7.9 *Case Study and Lab:* Rolling Dice

In this section we will implement the *Craps* program. Craps is a game played with dice. In Craps, each die is a cube with numbers from 1 to 6 on its faces. The numbers are usually represented by dots (Figure 7-1).

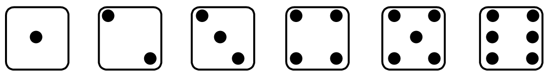


Figure 7-1. Dots configuration on a die

A player rolls two dice and adds the numbers of dots shown on them. If the total is 7 or 11, the player wins; if the total is 2, 3 or 12, the player loses. If the total is anything else, the player has to roll again. The total, called the "point," is remembered, and the objective now is to roll the same total as the "point." The player keeps rolling until he gets either "point" or 7. If he rolls "point" first, he wins, but if he rolls a 7 first, he loses. You can see why this game was chosen as a lab for `if-else` statements!

Our team has been asked to design and code a *Craps* program for our company's "Casino Night" charitable event. Three people will be working on this project. I am the project leader, responsible for the overall design and dividing the work between us. I will also help team members with detailed design and work on my own piece of code. The second person, Aisha, is a consultant; she specializes in GUI design and implementation.

The third person is you!

Run the executable *Craps* program by clicking in the `Craps.jar` file in `JM\Ch07\Craps`. When you click on the "Roll" button, red dice start rolling on a green "table." When they stop, the score is updated or the "point" is shown on the display panel (Figure 7-2). The program allows you to play as many games as you want.

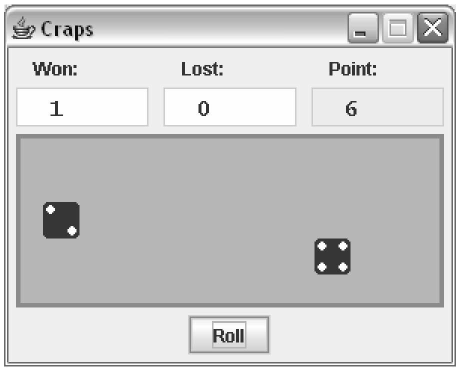


Figure 7-2. The *Craps* program

We begin the design phase by discussing which objects are needed for this application. One approach may be to try making objects in the program represent objects from the real world. Unfortunately, it is not always clear what exactly is a "real world" object. Some objects may simulate tangible machines or mechanisms, others may exist only in "cyberspace," and still others may be quite abstract and exist only in the designer's imagination.

Here we need one object that represents the program's window. Let us call this object `window` and its class `Craps`. As usual, we will derive this class from the `JFrame` class in Java's *Swing* package. The window (Figure 7-2) is divided into three "panels." The top panel displays the score and the current state of the game. Let's call it `display` and its class `DisplayPanel`. The middle panel represents the Craps table where the dice roll. Let's call it `table` and its class `CrapsTable`. The bottom panel holds the "Roll" button. Let's call it `controls` and its class `ControlPanel`. The control panel can also handle the "Roll" button's click events.

It makes sense that each of the `DisplayPanel`, the `CrapsTable`, and the `ControlPanel` classes extend the Java library class `JPanel`. For example:

```
public class DisplayPanel extends JPanel
{
  ...
}
```

The `table` object shows two "rolling dice," so we need a class that will represent a rolling die. Let's call it `RollingDie`.

These five classes, `Craps`, `DisplayPanel`, `CrapsTable`, `ControlPanel`, and `RollingDie`, form the GUI part of our *Craps* program (Figure 7-3).

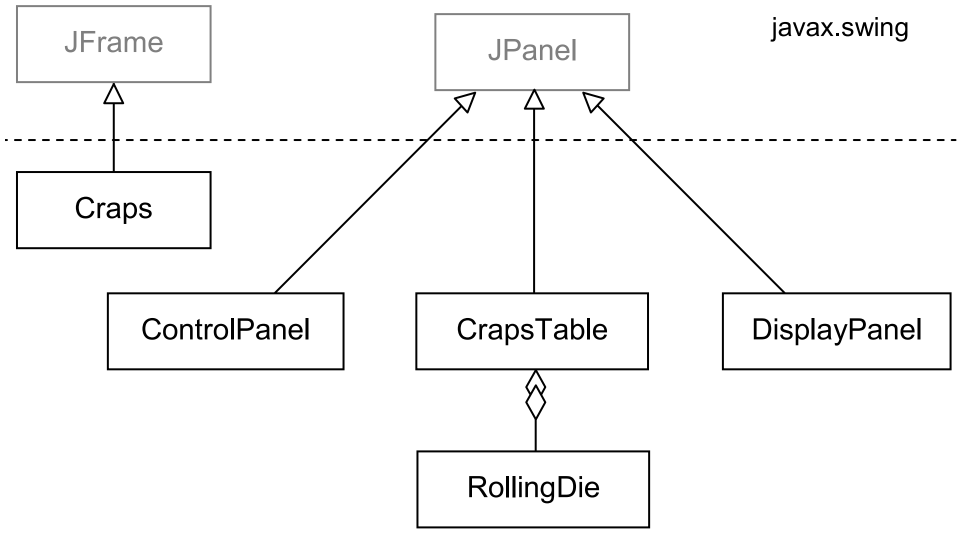
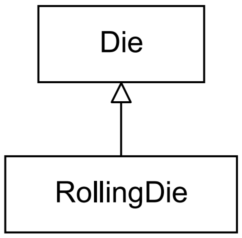


Figure 7-3. GUI classes in the *Craps* program

It makes sense to me to split the code for the visible and "numeric" aspects of a rolling die into two classes. The base class `Die` will represent a die as an abstract device that generates a random integer in the range from 1 to 6. The class `RollingDie` will extend `Die`, adding methods for moving and drawing the die:



My rationale for this design decision is that we might reuse the `Die` class in another program, but the dice there might have a different appearance (or may remain invisible).

Last but not least, we need an "object" that will represent the logic and rules of *Craps*. This is a "conceptual" object, not something that can be touched. Of course that won't prevent us from implementing it in Java. Let's call this object `game` and its class `CrapsGame`. The `CrapsGame` class won't be derived from anything (except the default, `Object`), won't use any Java packages, and won't process any events.

There are many good reasons for separating the rules of the game from the GUI part. First, we might need to change the GUI (if our boss doesn't like its "look and feel") while leaving the game alone. Second, we can reuse the `CrapsGame` class in other applications. For example, we might use it in a statistical simulation of Craps that runs through the game many times quickly and doesn't need a fancy GUI at all. Third, we might have a future need for a program that implements a similar-looking dice game but with different rules. Fourth, Aisha and I know only the general concept of the game and are not really interested in learning the details. And finally, it is a natural division of labor. We have a beginner on our team (you) and we have to give you a manageable piece of work.

❖ ❖ ❖

Now we need to decide how the objects interact with each other. Figure 7-4 shows the overall design for the *Craps* program that I have come up with.

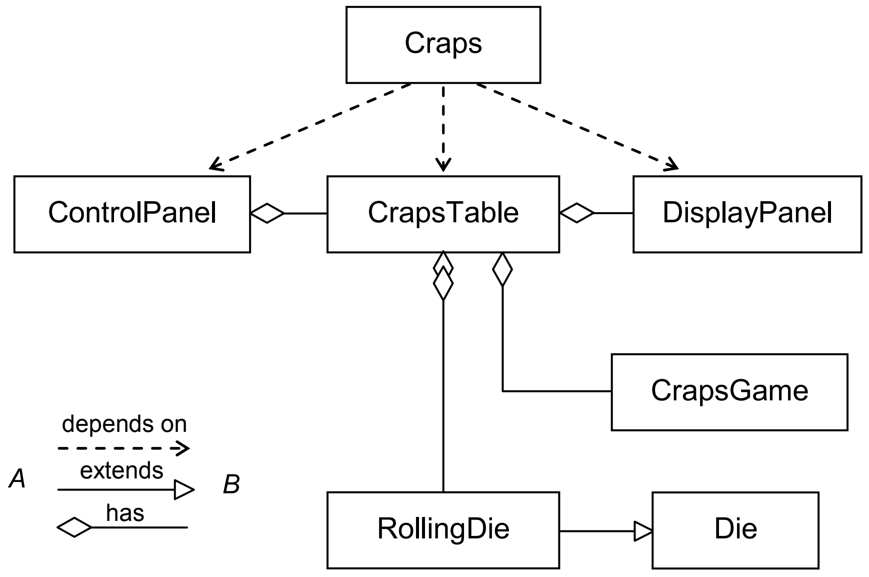


Figure 7-4. *Craps* classes and their relationships

There is no unique way, of course, of designing an application — a designer has a lot of freedom. But it is very helpful to follow some established *design patterns* and tested practices. We will talk about design patterns in Chapter 26. Here we want to emphasize two principles of sound design.

First, each class must represent a single concept, and all its constructors and public methods should be related to that concept. This principle is called *cohesion*.

In a good design, classes are cohesive.

Second, dependencies between classes should be minimized. The reason we can draw a class diagram in Figure 7-4 without lines crisscrossing each other in all directions is that not all the classes depend on each other. OO designers use the term *coupling* to describe the degree of dependency between classes.

In a good design, coupling should be minimized.

It is good when a class interacts with only few other classes and knows as little about them as possible. Low coupling makes it easier to split the work between programmers and to make changes to the code.

In our *Craps* program, for example, the ControlPanel class and the DisplayPanel class do not need to know about each other's existence at all. ControlPanel knows something about CrapsTable — after all, it needs to know what it controls. But ControlPanel knows about only a couple of simple methods from CrapsTable.

A reference to a CrapsTable object is passed to ControlPanel's constructor, which saves it in its field table. The ControlPanel object calls table's methods when the "roll" button is clicked:

```
// Called when the roll button is clicked
public void actionPerformed(ActionEvent e)
{
  if (!table.diceAreRolling())  // if dice are not rolling,
    table.rollDice();           //   start a new roll
}
```

Likewise, table has a reference to display, but it knows about only one of its methods, update. When the dice stop rolling, table consults game (the only class that knows the rules of the game) about the result of the roll and passes that result (and the resulting value of "point") to DisplayPanel's update method:

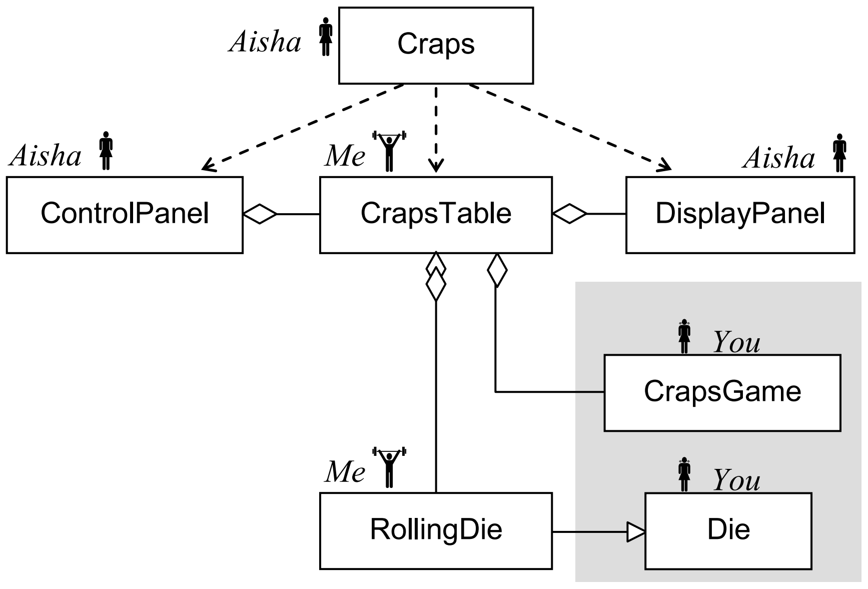


Figure 7-5. Task assignments in the *Craps* project

❖ ❖ ❖

I will call these methods from `RollingDie`. The `roll` method simulates one roll of a die. It obtains a random integer in the range from 1 to 6 and saves it in a field. The `getNumDots` method returns the saved value from that field. Do not define any constructors in `Die`: the default no-args constructor will do. To get a random number, use a call to `Math.random()`. This method returns a "random" `double` *x*, such that $0 \le x < 1$. Scale that number appropriately, then truncate it to an integer.

Now the `CrapsGame` class. My `CrapsTable` object creates a `CrapsGame` object called `game`:

```
private CrapsGame game;
...

// Constructor
public CrapsTable(DisplayPanel displ)
{
  ...
  game = new CrapsGame();
  ...
}
```

Again, no need to define a constructor in `CrapsGame`: we will rely on the default no-args constructor.

My `CrapsTable` object calls `game`'s methods:

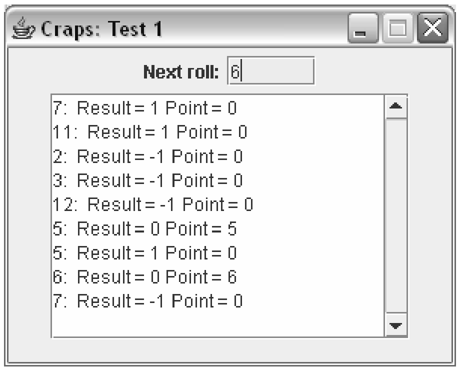
```
        int result = game.processRoll(total);
        int point = game.getPoint();
```

The `processRoll` method takes one `int` parameter — the sum of the dots on the two dice. `processRoll` should process that information and return the result of the roll: 1 if the player wins, -1 if he loses, and 0 if the game continues. In the latter case, the value of "point" is set equal to `total`. Define a private `int` field `point` to hold that value. If the current game is over, `point` should be set to 0. `getPoint` is an accessor method in your `CrapsGame` class. It lets me get the value of `point`, so that I can pass it on to `display`.

❖ ❖ ❖

We are ready to start the work. The only problem is the time frame. Aisha's completion date is unpredictable: she is very busy, but once she gets to work she works very fast. My task can be rather time-consuming. I will try to arrange a field trip to Las Vegas to film some video footage of rolling dice. But most likely our boss won't approve that, and I'll have to settle for observing rolling dice on the carpet in my office. Meanwhile you are anxious to start your part.

Fortunately, Aisha has found an old test program to which you can feed integers as input. She added a few lines to make it call `processRoll` and `getPoint` and display their return values (Figure 7-6). She called her temporary class `CrapsTest1`. Now you don't have to wait for us: you can implement and test your `CrapsGame` class independently. You won't see any dice rolling for now, but you will be able to test your class thoroughly in a predictable setting.



```
7:  Result = 1 Point = 0
11:  Result = 1 Point = 0
2:  Result = -1 Point = 0
3:  Result = -1 Point = 0
12:  Result = -1 Point = 0
5:  Result = 0 Point = 5
5:  Result = 1 Point = 0
6:  Result = 0 Point = 6
7:  Result = -1 Point = 0
```

Figure 7-6. The preliminary program for testing `CrapsGame`

1. Copy `CrapsTest1.java` and `CrapsGame.java` from `JM\Ch07\Craps` to your work folder. Fill in the blanks in the `CrapsGame` class, compile it, combine it with the `CrapsTest1` class into a program, and test it thoroughly.

2. Write the `Die` class and a small console application to test it by printing out the results of several "rolls." For example:

```
public static void main(String[] args)
{
  Die die = new Die();
  die.roll();
  System.out.println(die.getNumDots());
  die.roll();
  ...
}
```

3. After you get the `CrapsGame` and `Die` classes to work, test them with the *CrapsStats* application, which quickly runs the game multiple times and counts the number of wins. You will find `CrapsStats.java` in `JM\Ch07\Craps`. Note how we reuse for this task the `CrapsGame` and the `Die` classes that you have written for a different program.

   Compare your simulation result with the theoretical probability of winning in *Craps,* which is 244/495, or about 0.493. If you run 10,000 trial games, the number of wins should be somewhere between 4830 and 5030.

7.12 *Case Study and Lab:* Rolling Dice Continued

By this time you have finished your `CrapsGame` and `Die` classes and Aisha has found the time to put together her GUI classes. I myself have gotten bogged down with my `CrapsTable` and `RollingDie` classes, trying to perfect the animation effects. Meanwhile, not to stall Aisha's testing, I have written a *stub class* `CrapsTable` (Figure 7-7) to provide a temporary substitute for the actual class I am working on. A stub class has very simple versions of methods needed for testing other classes. This is a common technique when a programmer needs to test a part of the project while other parts are not yet ready.

```
public class CrapsTable
{
  private DisplayPanel display;
  private CrapsGame game;
  private Die die1, die2;

  // Constructor
  public CrapsTable(DisplayPanel displ)
  {
    display = displ;
    game = new CrapsGame();
  }

  // Rolls the dice
  public void rollDice()
  {
    die1.roll();
    die2.roll();
    int total = die1.getNumDots() + die2.getNumDots();
    int result = game.processRoll(pts);
    int point = game.getPoint();
    display.update(result, point);
  }

  public boolean diceAreRolling()
  {
    return false;
  }
```

Figure 7-7. Temporary "stub" class `CrapsTable.java`

My stub class includes a temporary version of the `rollDice` method that simply calls `game`'s `processRoll` method with a random sum of points and a version of `diceAreRolling` that always returns `false`.

You're certainly welcome to take a look at Aisha's GUI implementation (in `JM\Ch07\Craps\Source.zip`), but no one has time right now to explain to you how it works.

❖ ❖ ❖

Since you are done with your part, I thought you could help me out with my `RollingDie` class. I've made a lot of progress on it, but a couple of details remain unfinished.

I have coded the constructor, the `roll` method that starts the die rolling, and the `avoidCollision` method that keeps one die from overlapping with another. I have also provided the `boolean` method `isRolling`, which tells whether my die is moving or not. But I am still working on drawing a rolling and a stopped die. I took what is called *top-down* approach with *step-wise* refinement, moving from more general to more specific tasks. First I coded the `draw` method in general terms:

```
// Draws this die, rolling or stopped;
// also moves this die, when rolling
public void draw(Graphics g)
{
  if (xCenter < 0 || yCenter < 0)
    return;
  else if (isRolling())
  {
    move();
    drawRolling(g);
    xSpeed *= slowdown;
    ySpeed *= slowdown;
  }
  else
  {
    drawStopped(g);
  }
}
```

Note how I used the `if-else-if` structure to process three situations: my die is off the table, it is still moving, or it is stopped.

My `draw` method calls the more specialized methods `drawRolling` and `drawStopped`. I am still working on these, but I know that each of them will call an even lower-level method `drawDots` that will draw white dots on my die:

```
// Draws this die when rolling with a random number of dots
private void drawRolling(Graphics g)
{
  ...
  Die die = new Die();
  die.roll();
  drawDots(g, x, y, die.getNumDots());
}

// Draws this die when stopped
private void drawStopped(Graphics g)
{
  ...
  drawDots(g, x, y, getNumDots());
}
```

I have started `drawDots` (Figure 7-8) and am counting on you to finish it. (Naturally, it involves a `switch` statement.) Meanwhile I will finish `CrapsTable`, and we should be able to put it all together.

```
// Draws a given number of dots on this die
private void drawDots(Graphics g, int x, int y, int numDots)
{
  g.setColor(Color.WHITE);

  int dotSize = dieSize / 4;
  int step = dieSize / 8;
  int x1 = x + step - 1;
  int x2 = x + 3*step;
  int x3 = x + 5*step + 1;
  int y1 = y + step - 1;
  int y2 = y + 3*step;
  int y3 = y + 5*step + 1;

  switch (numDots)
  {
    case 1:
      g.fillOval(x2, y2, dotSize, dotSize);
      break;

    < missing code >

  }
}
```

Figure 7-8. A fragment from `JM\Ch07\Craps\RollingDie.java`

Copy `RollingDie.java`from `JM\Ch07\Craps` into your work folder and fill in the blanks in its `drawDots` method. (Figure 7-1 shows the desired configurations of dots on a die.) Collect all the files for the *Craps* program together: `Craps.jar` (from `JM\Ch07\Craps`); `CrapsGame.java` and `die.java` (your solutions from the lab in Section 7.9); and `RollingDie.java`. Compile them, and run the program.

7.13 Summary

The general form of a *conditional statement* in Java is:

```
if (condition)
{
  statementA1;
  statementA2;
  ...
}
else
{
  statementB1;
  statementB2;
  ...
}
```

`condition` may be any Boolean expression.

Conditions are often written with the *relational operators*

`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`!=`	not equal to

and the *logical operators*

`&&`	and
`||`	or
`!`	not