4.1 Introduction

In object-oriented systems it often happens that some object sends a common message to a number of other objects. For example, a bank may send the message ‘calculate charges’ to each of its accounts, or a student may send a common email to each of the other students in their class. In both of these examples, a message is distributed to each object in some given collection. In this chapter we look at how distributed message passing can be captured within an Object-Z specification.

A common pattern associated with distributed message passing involves the introduction of a mediator object. A message for distribution is first passed to the mediator object. This object has links to all relevant objects in the system and hence is able to distribute the message to the appropriate receiver objects, i.e. the mediator object acts like a message-clearing house. The existence of a mediator considerably reduces the inter-object links required in the system, as other objects need only communicate with the mediator. Furthermore, the mediator is able to maintain the inter-object links efficiently as the system evolves. This chapter discusses the formal specification of the architecture and functionality of a system with a mediator object, and demonstrates the informal refinement of this architecture to Java code.

4.2 Bingo: a case study

The game of bingo involves a collection of players each allocated a set of numbers. An announcer calls out numbers in some arbitrary order and the first player to have all their numbers called is the winner.

The main issue of interest in this case study is how to specify formally the distributed message passing that occurs when the announcer calls a number. From an object-oriented point of view, the number-call message from the announcer object is distributed to all the player objects.
Informal description

- There is a set of players each of whom is initially allocated a set of 16 numbers in the range 1..99. The sets allocated to any two players may have numbers in common, but must not be identical.

- There is an announcer who from time to time calls individual numbers in the range 1..99 in arbitrary order, with no number being called more than once.

- When all of the numbers allocated to a player have been called, that player can shout ‘bingo’. The first player to do so is the winner.

A player

A player object is specified by the class Player:

```
Player

Init
remain : P(1..99) #remain = 16

hearCall
(remain)
n? : 1..99
remain' = remain \ {n?}

shoutBingo
shout!: Player
remain = Ø
shout! = self
```

The attribute remain denotes the set of numbers that remain to be called, i.e. those numbers that were allocated to the player but have yet to be called by the announcer. Initially, remain contains 16 numbers.

The operation hearCall corresponds to the player hearing the latest number called (n?) and removing it from remain if it had been allocated to the player. Notice that if the number called is not in remain, this set is unaltered.

The operation shoutBingo can only occur when all the numbers allocated to the player have been called. In an actual game, the player shouts ‘bingo’, but as this is done only so that the winning player can be identified, in the model here the player is identified directly by having the output (shout!) take the value self.

The announcer

The announcer is modelled by the class Announcer in Figure 4.1. The boolean attribute active denotes whether or not the game is in progress. Initially active is true, and only becomes false when one of the players completes the game. This attribute is associated with the announcer to model the fact that this person takes responsibility for controlling the game, albeit with the cooperation of the players.
The attribute `remain` in this class denotes those numbers in the set `1..99` that have not yet been called. Initially `remain` is the entire set `1..99`.

The operation `callOut` corresponds to the announcer calling any remaining number. This can occur only if the game is active and not all numbers have been called. Notice that the order in which the numbers are called is not specified.

The operation `stopCalling` corresponds to the announcer receiving as input the identity of a player (`shout?`) who has completed the game. The game is active before the operation, but becomes inactive afterwards.

### The bingo system

The game of bingo involves a set of player objects and an announcer object, denoted by the attributes `players` and `announcer` respectively. It is specified by the class `Bingo` in Figure 4.2.

Initially, each player and the announcer are in their initial states, i.e. each player has 16 numbers and the announcer is active but has not yet called any numbers. Furthermore, distinct players initially have distinct sets of numbers.

The operation `numberCalled` models distributed message passing and is illustrated in Figure 4.3: the number called by the announcer is passed to all of the players. It involves the operation `announcer.callOut`, whereby the announcer calls out a number, in parallel with the distributed conjunction operation

\[ \land p : \text{players} \land p.\text{hearCall}, \]

whereby each player hears the number called. This distributed conjunction is equivalent to

\[ p_1.\text{hearCall} \land p_2.\text{hearCall} \land \cdots \land p_k.\text{hearCall} \]
The class `Bingo`:

```
players : P Player
announcer : Announcer

INIT
∀ p : players • p.INIT
announcer.INIT
∀ p₁, p₂ : players | p₁ ≠ p₂ •
  p₁.remain ≠ p₂.remain

numberCalled ≜ announcer.callOut
  \parallel
  (\land p : players • p.hearCall)

gameOver ≜ announcer.stopCalling
  \parallel
  (\Box p : players • p.shoutBingo)
```

Figure 4.2: the class `Bingo`

---

**Figure 4.3:** the `numberCalled` operation of `Bingo`
where \( p_1, p_2, \ldots, p_k \) are the players. As the input \( n \) to each \text{hearCall} operation in this conjunction is denoted by the same identifier, these inputs are all equated; hence the result of the distributed conjunction is that each player hears the same number. Furthermore, because the output \( n! \) from \text{announcer.callOut} has the same base name \((n)\) as the common input to the players, \( n? \) is equated to \( n! \) and both are hidden, with the overall result that the number called by the announcer is distributed to and heard by each player.

The operation \text{gameOver} is specified using the parallel and distributed choice operators and can only occur if some player has completed the game. If the operation does occur, the player to shout is selected non-deterministically from among those who have completed the game. This models the situation where several players may have completed the game but it is the first one to shout 'bingo' who is the winner. The output \text{shout!} of \text{p.shoutBingo} is equated to the input \text{shout?} of \text{announcer.stopCalling} and both are hidden, i.e. the announcer identifies the player concerned and stops the game.

4.3 A buttons-toggling puzzle: a case study

A simple puzzle that can be played on a computer consists of 16 square toggle buttons arranged on a board in a 4 by 4 array. A toggle button may be in one of two states: \textit{hollow} or \textit{solid}. Clicking (with the mouse) on any button toggles it and each of its horizontal and vertical neighbours between hollow and solid. (Buttons in the centre of the board have 4 neighbours; those at an edge have 3; those at a corner have just 2.) Initially all the buttons are hollow and the aim of the puzzle is to reach a state where all the buttons are solid. Figure 4.4 illustrates a typical intermediate state that could be reached while attempting to solve the puzzle. Hollow toggle buttons are shown in white, solid toggle buttons in black.

![Figure 4.4: the buttons-toggling puzzle](image)

The issue of distributed message passing arises in this case study when the mouse is clicked on a button: the message ‘toggle’ is sent not just to the button directly identified by the mouse, but also to each neighbouring button.
The specification presented here uses a mediator to facilitate this message passing. Section 4.4 presents a refinement of the specification to Java code.

**Initial abstractions**

Before modelling the board of toggle buttons and the way they interact, it is useful to establish a data structure of board positions so that the buttons can be easily referenced. As the board consists of 16 positions, one possible labelling of these positions is to use natural numbers as in Figure 4.5.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 4.5: the board positions

Hence define $\text{Posn} = 0..15$.

The 'neighbour' relationship between grid positions can now be expressed in terms of the corresponding coordinates.

\[
\forall i, j : \text{Posn} \quad i \text{ neigh } j \iff \begin{cases} 
  i - j \in \{-4, 4\} \\
  (i - j \in \{-1, 1\} \land i \text{ div } 4 = j \text{ div } 4)
\end{cases}
\]

(Here $\text{div} : \mathbb{N} \times \mathbb{N}_1 \rightarrow \mathbb{N}$ denotes integer division.) The relation $\text{neigh}$ captures the idea that two board positions are neighbours when they are adjacent either horizontally or vertically, i.e. the corresponding squares have an edge in common.

Define the global type $\text{Style}$ to model the two possible states of a toggle button.

\[
\text{Style ::= } \text{hollow} \mid \text{solid}
\]

A global type is available to each class in a specification (in fact, it is only used in the class $\text{ToggleButton}$ in this simple specification).
4.3 A buttons-toggling puzzle: a case study

**A toggle button**

A toggle button is modelled by the class:

```
ToggleButton

<table>
<thead>
<tr>
<th>style : Style</th>
<th>INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>style = hollow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>toggle</th>
<th>isSolid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δ(style)</td>
<td>style = solid</td>
</tr>
<tr>
<td>style' ≠ style</td>
<td></td>
</tr>
</tbody>
</table>
```

A toggle-button object has an attribute `style` of type `Style`: this will be used to indicate whether it is hollow or solid. Initially the button is hollow. If the operation `toggle` is performed the style of the button alternates between hollow and solid.

Notice that there is no attribute to record the position of a toggle button: it is not the concern of the button to know where on the board it may be placed, or which buttons are its neighbours. Also, at this level of abstraction the shape of a toggle button is immaterial: the fact that in an implementation a button is actually square is not relevant to the functionality and architecture being specified.

**The buttons puzzle**

The buttons puzzle is specified by the class `ButtonsPuzzle` in Figure 4.6. The attribute `board` is a function that associates a toggle button with each position in the set `Posn`. The arrow ‘↣’ indicates that `board` is a one-one total function, i.e. a unique toggle button is associated with each of the 16 positions. The secondary variable `buttons` is this set of 16 toggle buttons. The secondary variable `posn` is the inverse of the function `board` (the symbol ‘∼’ denotes relational inverse). That is, `posn` is a one-one partial function that associates with each toggle button its position on the board.

The operation `toggleButtonAndNeighbours` takes as its input one of the board’s buttons (`b?`) and toggles both this button (`b?.toggle`) and, using the distributed conjunction operator, each neighbouring button:

```
∧ b : buttons | posn(b) neigh posn(b?) • b.toggle
```

Notice how the function `posn` and the relation `neigh` are used to select the neighbours of `b`.

The operation `puzzleSolved` is applicable if all the buttons on the board are solid (i.e. if the puzzle is solved). However, even if it does occur, the state is unchanged and afterwards the operation `toggleButtonAndNeighbours` is still applicable and button toggling can continue. This is consistent with a physical array of buttons with no ‘lock-on-all-solid’ mechanism. Compare this with the
\begin{align*}
\text{ButtonsPuzzle} \\
\begin{align*}
\text{board} &: \text{Posn} \rightarrow \text{ToggleButton} \\
\Delta \\
\text{buttons} &: \mathbb{P} \text{ToggleButton} \\
\text{posn} &: \text{ToggleButton} \rightarrow \text{Posn} \\
\text{buttons} &= \text{ran board} \\
\text{posn} &= \text{board} \\
\end{align*}
\end{align*}

toggleButtonAndNeighbours \equiv 
\begin{align*}
[ b? : \text{buttons} ] \bullet \\
\ & \quad b?.\text{toggle} \\
\ & \quad \land \\
\ & \quad ( \land b : \text{buttons} \mid \text{posn}(b) \sim \text{neigh posn}(b?) \bullet b.\text{toggle} )
\end{align*}

\text{puzzleSolved} \equiv \land b : \text{buttons} \bullet b.\text{isSolid}

\text{Figure 4.6: the class ButtonsPuzzle}

gameOver \ operation \ for \ the \ bingo \ game \ in \ Section \ 4.2: \ once \ this \ operation \ is \ performed \ the \ game \ stops.

\textbf{Discussion}

In this specification, a buttons-puzzle object acts both as a system object whose attribute \textit{board} identifies the toggle-button objects in the system, and as a mediator for passing messages between the buttons. When a button is selected for toggling, the buttons-puzzle object ensures that the neighbours of the selected button are also sent the message \textit{toggle}. In this model a button has no direct reference to any of the other buttons.

It would have been possible to have specified a \textit{ToggleButton} class having attributes to record an associated grid position and to reference buttons in neighbouring positions. However, not only would this architecture result in a multitude of inter-object references, the button would have become highly specialised to this particular application making it less reusable in other contexts.

Notice that it is possible to solve this puzzle: starting from the initial state, click on the buttons in positions 1, 7, 8 and 14.
4.4 Refinement to code

This section illustrates informally the refinement of the Object-Z specification of the buttons-toggling puzzle to code implemented in the Java object-oriented programming language. In particular, we shall see how the pattern of mediated message passing underlying the specification is mirrored in code.

Java is used as the implementation language as its established library of classes means that the translation of predicates into code can be done without recourse to low-level detail. The refinement closely follows the functionality, architecture and class structure of the specification. However, a knowledge both of the basic syntax of Java and its AWT library of graphical classes is needed to fully appreciate the details of the code. (See for example, the book by Horstmann and Cornell listed in Appendix D.) The processes of the refinement to Java given below could be emulated in other object-oriented programming languages.

Code for the toggle-button class

The ToggleButton class specified in Section 4.3 can be refined to the Java class ToggleButton whose code is given in Figure 4.7. This class inherits the library class Canvas.

Each aspect of the specification of the Object-Z class ToggleButton has a direct counterpart in the Java class ToggleButton.

For example, the boolean instance variable hollow mirrors the attribute style in the specification and plays the same role: its value, true or false, determines respectively whether the button is hollow or solid. The method toggle implements the operation toggle in the specification: its effect is to toggle the button's style. The method isSolid implements the operation isSolid: it returns true if and only if the button is solid.

However, there are aspects of the code not derived directly from the specification. For example, unlike the specification, the implementation needs to be explicit about the shape of a button, and this is captured by the method paint which sets a button to be a 48 pixels by 48 pixels rectangle (i.e. a square).

Also, the instance variable mediator was not required in the specification. Its role in the code is to reference the mediator object that has responsibility for distributing messages to other buttons (see below for further discussion on this issue). This reference is needed by the method mousePressed which, when the mouse is pressed on the button, sends to the mediator object the message toggleButtonAndNeighbours with the button itself as argument. As we shall see in the code for the class ButtonsPuzzle, when the mediator object receives this messages it sends the message toggle to the button that sent the message (i.e. the button self) and to each of the button's neighbours. That is, the button and all of its neighbours are toggled.

The constructor method sets the value of the variable mediator to be the object passed as argument when the button is first created.
import java.awt.*;
import java.awt.event.*;

public class ToggleButton extends Canvas {

    // instance variable
    private boolean hollow = true;
    private ButtonsPuzzle mediator;

    // constructor
    public ToggleButton (ButtonsPuzzle med) {
        mediator = med;
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent evt) {
                mediator.toggleButtonAndNeighbours(ToggleButton.this);
            }
        });
    }

    public void paint (Graphics g) {
        // draw button
        if(hollow) g.drawRect(1, 1, 48, 48);
        else g.fillRect(1, 1, 48, 48);
    }

    public void toggle() {
        // toggle the button
        hollow = !hollow;
        repaint();
    }

    public boolean isSolid() {
        return !hollow;
    }
}

Figure 4.7: code for the ToggleButton class

Code for the buttons-puzzle class

The ButtonsPuzzle class specified in Section 4.3 can be refined to the Java class ButtonsPuzzle whose code is given in Figure 4.8. This class inherits the library class Applet.

In this code the instance variable board mirrors the attribute of the same name in the specification. It is set to be an array of integers of length 16, which is a way of implementing a function such as board in Java. In the method init a unique toggle button is created for each of the 16 possible positions. That is, the code creates the functional relationship between positions and buttons.
import java.applet.*;
import java.awt.*;

public class ButtonsPuzzle extends Applet {

    // instance variables
    private ToggleButton [] board = new ToggleButton[16];

    public void init () {
        for (int i=0; i<16; i++) {
            board[i] = new ToggleButton(this);
            add(board[i]);
        }
    }

    public void paint (Graphics g) {
        for (int i=0; i<16; i++)
            board[i].setBounds(50*(i%4), 50*(i/4), 50, 50);
    }

    public void toggleButtonAndNeighbours (ToggleButton but) {
        but.toggle();
        int butPosn = posn(but);
        for (int i=0; i<16; i++)
            if (neigh(butPosn, i)) board[i].toggle();
        if (puzzleSolved())
            showStatus("Puzzle solved. Congratulations!");
        else showStatus("Puzzle not solved. Keep trying!");
    }

    public int posn (ToggleButton but) {
        int index = 0;
        while (!(board[index] == but)) index++;
        return index;
    }

    public boolean neigh (int i, int j) {
        return
        i-j == 4
        || j-i == 4
        || ((i-j == 1 || j-i == 1) && i/4 == j/4);
    }

    public boolean puzzleSolved () {
        boolean result = true;
        for (int i=0; i<16; i++)
            result = result && board[i].isSolid();
        return result;
    }
}

Figure 4.8: code for the ButtonsPuzzle class
to mirror its declaration in the specification.

Also, as each button is created, its mediator variable is set to be this, i.e. the object of the class ButtonsPuzzle itself. Hence, just as in the specification the mediator was the object referencing all the toggle-button objects in the system, the mediator in the implementation is the object responsible for creating all the toggle-button objects of the system.

The secondary variables buttons and posn used in the specification to select the neighbours of a given button are not declared as variables in the implementation as the selection is performed by direct computation. For example, the relation neigh defined in the specification is implemented by the method neigh, but whereas the specification of neigh simply states a condition that must be true if two coordinate positions are neighbours, the implementation directly computes the boolean equivalent, i.e. it returns true if and only if the two given positions are neighbours.

Similarly, the inverse function posn defined in the specification is implemented by the method posn. This method returns the position of the given toggle button.

The method toggleButtonAndNeighbours: aButton is an implementation of the operation of the same name defined in the specification. First the identified button is sent the message toggle, and then this message is sent to each of the neighbouring buttons. In the implementation, the puzzle is checked to see if it has been solved, and an appropriate message displayed, every time a toggle button is pressed by the mouse. The method puzzleSolved is used to do this checking; this method is a direct implementation of the operation of the same name in the specification.

Discussion

The implementation of the buttons-toggling puzzle closely follows the specification. In particular, the implementation of a mediator object for distributed message passing is architecturally similar to the specification. Furthermore, the predicates in the specification are translated more or less directly into code. The implementation, however, requires that functions and relations be directly calculated, whereas the specification can be declarative.

In the implementation, each button object has a variable mediator. This was not needed in the specification because the button identified for toggling $b?$ was simply input directly to the mediator by the environment. In the implementation, this button is identified by the mouse, and the information has to be passed to the mediator object: to do this each button needs to reference the mediator object.

The INIT schema in each Object-Z class is not an operation; it simply gives the condition (predicate) for an object of that class to be in its initial state. In the code, however, this is implemented by an appropriate constructor method that sets the instance variables to take the required initial values.

The implementation of the button-toggling puzzle presented here can be executed as a Java applet on any Java-enabled web browser.