10.3 Behavioral Patterns

- 10.3.1 Design Pattern : Command
  - in this section
    - we focus on the behavior of the maze game
    - the player starts in a specified room of the maze board.
    - The player can be moved with the arrow keys
    - the player can move to an adjacent room through an open door
    - we support the undoing of moves by the player.
    - to support the undoing of actions is to use the command pattern.

10.3 Behavioral Patterns

- Design Pattern : Command
  - Category : Behavioral design pattern
  - Intent : To encapsulate an action as an object, so that actions can be passed as parameters, queued, and possibly undone.
  - Also Known As : Action.
  - Applicability : Use the Command design pattern
    - when actions need to be passed as parameters.
    - when actions need to be queued and then executed later.
    - when actions can be undone.

Design Pattern *Command*

Category: Behavioral design pattern.

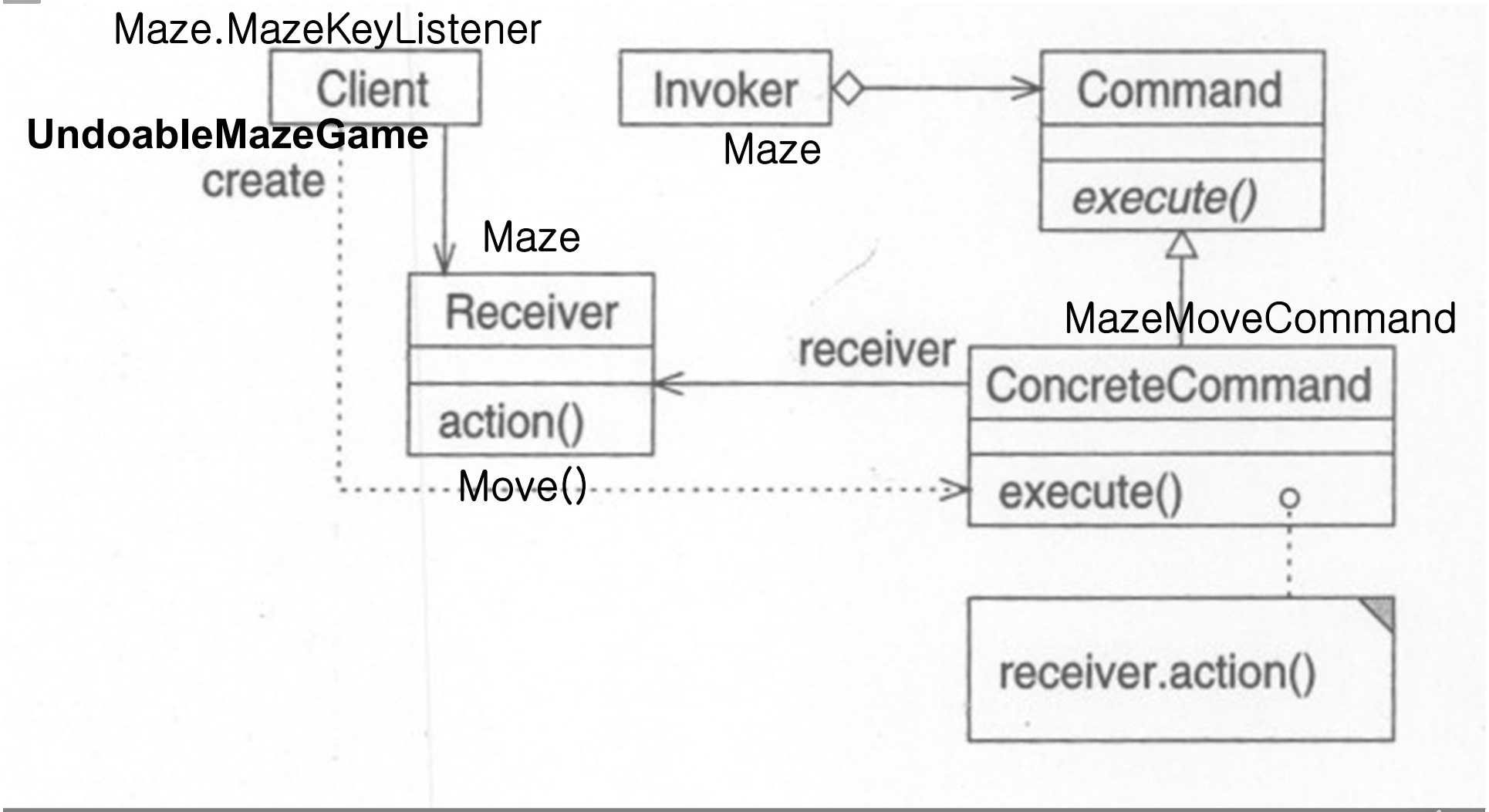
Intent: To encapsulate an action as an object, so that actions can be passed as parameters, queued, and possibly undone.

Also Known As: Action.

Applicability: Use the Command design pattern

- when actions need to be passed as parameters.
- when actions need to be queued and then executed later.
- when actions can be undone.

The structure of the Command design pattern (Page 508)



- *Command* (e.g., `Command`, `UndoableCommand`), which defines an interface to perform or undo an action.
- *Receiver* (e.g., `Maze`), which knows how to perform the actions.
- *ConcreteCommand* (e.g., `MazeMoveCommand`), which implements the *Command* interface and delegates the execution of the action to the *Receiver*.
- *Client* (e.g., `Maze.MazeKeyListener`), which creates the concrete commands and binds the concrete commands to their receivers.
- *Invoker* (e.g., `Maze`), which asks the command to carry out the action.

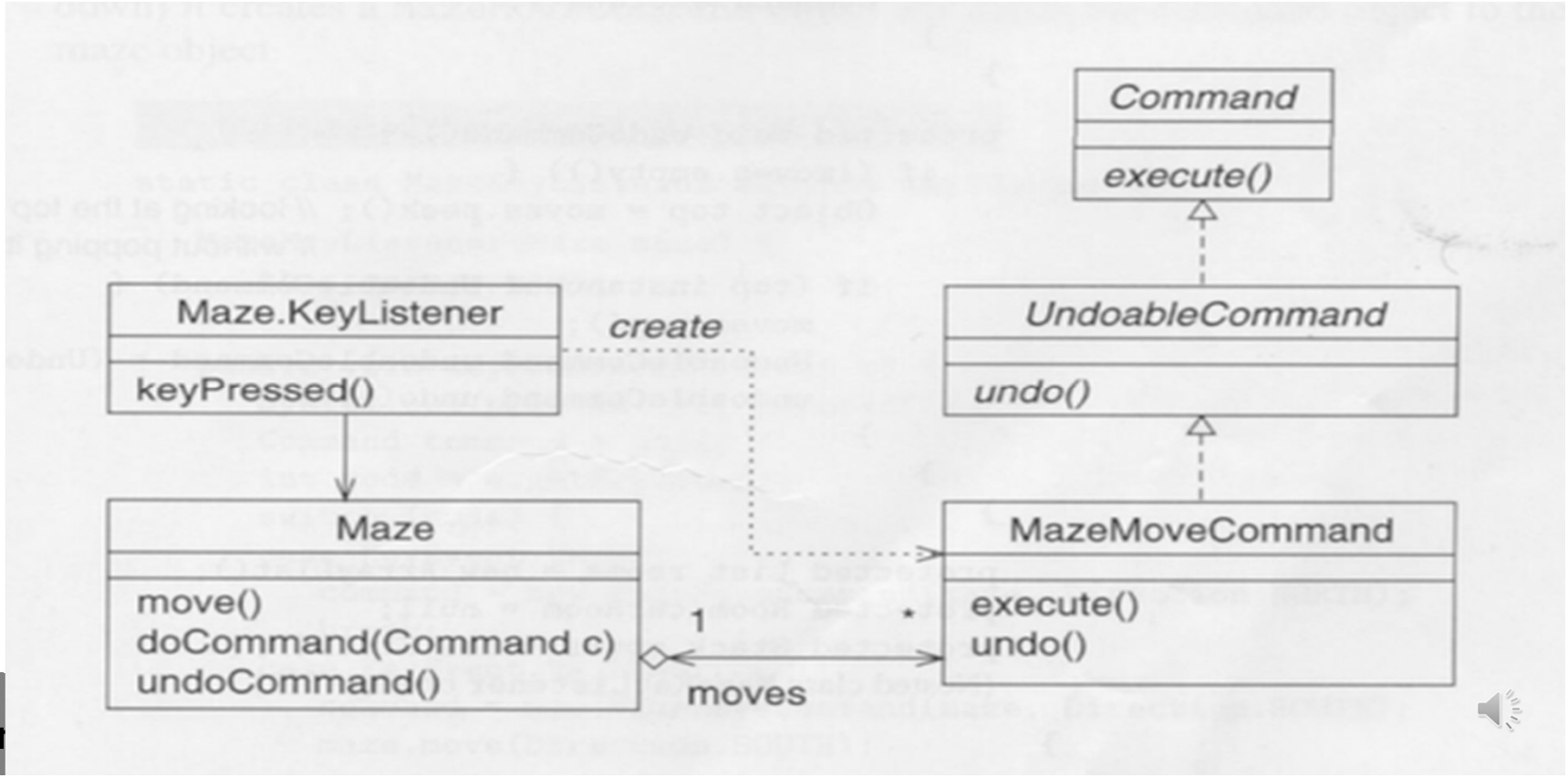
KeyPressed.

The participants of the Command design pattern

- *Command* (e.g., Command, UndoableCommand), which defines an interface to perform or undo an action.
- *Receiver* (e.g., Maze), which knows how to perform the actions.
- *ConcreteCommand* (e.g., MazeMoveCommand), which implements the *Command* interface and delegates the execution of the action to the *Receiver*.
- *Client* (e.g., Maze.MazeKeyListener), which creates the concrete commands and binds the concrete commands to their receivers.
- *Invoker* (e.g., Maze), which asks the command to carry out the action.

10.3.2 Supporting Undo

- Fig 10.8 : undoable moves in the maze game using the command pattern.



- The Command Interfaces : defines the basic interface of command objects.
  - execute() method : performs the action represented by the command object
  - undo() method : undoes the action performed by the execute() method

```
package maze;

public interface Command {
  public void execute();

}
```

```
package maze;

public interface UndoableCommand extends Command {
  public void undo();

}
```

The Receiver and Invoker of Command

- the receiver in the Command pattern
  - the class that is responsible for actually carrying out the actions represented by the command objects.
- The invoker
  - the class that invokes the command
  - has the action of the command object carried out
- in the maze game
  - the role of the receiver and the invoker are played by the same class : Maze
  - The move() method : the responsibility of the receiver
    - it attempted to move the player in the specified direction.
    - if there is an open door in the specified direction, the player enters the room on the other side of the door. Otherwise, the player stays in the current room.

The Receiver and Invoker of Command

- the doCOmmand() and undoCommand() method
  - fulfil the responsibility of the invoker
  - the doCommand() : execute the command and saves the command in the stack named moves for possible undoing of the command.
  - the undoCOmmand() method : pops the command at the top of the moves stack and attempts to undo the command.

The Receiver and Invoker of Command

```
//Class maze.Maze

public class Maze implements Cloneable {

  (Methods for building the maze board. See Section 10.2.1[p.475])

  public void move(Direction direction) {
    if (curRoom != null) {
      MapSite side = curRoom.getSide(direction);
      if (side != null) {
    side.enter(this);
      }
    }
  }

  protected void doCommand(Command command) {
    if (command != null) {
      moves.push(command);
      command.execute();
    }
  }

  protected void undoCommand() {
    if (!moves.empty()) {
      Object top = moves.peek(); // looking at the top element without popping it
      if (top instanceof UndoableCommand) {
    moves.pop();
    UndoableCommand undoableCommand = (UndoableCommand) top;
    undoableCommand.undo();
      }
    }
  }

  protected List rooms = new ArrayList();
  protected Room curRoom = null;
  protected Stack moves = new Stack();

  (Nested class MazeKeyListener on page 511)

}
```

```
package maze;

import java.util.Stack;
import java.util.List;
import java.util.ArrayList;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Maze implements Cloneable {

  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }

  public void addRoom(Room room) {
    if (room != null) {
      rooms.add(room);
    }
  }
```

```
public Room findRoom(int roomNumber) {
   for (int i = 0; i < rooms.size(); i++) {
     Room room = (Room) rooms.get(i);
     if (roomNumber == room.getRoomNumber()) {
            return room;
     }
   }
   return null;
 }

 public void setCurrentRoom(int roomNumber) {
   Room room = findRoom(roomNumber);
   setCurrentRoom(room);
 }

 public void setCurrentRoom(Room room) {
   if (room != curRoom) {
     if (curRoom != null) {
            curRoom.setInRoom(false);
     }
     if (room != null) {
            room.setInRoom(true);
            curRoom = room;
     }
     if (view != null) {
            view.repaint();
     }
   }
 }
```

```
public Room getCurrentRoom() {
  return curRoom;
}

// the responsibility of the receiver
//     it attempted to move the player in the specified direction.
//     if there is an open door in the specified direction,
//     the player enters the  room on the other side of the door.
//     Otherwise, the player stays in the current room.

public void move(Direction direction) {
  if (curRoom != null) {
    MapSite side = curRoom.getSide(direction);
    if (side != null) {
          side.enter(this);
    }
  }
}
```

```
public void draw(Graphics g) {
if (dim == null) {
  calculateDimension();
}
int dx = MARGIN + -offset.x * ROOM_SIZE;
int dy = MARGIN + -offset.y * ROOM_SIZE;

if (debug) {
  System.out.println("Maze.Draw(): offset=" + offset.x + ", " + offset.y);
}

// draw rooms first
for (int i = 0; i < rooms.size(); i++) {
  Room room = (Room) rooms.get(i);
  if (room != null) {
          Point location = room.getLocation();
          if (location != null) {

            if (debug) {
              System.out.println("Maze.Draw(): Room " + room.getRoomNumber() +
                                  " location: " + location.x + ", " + location.y);
            }

            room.draw(g,
                        dx + location.x * ROOM_SIZE,
                        dy + location.y * ROOM_SIZE,
                        ROOM_SIZE, ROOM_SIZE);
          }
  }
}
```

```
// draw walls and doors
for (int i = 0; i < rooms.size(); i++) {
  Room room = (Room) rooms.get(i);
  if (room != null) {
            Point location = room.getLocation();
            if (location != null) {
              for (Direction dir = Direction.first(); dir != null; dir = dir.next()) {
                MapSite side = room.getSide(dir);
                if (side != null) {
                  if (dir == Direction.NORTH) {
                            side.draw(g,
                                      dx + location.x * ROOM_SIZE - WALL_THICKNESS / 2,
                                      dy + location.y * ROOM_SIZE - WALL_THICKNESS / 2,
                                      ROOM_SIZE + WALL_THICKNESS,
                                      WALL_THICKNESS);
                  } else if (dir == Direction.EAST) {
                            side.draw(g,
                                      dx + location.x * ROOM_SIZE + ROOM_SIZE - WALL_THICKNESS / 2,
                                      dy + location.y * ROOM_SIZE - WALL_THICKNESS / 2,
                                      WALL_THICKNESS,
                                      ROOM_SIZE + WALL_THICKNESS);
                  } else if (dir == Direction.SOUTH) {
                            side.draw(g,
                                      dx + location.x * ROOM_SIZE - WALL_THICKNESS / 2,
                                      dy + location.y * ROOM_SIZE + ROOM_SIZE - WALL_THICKNESS / 2,
                                      ROOM_SIZE + WALL_THICKNESS,
                                      WALL_THICKNESS);
                  } else {
                            side.draw(g,
                                      dx + location.x * ROOM_SIZE - WALL_THICKNESS / 2,
                                      dy + location.y * ROOM_SIZE - WALL_THICKNESS / 2,
                                      WALL_THICKNESS,
                                      ROOM_SIZE + WALL_THICKNESS);
                  }
                }
              }
            }
  }
}
```

```
public Dimension getDimension() {
  if (dim == null) {
    calculateDimension();
  }
  return dim;
}

protected void calculateDimension() {
  if (rooms.size() > 0) {
    int minX = 0, maxX = 0, minY = 0, maxY = 0;
    Room room = (Room) rooms.get(0);
    room.setLocation(new Point(0, 0));
    boolean changed = true;
    while (changed &&
                !isAllRoomsSet()) {
              changed = false;
              for (int i = 0; i < rooms.size(); i++) {
                room = (Room) rooms.get(i);
                Point location = room.getLocation();
                if (location != null) {
                  for (Direction dir = Direction.first(); dir != null; dir = dir.next()) {
                    MapSite side = room.getSide(dir);
                    if (side instanceof Door) {
                              Door door = (Door) side;
                              Room otherSide = door.otherSideFrom(room);
                              if (otherSide != null &&
                                  otherSide.getLocation() == null) {
                                if (dir == Direction.NORTH) {
                                  otherSide.setLocation(new Point(location.x, location.y - 1));
                                  minY = Math.min(minY, location.y - 1);
```

```
                    } else if (dir == Direction.EAST) {
                      otherSide.setLocation(new Point(location.x + 1, location.y));
                      maxX = Math.max(maxX, location.x + 1);
                    } else if (dir == Direction.SOUTH) {
                      otherSide.setLocation(new Point(location.x, location.y + 1));
                      maxY = Math.max(maxY, location.y + 1);
                    } else {
                      otherSide.setLocation(new Point(location.x - 1, location.y));
                      minX = Math.min(minX, location.x - 1);
                    }
                    changed = true;
                  }
                }
              }
            }
          }
        }
        offset = new Point(minX, minY);
        dim = new Dimension(maxX - minX + 1, maxY - minY + 1);
      } else {
        offset = new Point(0, 0);
        dim = new Dimension(0, 0);
      }
    }
```

```
protected boolean isAllRoomsSet() {
    for (int i = 0; i < rooms.size(); i++) {
      Room room = (Room) rooms.get(i);
      if (room.getLocation() == null) {
            return false;
      }
    }
    return true;
  }
  protected void setView(Component view) {
    this.view = view;
  }
// fulfil the responsibility of the invoker
// the doCommand()
//    : execute the command and saves the command
//      in the stack named moves for possible undoing of the command.
  protected void doCommand(Command command) {
    if (command != null) {
      moves.push(command);
      command.execute();   /********* use method of interface ****/
    }
  }
```

```
  //  fulfil the responsibility of the invoker
  // the undoCOmmand() method :
  //       pops the command at the top of the moves stack and
  //       attempts to undo the command.
  protected void undoCommand() {
    if (!moves.empty()) {
      Object top = moves.peek();
                                  // looking at the top element without popping it
      if (top instanceof UndoableCommand) {
            moves.pop();
            UndoableCommand undoableCommand = (UndoableCommand)
top;
            undoableCommand.undo(); /********* use method of interface ****/
      }
    }
  }
  protected List rooms = new ArrayList();
  protected Dimension dim;
  protected Point offset;
  protected Room curRoom = null;
  protected Stack moves = new Stack();

  protected Component view;
```

```
private static final int ROOM_SIZE = 40;
private static final int WALL_THICKNESS = 6;
private static final int MARGIN = 20;

private static final boolean debug = true;

protected void showFrame(String frameTitle) {
  JFrame frame;
  frame = new JFrame(frameTitle);
  frame.setContentPane(new Maze.MazePanel(this));
  frame.pack();
  Dimension frameDim = frame.getSize();
  Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
  frame.setLocation(screenSize.width / 2 - frameDim.width / 2,
                    screenSize.height / 2 - frameDim.height / 2);
  frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
  frame.setVisible(true);
}
```

```
public static class MazePanel extends JPanel {

public MazePanel(Maze maze) {
  this.maze = maze;
  if (maze != null) {
        maze.setView(this);
        Dimension d = maze.getDimension();
        if (d != null) {
          dim = new Dimension(d.width * ROOM_SIZE + 2 * MARGIN,
                              d.height * ROOM_SIZE + 2 * MARGIN);
        }
        addKeyListener(new MazeKeyListener(maze));
  }
}

public void paint(Graphics g) {
  Dimension d = getSize();
  g.setColor(Color.white);
  g.fillRect(0, 0, d.width, d.height);
  if (maze != null) {
        maze.draw(g);
  }
  requestFocus();
}
```

```
// public boolean isFocusTraversable() { // pre 1.4
  public boolean isFocusable() { // 1.4
    return true;
  }

  public Dimension getPreferredSize() {
    return dim;
  }

  public Dimension getMinimumSize() {
    return dim;
  }

  private Maze maze;
  private Dimension dim;

}
```

```
// Client (e.g., Maze.MazeKeyListener),
// which creates the concrete commands
// and binds the concrete commands to their receivers.

    static class MazeKeyListener extends KeyAdapter {

    MazeKeyListener(Maze maze) {
      this.maze = maze;
    }

    public void keyPressed(KeyEvent e) {
      System.out.println("Key pressed");
      Command command = null;
      int code = e.getKeyCode();
      switch (code) {
      case KeyEvent.VK_UP:
            System.out.println("Up key");
            command = new MazeMoveCommand(maze, Direction.NORTH);
            break;
      case KeyEvent.VK_DOWN:
            System.out.println("Down key");
            command = new MazeMoveCommand(maze, Direction.SOUTH);
            // maze.move(Direction.SOUTH);  //???????????
            break;
```

```
case KeyEvent.VK_LEFT:
            System.out.println("Left key");
            command = new MazeMoveCommand(maze, Direction.WEST);
            break;
      case KeyEvent.VK_RIGHT:
            System.out.println("Right key");
            command = new MazeMoveCommand(maze, Direction.EAST);
            break;
      default:
            System.out.println("Key press ignored");
      }
      if (command != null) {
            maze.doCommand(command);
      }
    }

    Maze maze;
  }

}
```

- The Concrete Commands
  - the concrete commands in the maze game : is the MazeMoveCommand class
    - implements the UndoableCommand interface
    - execute() method : attempts to move the player in the specified direction
    - undo() method : attempts to move the player in the direction opposite to the specified direction.

The Concrete Command

```
// ConcreteCommand (e.g., MazeMoveCommand),
//which implements the Command interface
// and delegates the execution of the action to the Receiver.

package maze;

public class MazeMoveCommand implements UndoableCommand {

  public MazeMoveCommand(Maze maze, Direction direction) {
    this.maze = maze;
    this.direction = direction;
  }
  public void execute() {
    maze.move(direction); // delegates the execution of the action to the
Receiver.
  }
  public void undo() {
    maze.move(direction.opposite());
  }
  protected Maze maze;
  protected Direction direction;

}
```

- The Client of Command
  - the class that is responsible for creating the command objects
  - Maze.MazeKeyListener
    - listens for key strokes.
    - For the arrow keys(left, right, up, down) it create a MazeMoveCommand Object and sends the command object to the Maze Object.

■ The Client of Command

```
// Nested class of maze.Maze : MazeKeyListener
//  the class that is responsible for creating the command objects

static class MazeKeyListener extends KeyAdapter {
 MazeKeyListener(Maze maze) {
   this.maze = maze;
 }

 public void keyPressed(KeyEvent e) {
   System.out.println("Key pressed");
   Command command = null;
   int code = e.getKeyCode();
   switch (code) {
   case KeyEvent.VK_UP:
         System.out.println("Up key");
         command = new MazeMoveCommand(maze, Direction.NORTH);
         break;
   case KeyEvent.VK_DOWN:
         System.out.println("Down key");
         command = new MazeMoveCommand(maze, Direction.SOUTH);
         // maze.move(Direction.SOUTH);  //???????????
         break;
```

```
case KeyEvent.VK_LEFT:
            System.out.println("Left key");
            command = new MazeMoveCommand(maze, Direction.WEST);
            break;
      case KeyEvent.VK_RIGHT:
            System.out.println("Right key");
            command = new MazeMoveCommand(maze, Direction.EAST);
            break;
      default:
            System.out.println("Key press ignored");
      }
      if (command != null) {
            maze.doCommand(command);
      }
    }

    Maze maze;
  }
```

- The Undoable Maze Game
  - maze.UndoableMazeGame Class
    - the main class of the Maze game that supports undo commands
    - main() method : similar to the main() method of the MazeGameBuilder class, except that is also builds a menu bar that contains an undo menu item.
    - The undo command can be invoked from the undo menu items.

- The Undoable Maze Game
  - (Page 512) Class maze.UndoableMazeGame

```
package maze;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class UndoableMazeGame {

  public static void main(String[] args) {
    Maze maze;
    MazeBuilder builder;
    MazeFactory factory = null;

    if (args.length > 0) {
      if ("Harry".equals(args[0])) {
            factory = new maze.harry.HarryPotterMazeFactory();
      } else if ("Snow".equals(args[0])) {
            factory = new maze.snow.SnowWhiteMazeFactory();
      } else if ("Default".equals(args[0])) {
            factory = new MazeFactory();
      }
    }
```

```
if (factory != null) {
      builder = new FactoryMazeBuilder(factory);
    } else {
      builder = new SimpleMazeBuilder();
    }
    maze = MazeGameBuilder.createMaze(builder);
    maze.setCurrentRoom(1);

    JMenuBar menubar = new JMenuBar();
    JMenu menu = new JMenu("Command");
    JMenuItem undoMenuItem = new JMenuItem("undo");
    undoMenuItem.addActionListener(new MazeCommandAction(maze));
    menu.add(undoMenuItem);
    menubar.add(menu);

    JFrame frame;
    frame = new JFrame("Maze -- Builder");
    frame.getContentPane().setLayout(new BorderLayout());
    frame.getContentPane().add(menubar, BorderLayout.NORTH);
    frame.getContentPane().add(new Maze.MazePanel(maze), BorderLayout.CENTER);
    frame.pack();
    Dimension frameDim = frame.getSize();
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    frame.setLocation(screenSize.width / 2 - frameDim.width / 2,
                      screenSize.height / 2 - frameDim.height / 2);
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
```

```
static class MazeCommandAction implements ActionListener {

    public MazeCommandAction(Maze maze) {
      this.maze = maze;
    }

    public void actionPerformed(ActionEvent event) {
      maze.undoCommand();   // Client
    }

    protected Maze maze;
  }

}
```

```
C:¥Chapter10>java maze.UndoableMazeGame Harry
Maze.Draw(): offset=-2, -2
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: -1, 0
Maze.Draw(): Room 3 location: -2, 0
Maze.Draw(): Room 4 location: 0, -1
Maze.Draw(): Room 5 location: -1, -1
Maze.Draw(): Room 6 location: -2, -1
Maze.Draw(): Room 7 location: 0, -2
Maze.Draw(): Room 8 location: -1, -2
Maze.Draw(): Room 9 location: -2, -2
Key pressed
Up key
Maze.Draw(): offset=-2, -2
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: -1, 0
Maze.Draw(): Room 3 location: -2, 0
Maze.Draw(): Room 4 location: 0, -1
Maze.Draw(): Room 5 location: -1, -1
Maze.Draw(): Room 6 location: -2, -1
Maze.Draw(): Room 7 location: 0, -2
Maze.Draw(): Room 8 location: -1, -2
Maze.Draw(): Room 9 location: -2, -2
Key pressed
Left key
```