in Smalltalk code. The `playerInPlay` message sent to `Game1` returns a BJPlayer instance, which is then sent the `isNil` message. When the BJ Player receives the `isNil` message, it returns the object `false`. In other words, the expression can be parsed as `(Game1 playerInPlay) isNil.`

"Show it" simply provides a printable view of the object returned at the end of the message chain.

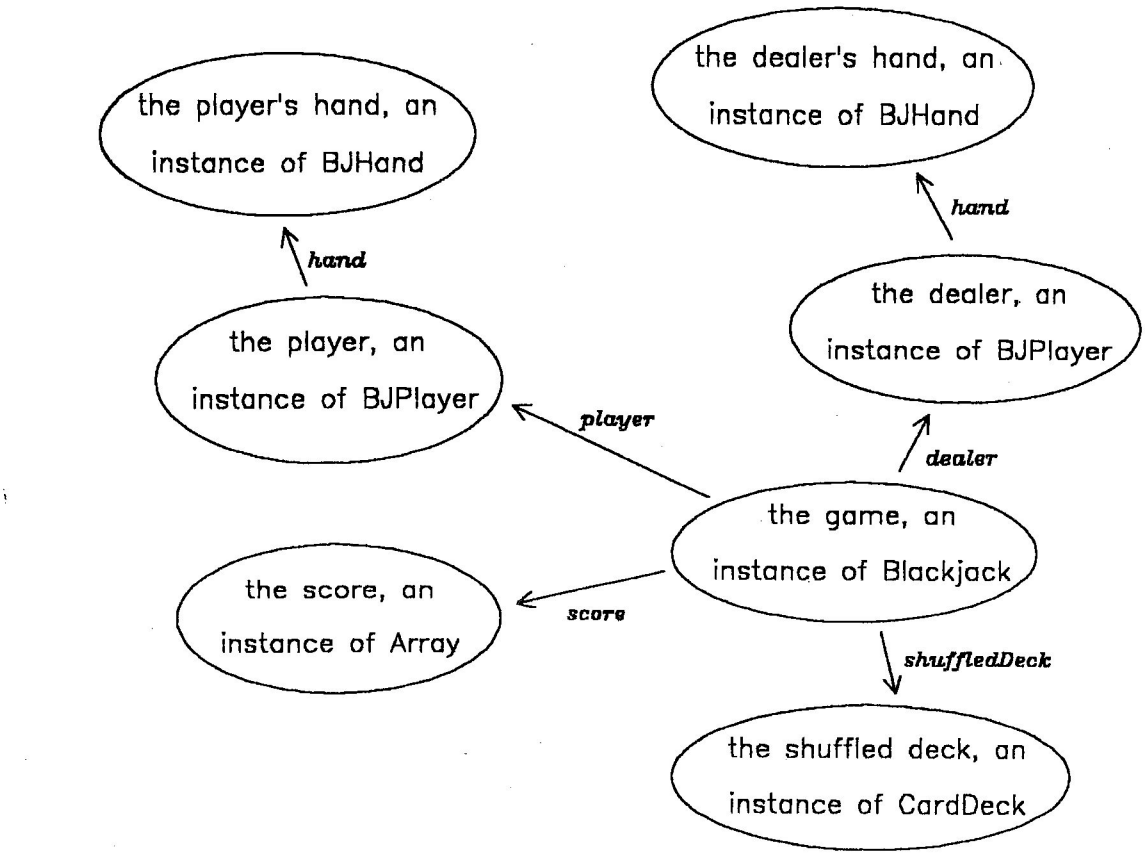

Figure 3. Objects participating in the BlackJack game. As in Topic 1, each node represents an instance of a class. Each link is now labeled with the instance variable used to refer to the other object.

◊ In the Bittitalk Browser, look again at the first expression in Blackjack's `hitPlayer` method.

Within method code, instance variables can be referred to directly: the Blackjack instance doesn't need to send itself a message to retrieve the value of its `playerInPlay` variable, it can just refer to it by name. Otherwise, the first line of this expression is the same as the one you evaluated in the workspace.

◊ In the workspace, type and show:
`Game1 noHandsDealt.`

A prompter with the message "Sorry...Need to deal hands first" is appears. The method then returns the object receiving the message, a Blackjack instance.

In the `hitPlayer` method, the Blackjack instance sends itself (referred to as self in the code) the `noHandsDealt` message if the `playerInPlay` variable has not yet been set.

◊ Look at the second line of the first expression in Blackjack's `hitPlayer` method.

The `ifTrue:` message is sent to the object returned by `playerInPlay isNil.` The `ifTrue:` message is a keyword message. — the message name consists of one or more keywords, each followed by a colon and an argument.

The square brackets ([ ]) indicate that the argument to `ifTrue:` is a *block*. When the `ifTrue:` message is received by `true,` the message expressions inside its block argument are evaluated; if the message is received instead by `false,` no evaluation of the block occurs.

The caret at the beginning of `^self noHandsDealt` identifies it as a *return expression*. When a return expression is encountered, the expression is evaluated and the method is exited, returning the object resulting from the return expression. (You have seen what some of the Blackjack messages return by using "Show it" in the workspace.)

A method can have any number of possible return points; you can see that `hitPlayer` has five. If no explicit return expression is specified, a method returns the value of `self` (i.e., the receiver of the message) when it finishes.

◊ Examine the second expression in Blackjack's `hitPlayer` method.

If the player is not currently in play, the method returns an appropriate prompt for the user.

◊ In the Workspace, type and show:

```
Game1 player takeCard: (Game1 shuffledDeck
removeFirst faceDown: false)
```

A new face-up card is added to the player's hand.

Parentheses are used to specify order of evaluation. The parenthesized expression removes the First card from the deck and turns it face up. This card then serves as the argument to the `takeCard:` message sent to the BJPlayer instance. Can you parse this expression?

◊ In the Browser, find and select the `takeCard:` message defined for the BJPlayer class.

The BJPlayer instance adds the card it is given to its hand, an instance of BJHand.

◊ Return to the `hitPlayer` method in the Blackjack class, and examine the remaining expressions.

After the Blackjack instance passes the card to the player, it asks the player if it now has a blackjack or a bust, and if so reports the outcome to the user. If no bust or blackjack occurs, the game simply returns an instance of Array initialized to hold the two hands, so that the user can see the result of taking a hit. Can you parse these expressions?

Now you've seen how the Blackjack instance and the BJPlayer instance work together to give the player a hit. The Blackjack instance performs an administrative function, checking to see that it is OK for the hit to be

given. If so, it passes a card to the BJPlayer instance, then checks with the player to see if a bust or blackjack has occurred.

Changing the meaning of the stay message

When a Blackjack instance receives the `stay` message, it first checks to see if the player is actually in play. If so, it asks the dealer to turn over its first card and then puts the dealer in play. At that point, the user can send the `playItOut` message, to play out the dealer's hand.

Notice that once the player decides to stay, there is no reason to make a separate request to play out the dealer's hand - that is the only option left. You might take advantage of this fact, making the `stay` message finish the round automatically. You can do this with a simple change to the `stay` method.

◊ In the Bittitalk Browser, select the Blackjack class and its `stay` method.

> Just as in the `hitPlayer` method, the Blackjack instance first checks to see that the player is currently in play. If it is, it asks the dealer to turn over its first card, and updates its `playerInPlay` variable. Can you parse these expressions?

◊ In the Browser, after the `playerInPlay := dealer.` expression, replace the existing return expression with a request for the Blackjack instance (i.e., `self`) to play out the dealer's hand.

◊ While in the pane containing the method text, click the right button to bring up the text-editing menu, and select "Save"

> Your new version of the stay method has replaced the previous version.

◊ Test your new method in the workspace, by sending `deal` and then `stay` to your Blackjack instance `Game1.`

> Instead of simply taking the player out of play and returning the current hands, the enhanced method also plays out the round.

Hint

If you don't get a prompt saying who won, make sure that your new return expression begins with a carat (`^`).

◊ You can explore other methods now if you want. You can see what methods return by trying them out in the workspace; just be sure to send the message to an appropriate receiver (e.g., you can retrieve an instance of BJPlayer by sending the `player` or `dealer` message to `Game1;` you can access a card deck by sending the message `shuffledDeck`).

◊ When you've finished exploring the classes and methods used in the blackjack game, you can close the workspace and browser.

Topic 4: Enhancing the Blackjack User Interface

In Topic 3, you changed the meaning of the `stay` message. Here you will explore the implications that this change has for the user interface, and then make a corresponding enhancement to the user interface, using the View Matcher to do the exploration and modification. You will then get a chance to make further changes to the game and to its user interface.

Playing the enhanced Blackjack game

◊ Start up Topic 4.

A View Matcher opens. As before the Bittitalk Browser is in the lower right, but now you will be able to see the full method text. There are also two new panes in-between the stack pane and the commentary pane; these will be described as you proceed.

◊ Start up the blackjack game.

A Blackjack window opens, and the Browser updates to show the relevant classes.

◊ Deal a hand and take as many hits as you want.

◊ Select "stay" from the player menu.

This selection now completes the round, reflecting the change you made in Topic 3. You can play as many rounds as you like.

◊ Open the dealer menu.

Even though you no longer need to make the "playItOut" request, it is still listed on that menu, and selecting it will result in a message sent to the Blackjack instance. But because of the `playerInPlay` check, you'll never be in the right state to make that request — try it and see!

Re-visiting the Blackjack menus

Before modifying the dealer menu, it may be useful to refresh your memory about how Smalltalk menus work. Now that the full method text is available in the Bittitalk Browser, you can use the View Matcher to analyze this component of the user interface in more detail.

◊ Change to **Halted Stack Mode** and deal a new hand.

The message execution stack displays the messages that led up to the "deal" menu request being sent. It reflects the cooperation among the BlackjackView, the MenuWindow, and the Blackjack instance in responding to the mouse click.

◊ Select the top line of the stack `Blackjack»deal`

As in Topic 2, the View Matcher commentary provides information about the role of `deal` in the application. The Bittitalk Browser updates to show you where the method is located in the class hierarchy; it also displays the method code.

The first (highlighted) expression in the `deal` method is
`self viewMatcherHalt.`
This is the expression that produces the View Matcher halts.

The two panes underneath the message execution stack are *inspector* panes: they allow you to examine the receiver of the selected message (referred to as `self`), as well as any arguments or temporary variables relevant to the processing of the message. If you select an item from the left pane, you can see the current value of the selected item in the right pane.

◊ In the message execution stack, select the line
`BlackjackView(Object)»perform:`

This is the point at which the object managing the user interface receives a menu request from the menu bar (an instance of MenuWindow).

This time, the list in the left-most inspector pane includes `aSymbol.` This is the name of the argument sent with the `perform:` message (see the method code in the browser). You can see what object was sent by selecting the argument name in the inspector list pane. If the code for a method uses any temporary variables (declared inside vertical bars at the top of the method), these would be listed as well.

◊ Now you can analyze in detail how the blackjack user interface responds to a menu selection by a user. Explore the stacked messages, using the View Matcher commentary, the list of objects in the inspector panes, and the method text to see how the user interface objects process and pass on the request to the Blackjack instance.

The Blackjack and BlackjackView classes were written especially for the blackjack application. Many of the other classes involved (e.g., NotificationManager, TopPane, Message, MenuWindow) are part of the base Smalltalk system, and needed no enhancement.

◊ Resume and continue to play and explore the game.

As in Topic 2, the game halts when menu requests are made and when panes are about to be updated. This time though you have the method code and can more closely follow the message passing involved in these user interface transactions. Remember that you can see the value of method arguments and temporary variables by selecting them in the left inspector pane.

Modifying the dealer menu

The dealer menu is out of date: it offers the "playItOut" option, but this is an option that doesn't make sense any more. In order to modify this menu, you must change its definition. This definition gets processed in the course of opening the blackjack window.

◊ Close the blackjack window, using the "End Application" choice in the View Matcher's application menu.

◊ Make sure that the View Matcher is in **Halted Stack Mode** and re-start the blackjack game.

The game stops in the process of opening itself.

◊ Select the stacked message, `BlackjackView»openOn:`

The method is organized into parts. First the BlackjackView sets some information about itself- its label, its game instance variable. 1t is also in this code that the special *dependents* relationship is established, with the `dependsOn:` message. The BlackjackView instance also sets up some event-handling to build its menu bar, and to close itself properly.

The messages in this first block of code are combined into a single long expression through the use of *cascaded* messages. Cascading is indicated by a semi-colon (;), and causes the next message to be sent to the same receiver as the previous message. This removes the need to specify the same receiver (in this case `self`, the instance of BlackjackView) over and over.

The second two blocks of code create the two subpanes. Can you tell which is which?

A subpane's behavior is determined by defining how it will respond to various *events* - these events can be both actions initiated by the user and internal events initiated by the system. Events are defined for a subpane via the `when: perform:` message. The effect of this message can be paraphrased as "when this subpane gets this event, it will ask its owner to perform this message." As you can see by looking at the code, the `owner` of the subpanes is the BlackjackView instance. The message sent as the result of such a definition will always have one argument, the subpane processing the event.

At the minimum, the `#getContents` event must be defined, because this is how the subpane's display contents are provided. What will happen when the dealer pane receives a `#getMenu` event?

◊ In the Browser, find the `dealerMenu:` method in the BlackjackView class.

This method returns an instance of the Menu class.

A menu has two main components: its "labels" (the words you see when the menu is displayed) and its "selectors" (the messages that are sent to the owner of the menu when a label is selected). Can you find these in the dealer menu method?

◊ Delete the "playItOut" references in the labels and in the selectors.

The \'s in the labels string cause carriage returns to be inserted between the words when the string receives the `withCrs` message.

◊ With the mouse in the pane containing the method text, click the right button to bring up the text-editing menu, and select "Save"

Your revised menu creation method is saved.

◊ Resume the game as many times as needed to finish opening the window.

The View Matcher title bar updates to "Running" to indicate that the application is ready for further input.

◊ Test your change by opening the "Dealer" menu.

Feel like cheating?

The View Matcher makes it easy for you to "get inside" the blackjack game. If you want, you can manipulate the objects that are participating in a message. You can make the player's hits always produce a blackjack!

◊ Make sure the View Matcher is in **Halted Stack Mode**, then deal a new hand and resume enough times to display the two new hands.

◊ Request a hit for the player.

As in Topic 2, the game halts in the midst of giving the player a card.

◊ Select the `BJPlayer»takeCard:` message at the top of the stack, and then select the argument to the message in the inspector pane below.

Are you happy with the card you were given? If not, you can change it!

◊ Open an inspector on `aCard`, by clicking the right button and selecting "inspect" from the single-item inspect menu.

The inspector is a special tool that gives you access to the current settings of an object's *instance variables*. Card instances have three such variables, and you can view their current settings by selecting them.

Notice that different objects have different ways of displaying

themselves: objects that are kinds of collections, for example, usually display their contents. Many objects indicate simply that they are an instance of their class (e.g., "a Blackjack"), and you must use an inspector to examine their internal state.]

◊ Select `value` in the instance variable list.

Its current setting appears in the right-hand pane. This pane is simply a TextPane, and you can use its editing capabilities to edit this setting.

◊ Highlight the current value in the right-hand pane, then type over it the card value you would rather have.

> **Hint**
>
> If you want a face card, you must precede it with a pound sign . This identifies it as an instance of the Symbol class. Even though the value of a card is treated like an integer (e.g., it gets summed), it is stored as a symbol, to allow for face cards (i.e., #A is an ace, #J is a jack, etc.).

◊ Click the right mouse button to pop up the text editing menu, and select "Save."

The new setting for `value` has been stored.

◊ Close the inspector and resume the game as many times as needed to see the player's new card - did it work?

The inspector lets you change the values of specific objects in the midst of an application's activities. If you want to make a permanent change, you must modify the method code used by the application. Think about what you might do to make your BJPlayer instance a permanent "card-fixer."

You could also make your BJPlayer a liar: it could report a blackjack to the dealer whenever it got a card, regardless of the card's value. How would you do this?

A more challenging enhancement

Modifying the dealer menu involved changes to a single method — specifically the method that creates the relevant menu object. Most changes to an application's user interface will not be so simple. Suppose you wanted to redefine the application to have *three* subpanes, one for each of the hands, and one to display the current score. The hints below will help you try this.

◊ Put the View Matcher in "Normal Execution" mode, so that it doesn't stop so much while you are working on these changes.

◊ Find the `openOn:` method in the BlackjackView class, and modify it so that it adds a third TextPane as a subpane.

Hints

Use the dealer subpane definition in this method as an example in building your score pane definition. You won't need to define a `#getMenu event`, but you should define a `#getContents` event that will send the `#scorePane:` message.

You'll need to define a `framingRatio:` message for the new subpane. The argument to this message is a rectangle specifying the relative position of the pane. The rectangle is defined as an extent beginning at a position relative to the four corners of the containing window, the `leftTopUnit`, `leftBottomUnit`, `rightTopUnit`, or `rightBottomUnit.` The extent itself is specified as a Point instance (i.e., x @ y). So, for example, in the original window, the player pane begins at the left top corner and its extent is 1/2 the width and the entire height of the window; in contrast, the dealer pane begins 1/2 way across (to the right) of the window's top left corner, but its extent is the same.

Notice that you will need to change the ratios for the player and dealer panes too. As a starting effort, try creating three equally sized vertical subpanes. Use parentheses liberally!

◊ Edit the `scorePane:` method in the Blackjack class, so that it will return the appropriate contents for the score pane.

Hints

Look at one of the other subpane contents methods in BlackjackView (e.g., `playerPane:`) to see how this is done for the other TextPanes.

All objects respond to the message `printString`, returning a printable "string view" of themselves. Remember that score belongs to the Blackjack instance, and that the BlackjackView instance points to the Blackjack instance via its game variable.

◊ The final step is to find the `incrementScore:` method in the Blackjack class and to insert a request for the score pane to be updated.

Hints

Look in methods that update the dealer and player subpanes (e.g., `deal`) for examples of how to do this.

◊ Test your changes by closing and re-opening the blackjack application.

◊ When you've finished working with the View Matchcr, you can close it.

Topic 5: Exploring Bibliography and Gomoku

In Topics 1-4, all of your work has centered on a single application, the blackjack game. In this topic, you'll have a chance to explore two other example applications, a bibliography and a gomoku game. These applications use other parts of the Smalltalk/V class hierarchy: the bibliography uses a ListPane in its user interface, and the gomoku game uses a subclass of GraphPane. Through your investigations of these examples, you will learn some additional techniques for building interactive applications. Each example has been incorporated into the View Matcher to help you do this. After exploring these applications in the View Matcher, you will use the standard environment to try out same enhancements to the applications.

◊ Start up Topic 5.

A View Matcher opens. Everything is as in the other topics. However, when you open the "Applications" menu, you'll find that in addition to "Blackjack," you can choose either of the two new applications.

◊ Explore the gomoku game.

Play the game first in normal execution mode. The goal of this game is to try to get 5 squares in a sequence, either a row, column or diagonal. See if you can figure out the kinds of strategies the game uses to counter your moves.

Put the View Matcher into **Halted Stack Mode** and use the message execution stack and commentary to see if you can follow the processing of your click on a square. How does your piece get drawn? Where is the counter move calculated?

◊ Explore the bibliography.

This application has been initialized to hold several references, using the categories "OOP Strategies" and "OOA/OOD." Try adding new references, both within the existing categories, and by first adding a new category. Then try deleting references or categories. Use the View Matcher's **Halted Stack Mode** to see how the bibliography application responds to requests from the user.

◊ When you have finished exploring gomoku and the bibliography, close the View Matcher.

Working with the system tools

You have been using two special tools — the Bittitalk Browser and the View Matcher — to explore the example applications. These tools were built from the standard system tools, and in your normal work with Smalltalk, you will carry out similar kinds of analysis and programming tasks with the standard tools.

◊ Select the "Browse Classes" option from the "File" menu of the Transcript window.

A Class Hierarchy Browser opens. This browser is just like the Bittitalk Browser you have been using, but it includes *all* the classes and methods in the system.

◊ See if you can find the classes you have been working with in the bibliography and gomoku applications.

> **Hints**
>
> The "Find class" function on the "Classes" menu can be used to find a class if you already know the name. When browsing for a class, ellipses (...) following a class name mean that it has a subhierarchy beneath it; you can expand the hierarchy by double-clicking on the class name.

◊ Find the `drawAt:` method in the CircularGamePiece class and change the first line of code from `self viewMatcherHalt` to `self halt.`

The `self halt` expression is used to interrupt message processing. You can then use the Debugger to explore the application.

◊ Open a new gomoku game by evaluating (in the Transcript window or in a Workspace) the expression:

```
GomokuView new openOn: (Gomoku newGame)
```

The parenthetic expression creates a new initialized instance of the Gomoku class, just as you did to start up the blackjack game in

Topics 1 and 3. This instance is used as the model by the GomokuView instance. Why must the new message be sent before the `openOn:` message?

◊ Select a square on the gomoku board.

A red "Walkback" window opens. The system uses walkbacks to indicate that a halt has been encountered. It also uses them to notify you about error conditions, for example if a message is not understood by an object, or if an illegal manipulation of an object is attempted. The list of messages in the walkback corresponds to the execution stack as of when the halt or error was encountered.

◊ Select the "debug" button in the walkback window.

A yellow debugger window opens. The debugger is similar to the View Matcher tool — using the stack, you can trace what is happening in the code at this point, and you can inspect the object receiving a message, as well as arguments or temporary variables defined within a method. You can open an inspector just as you did in Topic 4, by selecting a variable and then selecting "inspect" from the single-line inspect menu. Do you recognize any of the messages in the stack?

◊ Try using `self halt.` expressions to recreate other message stacks you explored for the gomoku game or the bibliography Try using the technique to explore other parts of the application.

Hints

Remember to remove the `self halt.` expression when you have finished exploring a piece of one of the example applications. Close each debugger window as you finish with it.

Enhancing the bibliography and Gomoku applications

You have seen how to use the standard system tools to analyze an example application. Now you can use them to make some simple enhancements to the gomoku and recipe applications.

◊ Try enhancing the gomoku game: Look at how the board and its pieces are drawn. Can you change the color of the board? Now about the color or shape of the game pieces?

> **Hints**
>
> The game pieces are created as needed by sending a *class* massage to CircularGamePiece. This message returns a piece that knows how to draw a circle and fill it with the appropriate color. If you want to change just the color, you can do this by adding a new class message. But if you want a new shape, you'll need to define a new subclass of BoardGamePiece, with its own draw methods, etc.

◊ Try building a tictactoe game: Using gomoku as a model, see if you can build a similar board game that plays tictactoe. You should be able to inherit and/or reuse much of the functionality that you will need for this.

> **Hints**
>
> You will need to create both a TicTacToe *class*, and a TicTacToeView class. You could create these classes as either subclasses or "siblings" of the corresponding Gomoku classes. lf you want to inherit functionality from Gomoku, make them subclasses. But if you need to modify a lot of the Gomoku behaviors, a better strategy will be to make the TicTacToe class a sibling of Gomoku, and then copy over and modify any useful methods.

◊ Try enhancing to the bibliography application: The application now requires that you select a category via a Prompter. This is not the best interface, because the current category remains hidden. A possible improvement is to add an additional ListPane, having it list reference categories (highlighting the currently selected one). This would make the bibliography more like the Class Hierarchy Browser.

Hints

Open a bibliography by evaluating the expression:
`BibliographyView new openOn: (Bibliography newForMiTTS).`

Remember that you not only need to create the new subpane (in the `openOn:` method of BibliographyView), but also that you need to set up methods for providing its contents and seeing that it gets updated correctly. You can use halts and the debugger to see how this is done for the ListPane currently used to display reference items.

Appendix A. Smalltalk/V Interaction Tips

Entering text	Point and click to move the text cursor to the desired insertion point and. type. The mouse pointer must be within the active window during text entry.
Selecting text	Point to the beginning of the text to be selected, then press and hold to highlight the desired text.
Deleting text	Press the backspace key, or use the “cut” option on the text menu to remove any selected (highlighted) text.
Copying text	Select the text to be copied, then choose “Copy” from the text-editing menu. Position the text cursor to the copy location, then choose “Paste” from the text-editing menu.
Moving text	Select the text to be moved, then choose “Cut” from the text-editing menu. Position the text cursor to the move location, then Choose “Paste” from the text-editing menu.
Scrolling	Click the up and down arrows of the scroll bars to move one line at a time; click inbetween the thumb and the arrows to move up or down one page at a time; press and drag the thumb to move anywhere in the text.
Moving a window	Point to the title bar, press and hold the left mouse button while moving in the desired direction.
Re-sizing a window	Position the cursor over a window border (it will change into a two-headed arrow), then press and hold the left mouse button, drag‘ ging the border to create the desired size.
Surfacing a window	If part of the window is visible, click on it. Otherwise, use the F9 key to cycle through all windows until you get to it.

Appendix B. Smalltalk/V Glossary

block - one or more message expressions enclosed in square brackets ([ ]), often employed in cases where conditional execution is needed.

binary message - one of a predefined set of messages that require a single argument but no colon. E.g., `anInteger + 1.`

cascading - a syntactic shortcut used for sending successive messages to the same receiver; the messages are separated by a semi-colon (;). E.g., `aCollection add: 'test1'; add: 'test2'`

class - a key abstraction in Smalltalk. A class is a special-purpose object that defines the structure and behavior shared by all of its instances. The main function of a class is to create instances of itself.

class message - a request for behavior sent to a class rather than an instance of a class, e.g. the `newGame` message sent to the Blackjack class.

class variable - a variable managed by a class object, but whose value is accessible by (i.c., global to) all instances of the class. Like instance variables, class variables are inherited by subclasses (e.g., because `Dependents` is a class variable of the class Object, an instance of any class can access it).

dependent - an object whose state depends on the state of another object. A dependent (c.g. a view manager) is sent an "update" request when the object it depends on (e.g., the application model object) notifies itself of a change.

event - an internal response to some action, often a user interface action (e.g., the `#button1Up:` event occurs when the user presses the left mouse button). A subpane is normally set up to recognize certain events, and to respond to an event by sending a specific message to its owner.

inheritance - a mechanism whereby subclasses can use instance variables and methods defined in their superclass(Es).

inspector - a Smalltalk tool allowing the user to examine and manipulate an instance's internal state.

instance - an object instantiated according to the definition of a class.

instance message - a request for behavior sent to instances of a class, e.g., the `deal` message sent to a Blackjack instance.

instance variable - a variable defined for any instance of a class. The value of the variable is private, and is managed by the instance itself.

keyword message - a request for an operation consisting of one or more words, each followed by a colon and argument. E.g., `aCollection at: 1 put: anObject.`

message - a request for an operation to be carried out.

message execution stack - an ordered list of message-sends, containing messages whose evaluation has begun but not completed.

method - the code executed in response to a message.

model - the object managing an application's underlying functionality. Normally this object communicates with the view manager object, so that the application can respond to user requests and the user interface can be updated as needed.

object - an instance of any class.

pane - a component of an application's display, generally used for presenting information made available by its model, and for event handling.

owner - a special instance variable used by components of the user interface to refer to their managing object, usually the view manager (e.g., an instance of BlackjackView is the owner of the window and subpanes in the blackjack game).

receiver - the object to whom a message has been sent.

return expression - a message expression prefaced by a caret (^), which when evaluated in a method will cause the method to exit immediately and return the value of the expression.

subpane - a component of most user interfaces that is contained by a window, and normally used to display application information and to process application-relevant events.