10-1 In Web pages, tables consist of rows, which in turn consist of cells. The actual width

and height of each cell is computed based in part on its content (e.g., the amount of text

in the cell, the size of an image in the cell), and the height of a row is the maximum of

the heights of all cells in the row. Consequently, the final layout of a table in a Web

page can only be computed once the content of each cell has been retrieved from the

Internet. Using the proxy pattern described in Figure 10-7, describe an object model

and an algorithm that would enable a Web browser to start displaying a table before the

size of all cells is known, possibly redrawing the table as the content of each cell is

downloaded.

Thinking of an object model that would allow a web browser to display a table before the content of cells is known can be done a few different ways. After reading the proxy model around figure 10-7 my first thought was that the content doesn’t need to be known to do this, the attributes of those particular cells, on the other hand, do need to be known. By providing attributes of the information that will fill the space, this takes up far less space/memory, and allows the design of a page to load, with the information being filled in as it becomes available. Take an image, for example, if you already know the size of the image, you can fill in a cell with blank space of the image size, then, once all the pixels are pulled from a disk to be displayed, you can replace the attribute info with the actual file information that should be displayed. Can complete something similar for text, by knowing the number of characters that need to be displayed, that space can be reserved in a cell and then filled in with the actual text once available.

10-6 Design a relational database schema for the object model of Figure 10-30. Assume

Leagues, Tournaments, Players, and Rounds have a name attribute and a unique

identifier. Additionally, Tournaments and Rounds have start and end date attributes.

When different transformations are available, explain the trade-off involved.

While this is a very basic layout, there is still plenty of other work that can and should be done to be more efficient and fit more with the normal forms. For example, there should be a separate table for start and end dates. There should be a better way to include leagues/teams that are involved in multiple tournaments. Should probably have different tables for each round to easier keep track of all the player/team information per round instead of requiring a query to see specific rounds. As far as other transformation and trade-offs maybe I didn’t fully understand the question?

11-1 Correct the faults in the isLeapYear() and getNumDaysInMonth() methods of

Figure 11-11 and generate test cases using the path testing method. Are the test cases

you found different than those of Table 11-4 and Figure 11-13? Why? Would the test

cases you found uncover the faults you corrected?

The fault in the isLeapYear() function is that it assumes that every leap year is divisible by only 4 which is not the case. You must meet 2 other conditions:

public static boolean isLeapYear(int year) {

boolean leap;

if ((year%400) == 0) {

leap = true;

} else if ((year%100) == 0) {

leap = false;

}else if ((year%4) == 0){

leap = true;

} else {

leap = false;

}

return leap;

}

For the getnumDaysInMonth() function there was a few things wrong, first of which it was saying months had 32 days and the other months were 30 days. Since month 2 is the only one affected by leap year this should not be the case. Another error was August was completely ignored as part of the equation. Corrected code:

public static int getNumDaysInMonth(int month, int year) throws MonthOutOfBounds, YearOutOfBounds {

int numDays;

if (year < 1) {

throw new YearOutOfBounds(year);

}

if (month < 1 || month > 12) {

throw new MonthOutOfBounds(month);

}

if (month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10 || month == 12) {

numDays = 31;

} else if (month == 4 || month == 6 || month == 9 || month == 11) {

numDays = 30;

} else if (month == 2) {

if (isLeapYear(year)) {

numDays = 29;

} else {

numDays = 28;

}

} else {

throw new MonthOutOfBounds(month);

}

return numDays;

}

Test cases:

|  |  |
| --- | --- |
| Test Case | Path |
| Year = 0, month = 1 | throw YearOutOfBound(year) |
| Year = 1901, month = 1 | numDays = 31 return |
| Year = 1901, month = 2 | numDays = 28 return |
| Year = 1901, month = 2 | numDays = 29 return |
| Year = 1901, month = 4 | numDays = 30 return |
| Year = 1901, month = 8 | numDays = 31 return |
| Year = 1901, month = 0 | throw MonthOutOfBound(month) |
| Year = 1901 | Leap = false return |
| Year = 1904 | Leap = true return |
| Year = 1700 | Leap = false return |
| Year = 2000 | Leap = true return |

These would uncover the faults I corrected because it includes month 8 which was not included on the initial release and the year 1700 returns false in this use case where as the original it would’ve returned true. The numDays returned is also correct, but without knowing the number of days in a month this could easily be missed in test cases.

11-2 Generate equivalent Java code for the state machine diagram for the SetTime use case

of 2Bwatch (Figure 11-14). Use equivalence testing, boundary testing, and path testing

to create test cases for the code you have just generated. How do these test cases

compare with those generated using state-based testing

class 2Bwatch {

String initial="Measuretime";

Define various methods of the use case 2Bwatch SetTime.

public void pressleftButton() {

initial="Measuretime";

System.out.println(initial);

}

public void pressBothButtons() {

initial="SetTime";

System.out.println(initial);

}

public void pressleftButton() {

initial="Measuretime";

System.out.println(initial);

}

public void batteryEmpty() {

nextstate="DeadBattery";

System.out.println(nextstate);

}

public void timeout() {

}

public static void main(String args[]) {

int choice;

2Bwatch m =new 2Bwatch();

System.out.println(m.initial);

System.out.println("Enter input of your choice range 1-8");

switch(choice) {

case 1:

System.out.println("Measuretime");

break;

case 2:

m.pressLeftButton();

m.pressRightButton();

break;

case 3:

m.pressBothButtons();

break;

case 4:

m.timeout();

break;

case 5:

m.pressBothButtons();

m.beep();

break;

case 6:

m.pressLeftButton();

m.pressRightButton();

break;

case 7:

m.batteryEmpty();

break;

case 8:

m.batteryEmpty();

break;

}

}

}

|  |  |  |
| --- | --- | --- |
| Test Case | Input | Expected Output |
| Equivalence Testing |  |  |
|  | Choice = 1 | MeasureTime |
|  | Choice = 3 | SetTime |
|  | Choice = 7 | DeadBattery |
| Boundary Testing |  |  |
|  | Choice = 1 | MeasureTime |
|  | Choice = 8 | DeadBattery |
|  | Choice = 0 | No output |
|  | Choice = 9 | No output |
| Path Testing |  |  |
|  | Choice = 1 | Measuretime |
|  | Choice = 2 | Measuretime <br> SetTime |
|  | Choice = 3 | SetTime |
|  | Choice = 4 | No Output |
|  | Choice = 5 | SetTime <br> Beep sound |
|  | Choice = 6 | Measuretime <br> SetTime |
|  | Choice = 7 | DeadBattery |
|  | Choice = 8 | DeadBattery |

With state-based testing tests the outcome of the system/code in it’s different, well, states. It validates specific output for different conditions. Whereas boundary testing checks the extremes of a program to make sure all outputs are handled correctly. Path testing goes down specific paths of the program to ensure output happens as expected. Finally, equivalence testing breaks the code/program/etc into sections to test them with the goal of reducing the number of test cases. The main difference of the 3 tests compared to state-based testing is that state based is focused more on output where equivalence, boundary, and path consider the input and the output. With their differences, if all are used together they would work together for an all-inclusive coverage of testing.