Chapter 6. Sprites

A game's active entities are often encoded as *sprites*. A sprite is a moving graphical object, which may represent the player (and so respond to key presses and mouse actions) or be driven by 'intelligent' code in the game.

The Sprite class developed in this chapter holds a sprite's position, its speed (coded as incremental steps in the x- and y- directions), and uses the image classes (ImagesLoader, ImagesPlayer) from chapter 4 to manage its graphical presence. Sprite's subclasses add user and environmental interactions, and audio effects. The coding of these classes is greatly helped by specifying them first with UML statecharts.

Many elements are utilized from earlier chapters: the animation framework of chapters 1 and 2, the image loader classes of chapter 4, and the audio loaders of chapter 5.

1. The BugRunner Application

The BugRunner is an ant sprite which can be moved left and right across the base of the gaming pane to stop falling ball sprites from hitting the floor. Figure 1 shows BugRunner in action.

Figure 1. The BugRunner Application.

The ant is controlled either with the arrows keys or by clicking with the mouse. The left arrow key makes the ant move to the left, the right arrow key makes it go right, and the down key stops it. If the mouse is clicked when the cursor is to the left of the ant, it makes the ant walk to the left, when the cursor is to the ant's right, then the ant will go right instead. The ant's legs move as it walks.

Once the ant is set in motion, it continues moving until its direction is changed or it is stopped. When the ant reaches the left or right walls, it continues walking off-screen until it has disappeared, then appears again at the other edge.

A ball is dropped at varying speeds and trajectories from the top of the panel. If the ball touches a wall, it rebounds. If the ball reaches the floor, it continues off screen, and the number of returns is decremented. This number is displayed in the top-left corner of the screen, as a total out of 16. When it drops to 0, the game is over.

If the player manages to position the ant under the ball, it will rebound, and the number of returns will be incremented as the ball disappears off the top. When the number of returns reaches 16, the game finishes.

Only one ball is sent falling at a time, and the ball graphic varies each time, cycling through several possibilities.

A MIDI sequence (the *BladeRunner* theme by Vangelis) is continuously played in the background, and various thumps, bangs, and boings are heard when the ball hits the walls or the ant. The game finishes with applause (no matter what the score).

The ant images come from the SpriteLib sprite library by Ari Feldman at http://www.arifeldman.com/games/spritelib.html.

2. UML Diagrams for BugRunner

Figure 2 shows the UML diagrams for the classes in the BugRunner application. The class names and public methods are given for the new classes, but only class names are supplied for the imaging and audio classes, which are unchanged from earlier chapters.

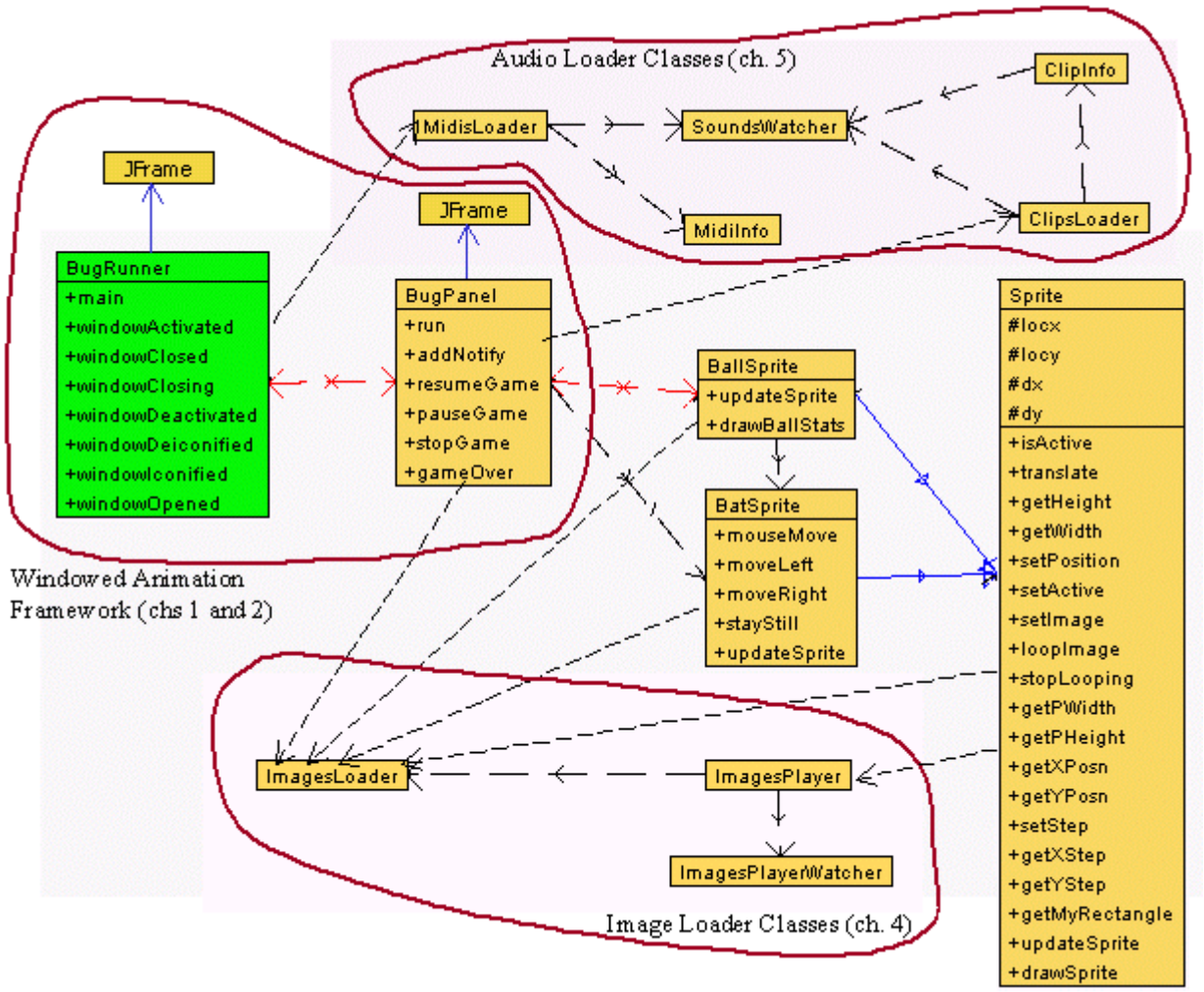

Figure 2. Class Diagrams for BugRunner.

The image and audio loader classes will not be explained again in any detail, so if you've skipped ahead to this chapter, you may want to go back to chapters 4 and 5.

The top-level JFrame (BugRunner) and the games panel (BugPanel) use the windowed animation framework developed in chapters 1 and 2. In particular, the complicated run() method in BugPanel is virtually identical to the one in WormPanel in chapter 2. If you're unfamiliar with it, then you should have a look at that chapter before continuing.

The new material is mostly concentrated in the Sprite class, and its subclasses, BallSprite and BatSprite. BallSprite manages each ball, while BatSprite handles the ant.

3. The BugRunner Class

BugRunner fixes the frame rate to be 40 FPS; anything faster makes it almost impossible to move the ant quickly enough to intercept a dropping ball.

The constructor loads and starts the *BladeRunner* sequence:

```
// load the background MIDI sequence
midisLoader = new MidisLoader();
midisLoader.load("br", "blade_runner.mid");
midisLoader.play("br", true);   // repeatedly play it
```

Since BugRunner only plays a single sequence, it's loaded directly via a call to load() rather than being specified in a MIDI information file. MidisLoader assumes that the sequence is in the Sounds/ subdirectory.

It is almost certainly a bad idea to use a well-known piece of music, like the *BladeRunner* theme, in a game intended for widespread distribution. The thorny issue of copyright is bound to come up. I've thrown caution to the wind since *BladeRunner* is one of my favourite sci-fi movies, and the music is great to.

BugRunner sets up window listener methods for pausing and resuming the game, in a similar manner to the WormChase application in chapter 2. windowClosing() is a little different:

```
public void windowClosing(WindowEvent e)
{  bp.stopGame();
   midisLoader.close();
}
```

The call to close() in MidisLoader ensures that the sequence is definitely stopped at termination time.

4. The BugPanel Class

BugPanel is a subclass of JPanel which implements the animation framework described in chapter 1 and 2; BugPanel closely resembles the WormPanel class.

The constructor sets up keyboard and mouse listeners, prepares the ImagesLoader and ClipsLoader objects, and creates the bat and ball sprites.

```
public BugPanel(BugRunner br, long period)
{
  bugTop = br;
  this.period = period;

  setDoubleBuffered(false);
  setBackground(Color.black);
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

  setFocusable(true);
  requestFocus();    // now has focus, so receives key events

    addKeyListener( new KeyAdapter() {
```



```
      public void keyPressed(KeyEvent e)
      { processKey(e);  }   // handle key presses
    });

    // load the background image
    ImagesLoader imsLoader = new ImagesLoader(IMS_INFO);
    bgImage = imsLoader.getImage("bladerunner");

    // initialise the clips loader
    clipsLoader = new ClipsLoader(SNDS_FILE);

    // create game sprites
    bat = new BatSprite(PWIDTH, PHEIGHT, imsLoader,
                                (int)(period/1000000L) ); // in ms
    ball = new BallSprite(PWIDTH, PHEIGHT, imsLoader,
                               clipsLoader, this, bat);

    addMouseListener( new MouseAdapter() {
      public void mousePressed(MouseEvent e)
      { testPress(e.getX()); }  // handle mouse presses
    });

    // set up message font
    msgsFont = new Font("SansSerif", Font.BOLD, 24);
    metrics = this.getFontMetrics(msgsFont);

  }  // end of BugPanel()
```

The image loaded by ImagesLoader is stored in the global bgImage, and later used as the game's background image (see Figure 1). *BladeRunner* fans will recognize it as a still from the movie, and so another source of copyright problems in a commercial game.

The ClipsLoader object is stored in BugPanel, *and* passed as an argument to the ball sprite, which plays various clips when its ball hits the walls or bat. The clips information file SNDS_FILE (clipsInfo.txt) is assumed to be in the Sounds/ subdirectory. For this example, it contains:

```
hitBat jump.au
hitLeft clack.au
hitRight outch.au
gameOver clap.wav
```

The gameOver clip is used by BugPanel when the game finishes, the others are utilized by BallSprite.

4.1. User Interaction

The game panel supports user input via the keyboard and mouse, which is dealt with by processKey() and testPress(). They are attached to the listeners in the BugPanel() constructor.

processKey() handles two kinds of key operations: those related to termination (e.g. ctrl-c), and those affecting the ant (the arrow keys).

```
private void processKey(KeyEvent e)
```



```
{
  int keyCode = e.getKeyCode();

  // termination keys
  if ((keyCode==KeyEvent.VK_ESCAPE) || (keyCode==KeyEvent.VK_Q) ||
      (keyCode == KeyEvent.VK_END) ||
      ((keyCode == KeyEvent.VK_C) && e.isControlDown()) )
    running = false;

  // game-play keys
  if (!isPaused && !gameOver) {
    if (keyCode == KeyEvent.VK_LEFT)
      bat.moveLeft();
    else if (keyCode == KeyEvent.VK_RIGHT)
      bat.moveRight();
    else if (keyCode == KeyEvent.VK_DOWN)
      bat.stayStill();
  }
}  // end of processKey()
```

The game-related keys are normally mapped to calls to BatSprite methods, but are ignored if the game has been paused or finished. These extra tests aren't applied to the termination keys since it should be possible to exit the game whatever its current state.

testPress() passes the cursor's x-coordinate to BatSprite to determine which way to move the ant.

```
private void testPress(int x)
{ if (!isPaused && !gameOver)
    bat.mouseMove(x);
}
```

4.2. The Animation Loop

BugPanel implements the Runnable interface, allowing its animation loop to be placed in the run() method. run() is almost the same as the one in the WormPanel class, but without the overheads of FPS statistics gathering.

```
public void run()
/* The frames of the animation are drawn inside the while loop. */
{
  long beforeTime, afterTime, timeDiff, sleepTime;
  long overSleepTime = 0L;
  int noDelays = 0;
  long excess = 0L;

  gameStartTime = J3DTimer.getValue();
  beforeTime = gameStartTime;

  running = true;

  while(running) {
    gameUpdate();
    gameRender();
    paintScreen();
```



```
      afterTime = J3DTimer.getValue();
      timeDiff = afterTime - beforeTime;
      sleepTime = (period - timeDiff) - overSleepTime;

      if (sleepTime > 0) {   // some time left in this cycle
        try {
          Thread.sleep(sleepTime/1000000L);  // nano -> ms
        }
        catch(InterruptedException ex){}
        overSleepTime = (J3DTimer.getValue() - afterTime)
                                                - sleepTime;
      }
      else {    // sleepTime <= 0; frame took longer than period
        excess -= sleepTime;  // store excess time value
        overSleepTime = 0L;

        if (++noDelays >= NO_DELAYS_PER_YIELD) {
          Thread.yield();   // give another thread a chance to run
          noDelays = 0;
        }
      }

      beforeTime = J3DTimer.getValue();

      /* If frame animation is taking too long, update the game state
         without rendering it, to get the updates/sec nearer to
         the required FPS. */
      int skips = 0;
      while((excess > period) && (skips < MAX_FRAME_SKIPS)) {
        excess -= period;
        gameUpdate();    // update state but don't render
        skips++;
      }
    }
    System.exit(0);   // so window disappears
  } // end of run()
```

The Java 3D timer is used mainly because it is a excellent timer for J2SE 1.4.2 across a range of platforms. However, as J2SE 1.5 gains popularity, a better choice will be System.nanoTime(), a nanosecond timer that's part of Java's core packages (rather than an extension as in the case of Java 3D). Porting the code is simply a matter of replacing calls to J3DTimer.getValue() with System.nanoTime().

[Eventually, I'll rewrite all the early chapters of the book to use nanoTime() rather than the Java 3D timer.]

The application-specific elements of the animation are located in gameUpdate() and gameRender(). Also, a new gameStartTime variable is initialized at the start of run(); it is used later to calculate the elapsed time displayed in the game panel.

gameUpdate() updates the active game entities – the ball and bat sprites.

```
  private void gameUpdate()
  { if (!isPaused && !gameOver) {
      ball.updateSprite();
      bat.updateSprite();
```



```
    }
  }
```

gameRender() draws the background, the sprites, and the game statistics (the number of rebounds and the elapsed time).

```
  private void gameRender()
  {
    if (dbImage == null){
      dbImage = createImage(PWIDTH, PHEIGHT);
      if (dbImage == null) {
        System.out.println("dbImage is null");
        return;
      }
      else
        dbg = dbImage.getGraphics();
    }

    // draw the background: use the image or a black screen
    if (bgImage == null) {  // no background image
      dbg.setColor(Color.black);
      dbg.fillRect (0, 0, PWIDTH, PHEIGHT);
    }
    else
      dbg.drawImage(bgImage, 0, 0, this);

    // draw game elements
    ball.drawSprite(dbg);
    bat.drawSprite(dbg);

    reportStats(dbg);

    if (gameOver)
      gameOverMessage(dbg);
  }  // end of gameRender()
```

gameUpdate() and gameRender() show the main way that the sprites are utilized: first their states are updated via calls to updateSprite(), then they are drawn by invoking drawSprite().

reportStats() calculates and renders the current time and the number of rebounds.

```
  private void reportStats(Graphics g)
  {
    if (!gameOver)   // stop incrementing timer once game is over
      timeSpentInGame =
          (int) ((J3DTimer.getValue() - gameStartTime)/1000000000L);
                    // ns --> secs

    g.setColor(Color.yellow);
    g.setFont(msgsFont);

    ball.drawBallStats(g, 15, 25);  // ball sprite reports ball stats
    g.drawString("Time: " + timeSpentInGame + " secs", 15, 50);

    g.setColor(Color.black);
  }
```


The number of rebounds is reported by the ball sprite, which is passed the graphics context in the drawBallStats() call.

4.3. Finishing the Game

The game is terminated when the gameOver boolean is set to true. This stops any further updates to the active entities via gameUpdate(), and disables the processing of keyboard and mouse actions. However, the screen is still periodically redrawn, and the background music keeps playing until the application is closed.

The gameOver boolean is set by the BallSprite object calling gameOver() in BugPanel.

```
public void gameOver()
{ int finalTime =
        (int) ((J3DTimer.getValue() - gameStartTime)/1000000000L);
                      // ns --> secs
  score = finalTime;   // could be more fancy!
  clipsLoader.play("gameOver", false);   // play clip once
  gameOver = true;
}
```

A score is calculated and the gameOver clip (polite applause) is played.

5. Defining a Sprite

A general-purpose Sprite class is surprisingly hard to design, since many of its features depend on the application and the gaming context.

For example, a sprite's on-screen movement greatly depends on the type of game. In Tetris, Breakout, and Space Invaders (and many more), the sprite moves within the gaming area while the background scenery remains stationary. In some of these games, the sprite may be unable to move beyond the edges of the panel, in others it can wraparound to the opposite edge. In side-scrolling games, such as Super Mario, the sprite hardly moves at all (perhaps only up and down); instead the background shifts behind it.

A sprite must monitor the game environment, for example, reacting to collisions with different sprites, or stopping when it encounters an obstacle.

Collision processing can be split into collision detection and response, with the range of responses being very application specific. There are also many varieties of collision detection: a sprite may be represented by a single bounding box, a reduced size bounding box, or several bounding areas. Examples are shown in Figure 3 (the bounding regions are the red dotted boxes around the pigeon and donkey).

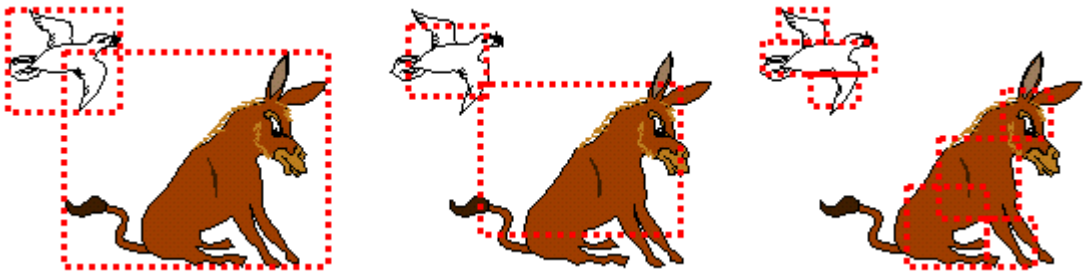

Figure 3. Three Types of Collision Detection.

A single bounding box is simple to manipulate but prone to inaccuracy. The reduced bounding box is better, but choosing a suitable reduction factor is difficult. The greatest accuracy can be achieved with several boxes for each sprite, but at the expense of additional calculations.

Sometimes a 2D sprite will have a z coordinate (or z-level) which dictates its drawing position (or order) on screen. For instance, if two sprites have the same z-level then they'll be unable to move past each other, and so collide. However, a sprite with a smaller z-level is 'in front' of a sprite with a larger z-level and so can pass it. Sprites are drawn in decreasing z-level order, so that sprites in the foreground appear in front of those further back.

The visual appearance of a sprite typically changes over time, either in response to important events (e.g. being shot), or by cycling through a series of images (e.g. moving its arms and legs as it walks about). There may be associated audio effects (e.g. a gun shot sound), triggered by events or played periodically.

6. The Sprite Class

Our Sprite class is quite simple, storing little more than the sprite's current position, its speed specified as step increments in the x- and y- direction, with imaging managed by ImagesLoader and ImagesPlayer objects. The ImagesPlayer class allows the sprite to show a sequence of images repeatedly (this is how the ant moves its legs).

The Sprite subclasses, BatSprite and BallSprite in BugRunner, manage user interactions, environment concerns (e.g. collision detection and response), and audio effects. These elements are too application specific to be placed in Sprite.

6.1. The Sprite() Constructor

A sprite is initialized with its position, the size of the enclosing panel, an ImagesLoader object, and the name of an image:

```
// default step sizes (how far to move in each update)
private static final int XSTEP = 5;
private static final int YSTEP = 5;
     :

private ImagesLoader imsLoader;
private int pWidth, pHeight;   // panel dimensions
     :

// protected vars
protected int locx, locy;      // location of sprite
protected int dx, dy;          // amount to move for each update


public Sprite(int x, int y, int w, int h,
                   ImagesLoader imsLd, String name)
{ locx = x; locy = y;
  pWidth = w; pHeight = h;
  dx = XSTEP; dy = YSTEP;
  imsLoader = imsLd;
  setImage(name);    // the sprite's default image is 'name'
}
```

The sprite's coordinate (locx, locy) and its step values (dx, dy) are stored as integers. This simplifies certain tests and calculations, but restricts positional and speed precision. For instance, a ball cannot move 0.5 pixels at a time. The alternative is to use floats or doubles to hold the coordinates and velocities.

locx, locy, dx, and dy are protected rather than private, due to their widespread use in Sprite subclasses. They also have get and set methods, so can be accessed and changed by objects outside of the Sprite hierarchy.

Sprite only stores (x, y) coordinates, no z-coordinate or z-level; such functionality is unnecessary in BugRunner. Simple z-level functionality can be achieved by ordering the calls to drawSprite() in gameRender(). Currently, the code is:

```
ball.drawSprite(dbg);
bat.drawSprite(dbg);
```


The ball is drawn before the bat, and so will appear behind it if they happen to overlap on-screen.

6.2. A Sprite's Image

setImage() assigns the named image to the sprite:

```
// default dimensions when there is no image
private static final int SIZE = 12;
      :

// image-related globals
private ImagesLoader imsLoader;
private String imageName;
private BufferedImage image;
private int width, height;     // image dimensions

private ImagesPlayer player;  // for playing a loop of images
private boolean isLooping;
     :

public void setImage(String name)
{
  imageName = name;
  image = imsLoader.getImage(imageName);
  if (image == null) {    // no image of that name was found
    System.out.println("No sprite image for " + imageName);
    width = SIZE;
    height = SIZE;
  }
  else {
    width = image.getWidth();
    height = image.getHeight();
  }
  // no image loop playing
  player = null;
  isLooping = false;
}
```

setImage() is a public method, permitting the sprite's image to be altered at runtime.

An ImagesPlayer object, player, is available to the sprite for looping through a sequence of images. Looping is switched on with the the loopImage() method:

```
public void loopImage(int animPeriod, double seqDuration)
{
  if (imsLoader.numImages(imageName) > 1) {
    player = null;   // for garbage collection of previous player
    player = new ImagesPlayer(imageName, animPeriod, seqDuration,
                                    true, imsLoader);
    isLooping = true;
  }
  else
    System.out.println(imageName + " is not a sequence of images");
}
```


The total time for the loop is seqDuration secs. The update interval (supplied by the enclosing animation panel) is animPeriod ms.

Looping is switched off with stopLooping():

```
public void stopLooping()
{ if (isLooping) {
    player.stop();
    isLooping = false;
  }
}
```

6.3. A Sprite's Bounding Box

Collision detection and the response is left to subclasses. However, the bounding box for the sprite is available through the getMyRectangle() method:

```
public Rectangle getMyRectangle()
{  return  new Rectangle(locx, locy, width, height);  }
```

Sprite uses the simplest form of bounding box, but it wouldn't be difficult to introduce a reduction factor.

6.4. Updating a Sprite

A sprite is updated by adding its step values (dx, dy) to its current location (locx, locy).

```
// global
private boolean isActive = true;
  // a sprite is updated and drawn only when active
     :

public void updateSprite()
{
  if (isActive()) {
    locx += dx;
    locy += dy;
    if (isLooping)
      player.updateTick();  // update the player
  }
}
```

The isActive boolean allows a sprite to be (temporarily) removed from the game, since it will not be updated or drawn when isActive is false. There are public isActive () and setActive() methods for manipulating the boolean.

No attempt is made in updateSprite() to test for collisions with other sprites, obstacles, or the edges of the gaming pane. These must be added by the subclasses when they override updateSprite().

Sprites are embedded in an animation framework which works very hard to maintain a fixed frame rate. run() calls updateSprite() in all the sprites at a frequency as close to

the specified frame rate as possible. For example, if the frame rate is 40 FPS (as it is in BugRunner), then updateSprite() will be called 40 times per second in each sprite.

This allows us to make assumptions about a sprite's update timing. For instance, if the x-axis step value (dx) is 10, then the sprite will be moved 10 pixels in each update. This corresponds to a speed of 10*40 = 400 pixels/second along that axis. This calculation is only possible because the frame rate is tightly constrained to 40 FPS.

An alternative approach is to call updateSprite() with an argument holding the elapsed time since the previous call. This time value can be multiplied to a velocity value to get the step amount for this particular update. This technique is preferably in animation frameworks where the frame rate can vary during execution.

6.5. Drawing a Sprite

The animation loop will call updateSprite() in a sprite, followed by drawSprite() to draw it.

```
public void drawSprite(Graphics g)
{
  if (isActive()) {
    if (image == null) {   // the sprite has no image
      g.setColor(Color.yellow);   // draw a yellow circle instead
      g.fillOval(locx, locy, SIZE, SIZE);
      g.setColor(Color.black);
    }
    else {
      if (isLooping)
        image = player.getCurrentImage();
      g.drawImage(image, locx, locy, null);
    }
  }
}
```

The two special cases are when the image is null and when a looping sequence of images is being displayed. If the image is null, then the sprite's default appearance is a small yellow circle. The current image in the looping series is obtained by calling ImagesPlayer's getCurrentImage() method.

7. Specifying a Sprite with a Statechart

A sprite is a reactive object: it responds dynamically to events, changing its state, modifying its behaviour. For all but the simplest example, it's a good idea to specify the sprite's behaviour before becoming entangled in code. The UML statechart is an excellent tool for defining a sprite, and there are even tools for generating Java code from the statecharts (or their close equivalents).

The rest of this section gives a brief introduction to statecharts, and explains how they can be translated to Java. However, this isn't a book about UML, and so I refer the interested reader to:

UML Distilled: A Brief Guide to the Standard Object Modeling Language
Martin Fowler
Addison-Wesley, 3rd edition, 2003

A UML text with a Java slant:

UML for Java Programmers
Robert C. Martin
Prentice Hall, 2003
`http://www.objectmentor.com`

Martin's Web site offers SMC, a translator which takes textual statechart information and generates Java (or C++).

A simple statechart for a subway turnstile is shown in Figure 4 (this example comes from a tutorial by Robert C. Martin).

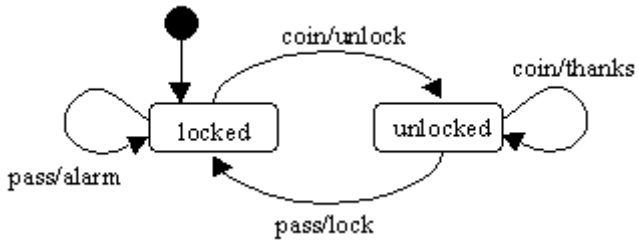

Figure 4. A Subway Turnstile Statechart.

The states are inside rounded rectangles, the transitions (arrows) indicate how the system responds to events. The syntax for a transition is:

event [condition] / action

An event arrives when the system is in a given state, causing the associated transition to be executed. First the condition is evaluated (if one is present). If the result is true, then the associated action is carried out, and the system follows the transition to a new state.

The solid black circle points to the starting state for the system.

The statechart in Figure 4 specifies what happens when a 'pass' or 'coin' event occurs in the 'locked' or 'unlocked' states for a turnstile. A 'coin' event corresponds to a customer placing a coin in the turnstile; a 'pass' event is generated when the turnstile rotates to allow the customer to pass through.

Statecharts can be *considerably* more sophisticated than this example. For instance, they can represent activities carried out when a state is entered and exited, states can be nested to form hierarchies, states can be grouped together to model concurrent actions, and there is a history mechanism.

7.1. Translating a Statechart to Code

There are various approaches for translating a statechart into executable code. One of the simplest is to first convert the graphical notation into table form, called a State Transition Table (STT). Table 1 shows the STT for the turnstile statechart.

Current State	Event	Action	New State
locked	coin	unlock	unlocked
locked	pass	alarm	locked
unlocked	coin	thanks	unlocked
unlocked	pass	lock	locked

Table 1. Turnstile STT.

An imperative-style translation of the table converts it to a series of if statements in makeTransition():

```
// state constants
private static final int LOCKED = 0;
private static final int UNLOCKED = 1;

// event constants
private static final int COIN = 2;
private static final int PASS = 3;


public static void main(...)
{
  int currentState = LOCKED;
  int event;
  while (true) {
    event = /* get the next event */;
    currentState = makeTransition(currentState, event);
  }
} // end of main()


private static int makeTransition(int state, int event)
// a translation of Table 1
{
  if ((state == LOCKED) && (event == COIN)) {
    unlock();
    return UNLOCKED;
  }
  else if ((state == LOCKED) && (event == PASS)) {
    alarm();
    return LOCKED;
  }
  else if ((state == UNLOCKED) && (event == COIN)) {
    thanks();
    return UNLOCKED;
  }
  else if ((state == UNLOCKED) && (event == PASS)) {
    lock();
    return LOCKED;
```



```
  }
  else {
    System.out.println("Unknown state event");
    System.exit(0);
  }
}  // end of makeTransition()
```

The translation strategy is to represent states and events as integers, and transition actions as method calls. If a transition has a condition, then it is added to the if-test for that transition.

The drawback of this approach is the generation of very long sequences of if-tests, often with multiple conditions. Fortunately, the code can often be rewritten to make it easier to understand (e.g. makeTransition() could be divided into several smaller methods).

As states become more complex (e.g. hierarchical, with internal activities), it is more natural to map a state to a class, and a transition to a method. The SMC translator takes this approach (http://www.objectmentor.com/resources/downloads/).

A third coding solution is to employ a set of existing statechart classes (e.g. State, Transition), subclassing them for the particular application. Excellent examples of this approach can be found in:

> *Practical Statecharts in C/C++*
> Miro Samek
> CMP Books, July 2002
> `http://www.quantum-leaps.com`

As the book title suggests, the emphasis is on C and C++, but the author's Web site contains a Java version of the software (which requires a password generated from the book).

8. The BallSprite Class

We begin with a textual specification of what a ball sprite should do, translate this into a statechart, and then manually convert it into BallSprite, a subclass of Sprite.

8.1. Textual Specification

The ball drops from the top of the panel at varying speeds and angles of trajectory. It will bounce off a wall if it hits one, reversing its x-axis direction. If the ball hits the bat, it will rebound and eventually disappear off the top of the panel. If the ball passes the bat, it will disappear through the panel's base.

After leaving the panel, the ball is reused: it is placed back at the top and put into motion again. The image associated with the ball is also changed.

The number of 'returned' balls is incremented when the ball bounces off the bat (numRebounds is incremented). When the ball drops off the bottom, numRebounds is decremented. If numRebounds reaches MAX_BALLS_RETURNED, the game is over. If numRebounds reaches 0, the game also terminates.

Sound effects are played when the ball hits the walls and the bat.

8.2. Statechart Specification

The statechart in Figure 5 specifies the ball sprite.

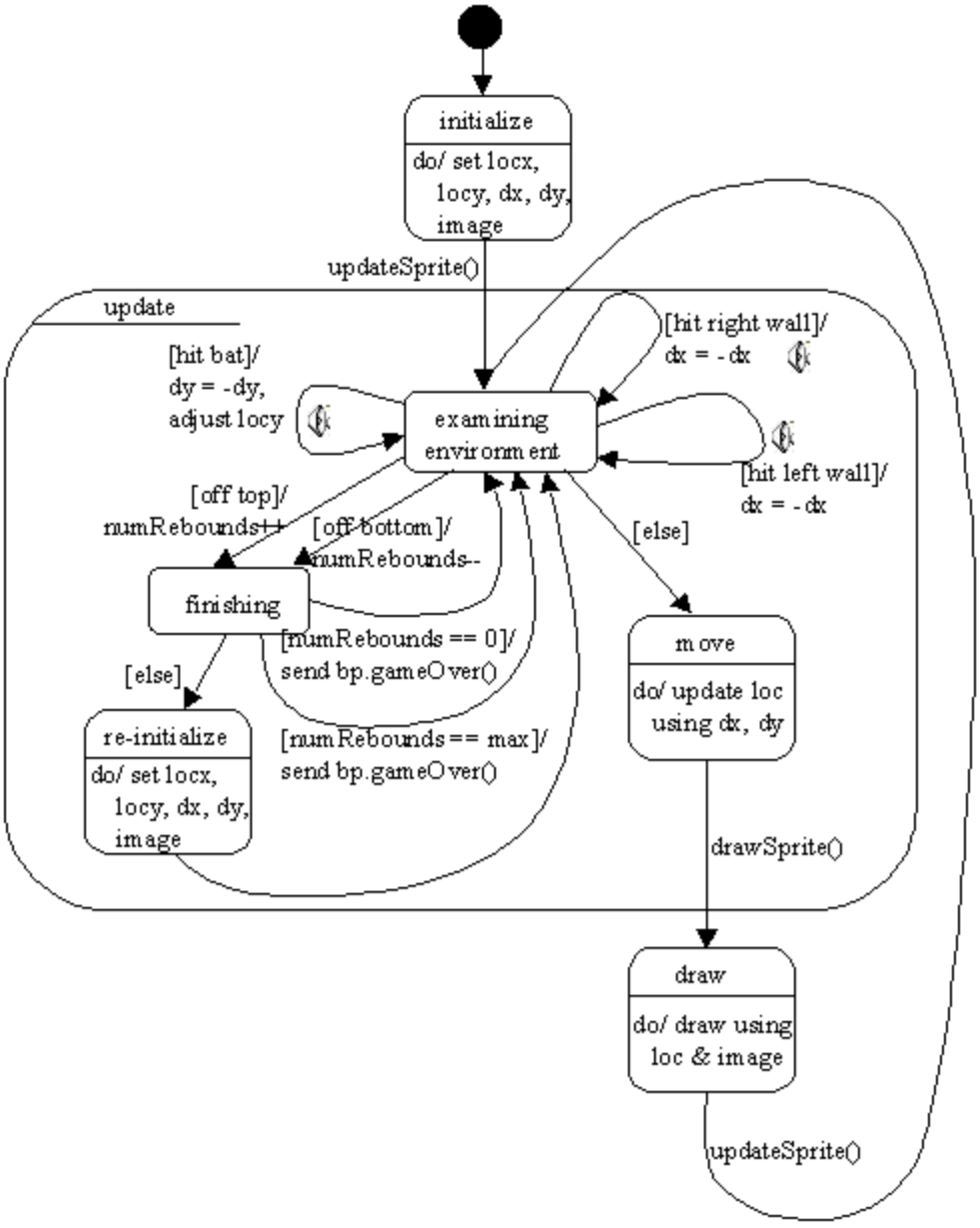

Figure 5. The BallSprite Statechart.

The statechart uses an 'update' *superstate* which encapsulates the states which modify the sprite's location, step sizes, and other values before the ball is moved and drawn.

The 'update' superstate highlights the sprite's update/draw cycle, which is driven by the method calls updateSprite() and drawSprite(), originating from BugPanel's animation loop.

A "do/" activity inside a state is carried out while the sprite occupies that state.

The 'examining environment' state deals with the unusual situations that may occur as the ball descends through the JPanel. The transitions leaving 'examining environment' are triggered by tests on the sprite's current location. Hitting the left or right walls causes a change in direction along the x-axis. Moving past the top or bottom edges of the panel places the ball in a finishing state. This may result in the sprite notifying the BugPanel object (bp) that the game is over, or the ball may be reused.

When none of the special environmental conditions apply, the sprite is moved and redrawn.

The speaker icons next to the [hit bat], [hit right wall], and [hit left wall] conditional transitions indicate that a sound effect will be played when the condition evaluates to true.

8.3. Translating the Statechart

The 'initialize' state is covered by BallSprite's constructor.

```
// images used for the balls
private static final String[] ballNames =
            {"rock1", "orangeRock", "computer", "ball"};

// reach this number of balls to end the game
private static final int MAX_BALLS_RETURNED = 16;
           :

// globals
private ClipsLoader clipsLoader;
private BugPanel bp;
private BatSprite bat;
private int numRebounds;
           :


public BallSprite(int w, int h, ImagesLoader imsLd, ClipsLoader cl,
                                                BugPanel bp, BatSprite b)
{ super( w/2, 0, w, h, imsLd, ballNames[0]);
    // the ball is positioned in the middle at the top of the panel
  clipsLoader = cl;
  this.bp = bp;
  bat = b;

  nameIndex = 0;
  numRebounds = MAX_BALLS_RETURNED/2;
      // the no. of returned balls starts half way to the maximum
  initPosition();
}
```

The names of four ball images are fixed in ballNames[]; they are the names of four GIF files stored in the Sounds/ subdirectory.

BallSprite stores a reference to the BugPanel so it can notify the panel when the game is over. The BatSprite reference is used for collision detection (carried out by the [hit bat] condition for 'examining environment'). The ClipsLoader reference enables the sprite to play sounds when it hits the bat or rebounds from the walls.

The numRebounds variable performs the same task as the variable in the statechart. It begins with a value half way between 0 and the maximum number of returns required for winning the game.

initPosition() initializes the ball's image, position and step values.

```
private void initPosition()
{
  setImage( ballNames[nameIndex]);
  nameIndex = (nameIndex+1)%ballNames.length;

  setPosition( (int)(getPWidth() * Math.random()), 0);
                                // somewhere along the top

  int step = STEP + getRandRange(STEP_OFFSET);
  int xStep = ((Math.random() < 0.5) ? -step : step);
                                  // move left or right
  setStep(xStep, STEP + getRandRange(STEP_OFFSET));   // move down
}


private int getRandRange(int x)
// random number generator between -x and x
{   return ((int)(2 * x * Math.random())) - x;  }
```

setImage(), setPosition() and setStep() are methods inherited from Sprite.

The 'update' superstate is represented by an overridden updateSprite():

```
public void updateSprite()
{
  hasHitBat();
  goneOffScreen();
  hasHitWall();

  super.updateSprite();
}
```

The calls to hasHitBat(), goneOffScreen(), and hasHitWall() roughly correspond to the 'examining environment', 'finishing', and 're-initialize' states and transitions, while the call to Sprite's updateSprite() implements the 'move' state.

The looping behaviour of 'examining environment' has been simplified to a sequential series of tests. This is possible since the special conditions (e.g. hit a wall, hit the bat) do not occur more than once during a single update, and are independent of each other. However, such optimizations should be done carefully to avoid changing the sprite's intended behaviour.

hasHitBat() implements the [hit bat] conditional transition:

```
private void hasHitBat()
{
  Rectangle rect = getMyRectangle();
  if (rect.intersects( bat.getMyRectangle() )) {  // bat collision?
    clipsLoader.play("hitBat", false);
```



```
      Rectangle interRect = rect.intersection(bat.getMyRectangle());
      dy = -dy;       // reverse ball's y-step direction
      locy -= interRect.height;    // move the ball up
    }
  }
```

Collision detection is a matter of seeing if the bounding boxes for the ball and the bat intersect. A sound effect is played if they overlap, and the ball is made to bounce by having its y-step reversed.

The ball's y-axis location is also moved up slightly, so it no longer intersects the bat. This rules out the (slim) possibility that the collision test of the ball and bat during the next update will find them still overlapping. This would occur if the rebound velocity was too small to separate the objects within one update.

The collision algorithm could be improved. For instance, some consideration could be given to the relative positions and speeds of the ball and bat to determine the direction and speed of the rebound. This would complicate the coding, but improve the ball's visual appeal.

hasHitWall() handles the [hit right wall] and [hit left wall] conditional transitions:

```
  private void hasHitWall()
  {
    if ((locx <= 0) && (dx < 0)) {  // touching lhs and moving left
      clipsLoader.play("hitLeft", false);
      dx = -dx;   // move right
    }
    else if ((locx+getWidth() >= getPWidth()) && (dx > 0)) {
                                    // touching rhs and moving right
      clipsLoader.play("hitRight", false);
      dx = -dx;   // move left
    }
  }
```

hasHitWall() is made easier to understand by having it directly referring to the sprite's x-axis location (locx) and step sizes (dx and dy). They are protected variables, inherited from Sprite.

The if-tests illustrate a very common form of testing: a combination of location and direction tests. It isn't enough just to examine the sprite's location, we must also determine whether the sprite is heading out of the panel. This will correctly exclude a recently rebounded sprite, which is still near an edge but moving away from it.

A subtle coding assumption is that the boundaries of the BufferedImage correspond to the visible edges of the image on-screen. For example, a sprite surrounded by a large transparent border, as shown in Figure 6, would not seem to be touching the panel's left edge when its top-left x-coordinate (locx) is at pixel 0.

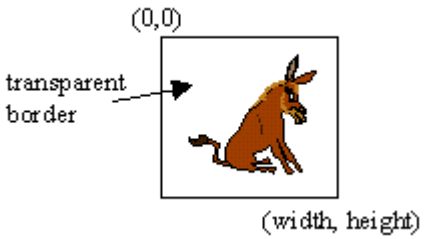

Figure 6. An Image with a large Transparent Border.

The simplest solution is to ensure that images are cropped to have as small a transparent border as possible. Alternatively, bounding box calculations could use a reduction factor to shrink the bounding region.

goneOffScreen() implements the [off top] and [off bottom] conditional transitions, and the 'finishing' and 're-initialize' states connected to them.

```
private void goneOffScreen()
{
  if (((locy+getHeight()) <= 0) && (dy < 0)) {
    numRebounds++;      // off top and moving up
    if (numRebounds == MAX_BALLS_RETURNED)
      bp.gameOver();    // finish
    else
      initPosition();   // start the ball in a new position
  }
  else if ((locy >= getPHeight()) && (dy > 0)) {
    numRebounds--;      // off bottom and moving down
    if (numRebounds == 0)
      bp.gameOver();
    else
      initPosition();
  }
}
```

The 'finishing' state and its exiting transitions have been mapped to the bodies of the if-tests. The 're-initialize' state has been implemented by reusing initPosition(), which is part of the 'initialize' code.

The 'draw' state in BallSprite's statechart has no equivalent method in the BallSprite class. It is handled by the inherited drawSprite() method from Sprite.

9. The BatSprite Class

We proceed in the same way as for BallSprite. A textual specification is given for what a bat sprite should do, it is translated into a statechart, and then manually converted into BatSprite, a subclass of Sprite.

9.1. Textual Specification.

The bat can only move horizontally across the floor, controlled by arrow keys and mouse presses. Once the bat is set in motion, it continues moving until its direction is changed or it is stopped. As the bat leaves one side of the panel, it appears on the other side. The bat is assigned a left-facing and right-facing set of images (a walking ant), which are cycled through as the ant moves.

9.2. Statechart Specification

A bat sprite statechart appears in Figure 7.

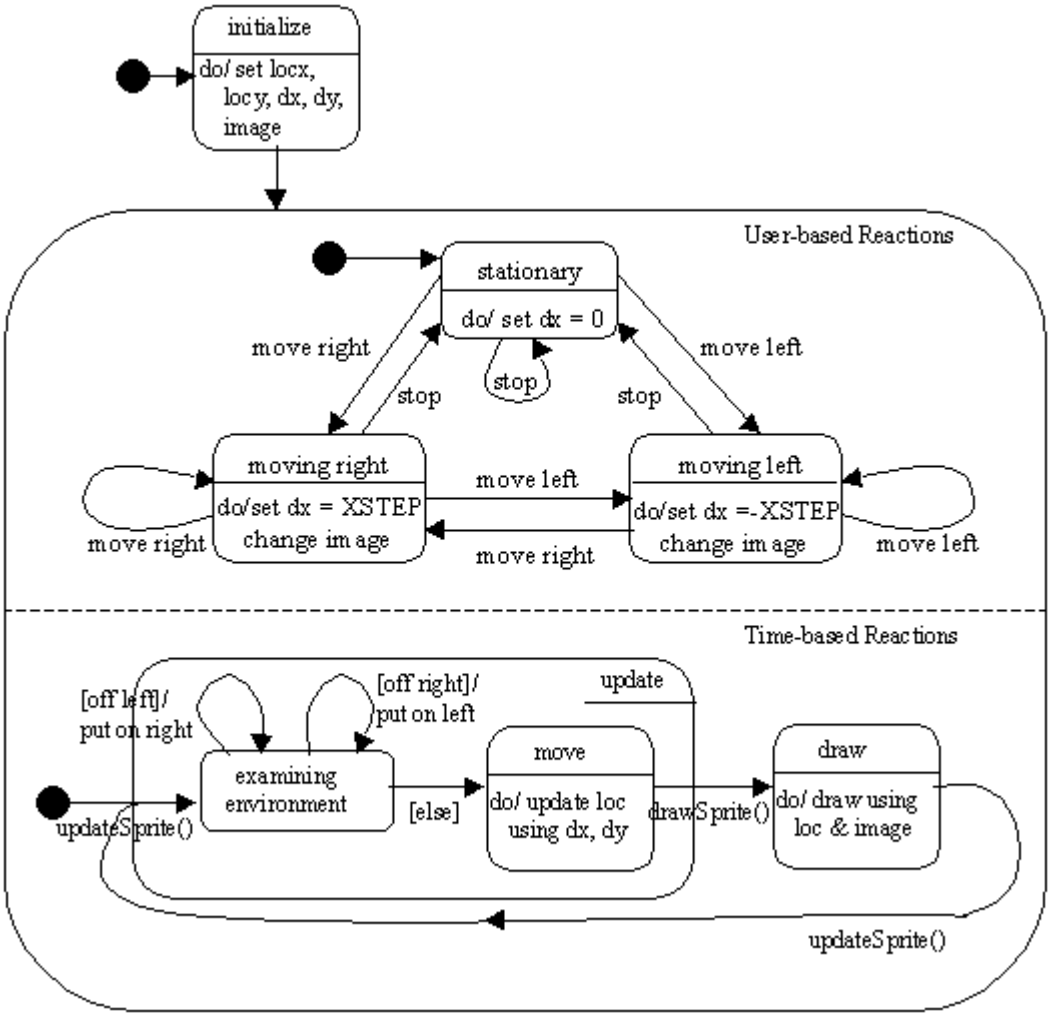

Figure 7. The BatSprite Statechart.

The new statechart element in Figure 7 is the *concurrent state diagram* with two sections labeled 'User-based Reactions' and 'Time-based Reactions'. Concurrent state

diagrams allow the modeling of concurrent activities, which are present in all sprites controlled by the user.

The 'Time-based Reactions' section illustrates the update/draw cycle carried out by the animation loop.

The 'User-based Reactions' section encapsulates the changes made to the sprite by the user pressing the arrow keys and/or the mouse. A 'move right' transition occurs when the user presses the right arrow key or clicks the mouse to the right of the bat. The 'move left' transition handles the left arrow key and a mouse press to the left of the bat. The 'stop' transition deals with the down arrow key and a mouse press over the bat.

The implementation of 'User-based Reactions' uses listener methods, which are processed in Swing's event thread. This means there is no need for explicit threaded coding in the BatSprite class.

The statechart nicely highlights an important issue with user-controlled sprites: the concurrent sharing of data between the animation loop and the event processing code. The statechart shows that the shared data will be the sprite's x-axis step size (dx), and the current image.

9.3. Translating the Statechart

The 'initialize' state is covered by BatSprite's constructor.

```
// globals
private static final int FLOOR_DIST = 41;
    // distance of ant's top from the floor
     :

private int period;
 /* in ms. The game's animation period used by the image
    cycling of the bat's left and right facing images. */
     :


public BatSprite(int w, int h, ImagesLoader imsLd, int p)
{
  super( w/2, h-FLOOR_DIST, w, h, imsLd, "leftBugs2");
      // positioned at the bottom of the panel, near the center
  period = p;
  setStep(0,0);  // no movement
}
```

9.3.1. User-based Reactions

The key presses which trigger 'move left', 'move right', and 'stop' events are caught by BugPanel's key listener, which calls processKey(). Inside processKey(), the code for responding to the arrow keys is:

```
if (!isPaused && !gameOver) {
  if (keyCode == KeyEvent.VK_LEFT)
```



```
      bat.moveLeft();
    else if (keyCode == KeyEvent.VK_RIGHT)
      bat.moveRight();
    else if (keyCode == KeyEvent.VK_DOWN)
      bat.stayStill();
  }
```

moveLeft() implements the 'moving left' state in Figure 7.

```
  // global
  private static final int XSTEP = 10;
     // step distance for moving along x-axis
      :

  public void moveLeft()
  { setStep(-XSTEP, 0);
    setImage("leftBugs2");
    loopImage(period, DURATION);   // cycle through leftBugs2 images
  }
```

leftBugs2 is the name for a GIF file in Sounds/ which contains an image strip of ants walking to the left.

moveRight() handles the 'moving right' state in Figure 7.

```
  public void moveRight()
  { setStep(XSTEP, 0);
    setImage("rightBugs2");
    loopImage(period, DURATION);  // cycle through the images
  }
```

The 'stationary' state is encoded by stayStill().

```
  public void stayStill()
  { setStep(0, 0);
    stopLooping();
  }
```

This translation of the statechart is possible because of a property of the events. A 'move left' event always enters the 'moving left' state, a 'move right' event always enters 'moving right', and 'stop' always goes to the 'stationary' state. This means that we don't need to consider the current state in order to determine the next state when a given event arrives.

'move left', 'move right', and 'stop' events can also be triggered by mouse actions. BugPanel employs a mouse listener to call testPress() when a mouse press is detected.

```
  private void testPress(int x)
  { if (!isPaused && !gameOver)
      bat.mouseMove(x);
  }
```


BatSprite's mouseMove() calls one of its move methods depending on the cursor's position relative to the bat.

```
public void mouseMove(int xCoord)
{
 if (xCoord < locx)  // click was to the left of the bat
   moveLeft();       // make the bat move left
 else if (xCoord > (locx + getWidth()))  // click was to the right
   moveRight();      // make the bat move right
 else
   stayStill();
}
```

9.3.2. Time-based Reactions

The 'update' superstate in the 'Time-based Reactions' section is coded by overriding updateSprite() (in a similar way to in BallSprite).

```
public void updateSprite()
{
  if ((locx+getWidth() <= 0) && (dx < 0))   // almost gone off lhs
    locx = getPWidth()-1;      // make it just visible on the right
  else if ((locx >= getPWidth()-1)&&(dx>0)) // almost gone off rhs
    locx = 1 - getWidth();     // make it just visible on the left

  super.updateSprite();
}
```

The looping behaviour of the 'examining environment' has been simplified so that the [off left] and [off right] conditional transitions are implemented as two sequential if-tests. The 'move' state is handled by calling Sprite's updateSprite().

The 'draw' state in BatSprite's statechart has no equivalent method in the BatSprite class. As in BallSprite, it is handled by the inherited drawSprite() method from Sprite.

9.3.3. Concurrently Shared Data

BatSprite makes no attempt to synchronize the accesses made to the data shared between the 'User-based Reactions' and 'Time-based Reactions' sections. The shared data is the sprite's x-axis step size (dx), and the current image.

The step size is modified by moveLeft(), moveRight(), and stayStill() when they call the inherited setStep() method. Possibly at the same time, the step is used by Sprite's updateSprite() method. In practice however, there is little chance of the assignment interfering with the read.

moveLeft() and moveRight() assign a new object to the image reference by calling the inherited setImage() method. Meanwhile, in Sprite, drawSprite() passes the reference to drawImage() to draw the image on-screen. The same argument applies: the assignment is a single operation which is unlikely to affect the rendering.

