

Inft2012 Application Programming

Lecture 9

Testing, problem solving

How to use software specifications

- Before designing a program, read the specifications
- If the specifications are not clear – ask!
- Every time a new section is designed, read the specifications again
- Every time a new section is implemented, read the specifications again
- When the development process seems to be completed, read the specifications again
- Keep fixing the problems you've found until you've really finished, then read the specifications again
- Repeat the last two steps until the application actually does what the specifications say it should do

Program testing

- How do you know whether the program does what it's meant to?
- Program testing is an essential component of programming
- A program that produces wrong results is worse than a program that produces no results, because some users will believe the wrong results
- Here are some thoughts about program testing

3

Exhaustive testing

- Exhaustive testing means testing a program with every possible combination of inputs
- For example, a program is required to input an integer and output its square
- Exhaustive testing would mean trying the program with every possible integer input – and there are a lot of possible integers!
- Exhaustive testing is almost never possible
- Fortunately, there are other approaches that do the job with a great deal less effort

4

Boundary testing

- The idea of boundary testing is to test values on and close to the *boundaries*, the places where things change
- Imagine a vehicle control program that cuts in when the car starts, starts beeping when the speed exceeds 100km/h, and turns off the fuel supply when the speed exceeds 120km/h
- It would make sense to test this program at
 - 0km/h, 1km/h
 - 99km/h, 100km/h, 101km/h
 - 119km/h, 120km/h, 121km/h
- If it works correctly for these values (and perhaps one other), we *expect* it to work correctly for all values

5

Knowing what to expect

- So we try the program on our test input, and look at the output. How do we know whether it's right?
- We must know what output to expect for all our test inputs
- And you can only do this if you fully understand the problem
- For example, a program to calculate the areas of circles tells us that a circle of radius 2.75 has area 8.6428571
- Is it correct? (*Don't* use a calculating device to check!)
- As part of your analysis and design, you should determine the test data and the corresponding output that will show your program is working

6

Testing in manageable pieces

- A complete program is often too complex to test thoroughly
- On the other hand, it's often quite easy to test each method of the program
- The first phase of good testing is therefore to test each method *individually*
- The second phase is then to establish that all of the methods work properly *together* to solve the problem
- A second phase that just tests the *interactions* between methods is often less complex than a comprehensive test of the whole program

7

Black box testing

- Black box testing tests a program or method by recognising that different sets of data would expect different sorts of output
- It's also called functional testing, because it checks whether the program functions correctly
- It chooses sample data from each set, including boundary data, and tests with that data
- It regards the program or method as a 'black box'; we can see what goes in and what comes out, but not what goes on inside
- But if all the right outputs are produced from the test inputs, we don't need to know what goes on inside

8

White box testing

- White box testing (which should be called clear box testing) tests a program or method by ensuring that every line of code is executed
- It's also called structural testing, because it tests according to the known structure of the program
- It regards the program or method as a 'white box'; we can see what goes in and what comes out, and also what goes on inside; we are familiar with the logic and the code, and can see what path any input will take
- It chooses sample data (including boundary data) explicitly to ensure that each line of code is executed, and tests with that data

9

'Testing' by inspection

- Many programmers test their programs by inspection: they look at the code they've written, and it looks right, so the program must be correct
- Despite years of evidence to the contrary, they still seem to believe that this is satisfactory, and all that is required to 'test' a piece of software!
- While inspection can help to find errors, it is severely limited as a way of establishing that a program is correct
- However, there is some merit in having a program inspected by another programmer; other programmers are likely to see flaws that the writer is blind to

10

Testing by desk-checking

- Desk-checking is a useful way of checking what's going on at the detailed code level – so long as you write down what the code will actually do, not just what you think it will do
- Stepping with the Debug tool is like desk-checking, but is definitely doing what the code will actually do
- While these methods can be used to confirm that the program is doing what it should, they are more often used to figure out why it isn't doing what it should; that is, they are used once testing has uncovered a problem

11

Testing is no substitute for design

- “Testing does not increase quality; programming and design do.”
- “Testing just provides the insights that the team lacked to do a correct design and implementation.”

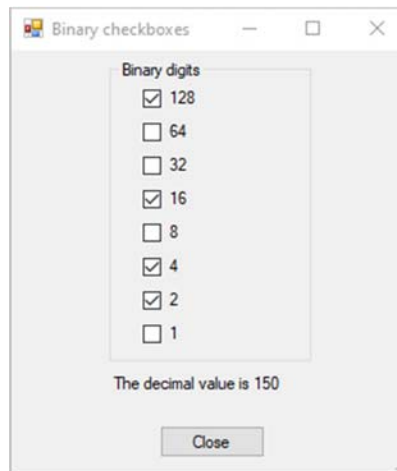
James O Coplien, Why most unit testing is waste.

<http://www.rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf>

12

Problem-solving technique

- Here is a general approach that might be valuable when tackling a programming problem
- The approach is presented on the following slides, as applied to an example – the binary digits calculator from a previous lab



13

Problem-solving technique

1. Identify the problem (be sure you know what is required)
 - Thoroughly read the specification and examine the sample screen
2. Identify resources required for its solution (input, output, storage space for calculations); some of these may not be known until the solution algorithm is produced
 - Inputs are specified: checkboxes and their values
 - Output is specified: a label
 - We're not yet sure of any storage requirements

14

Problem-solving technique

3. If possible, simplify the problem domain
 - Start with 3 checkboxes – simpler to implement and test
4. Set out the user view (what the user will see)
 - You'd normally do this on paper, but it's already presented as a screen shot
5. Create the user view
 - Create a form to implement your design
6. Try a pen and paper solution: use simple data and desk-checking techniques to try to work out a solution
 - Tick Chbx1 and Chbx4 (meaningful names representing the 1 bit and the 4 bit) and work out how to get the result
 - This involves adding the 'values' of all checked boxes

15

Problem-solving technique

7. Where appropriate, decompose the problem into simpler tasks
 - Write an event handler for a single checkbox, but still get it to check the values of all checkboxes
8. For each task in turn, create a solution algorithm (this may be for a simpler solution than will finally be required), code the simple solution (with good programming style) and test it
 - From step 6 we know to look at each checkbox in turn, and if it's checked, add its number to a running total – ah, that means we need storage for the running total
 - After checking all checkboxes, display the running total
 - Write this in C#, enter it, clean up any syntax errors, then test it to be sure it's working correctly

16

Problem-solving technique

9. Combine the decomposed tasks

- The event handler only responds when one particular checkbox (the top one) changes
- Now change its name to suit, and adjust it to handle the CheckedChanged (or Click) event of all the checkboxes
- See the next slide for a neat way to do this

10. Undo any simplifications to the decomposed tasks

- We didn't simplify the decomposed tasks any further, so there's nothing to undo

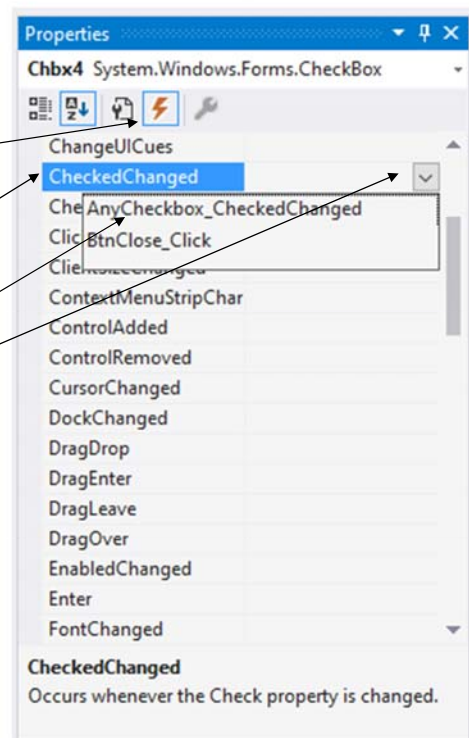
11. Undo any simplifications to the problem domain

- We reduced the 8 checkboxes to 3; once they are working, we have to add the other 5 and their code

17

Multiple events, one handler

- In the properties window, when you have a control selected, click the 'Events' icon
- This lists the events for that control
- Choose an event and use the drop-down menu to select a suitable event-handler – or double-click the event to create a new event handler



Problem-solving technique

12. Be sure that the solution does all that was required

- Now do some reasonably thorough testing on the full solution and check for any slips (for example, we might have accidentally added 8 for the 16 bit)
- When satisfied, re-read the specification and see if the program is really doing what it was supposed to

13. If appropriate, now that you understand the problem and the solution, consider whether a better solution might be found

- It's often the case that once you understand the problem this well, you can see a better solution
- However, if this solution is clear, works, and can be easily maintained, you might decide not to bother coding the improved version

19

The *foreach* statement

- Remember, the *foreach* statement loops through all the items in a collection. All the controls in a group box happen to form a collection (but note – *all* the controls, whatever their type).
- What about this for the binary numbers form?

```
CheckBox chbx;  
int iValue = 0;  
foreach (chbx in GpbxBits.Controls)  
    if (chbx.Checked)  
        iValue += Convert.ToInt32(chbx.Text);  
LblValue.Text = Convert.ToString(iValue);
```

- Is this an improvement that warrants rewriting the program? See the Checkboxes forms in Lec9Demo.

20

Incremental development

- In keeping with the idea of testing one method at a time, it makes sense to thoroughly test each method as soon as it's written
- If you wait until you've written many methods, it's much harder to find the source of any errors
- This suggests the following approach
 1. Design and create the form
 2. Write a small part of the program
 3. Type in that part, fix the syntax errors
 4. Test that part and debug it, get rid of any logic errors
 5. Repeat from step 2 until the program is complete

21

Incremental development example

- A friend tells you that superstitious people made a bad choice when they decided to fear Friday the 13th, because it occurs more often than Monday the 13th, Tuesday the 13th, or any other day the 13th
- You say this is rubbish: it's obvious that over time, the 13th will fall the same number of times on every day
- Your friend says no: the calendar repeats every 400 years, and if you count the 13ths in any 400 years, you will find that Friday falls more often than the rest
- You decide to practice your programming skills to check this

22

First confirm what you've been told

- Does the calendar repeat itself every 400 years? If not, it would be a waste of time counting the 13ths in that period.
- Well, 400 years is the minimum possible cycle length, because the pattern of leap years is 400 years long. For example, there was one more leap year in the 20th century (counting from 1901) than there will be in the 21st century (counting from 2001).
- Did you know that? If not, it would have been pointless trying to write the program.

23

Leap years

- Most people know that if a year can be divided by 4 with no remainder, it's a leap year. (Some people even know if the summer Olympics is on, it's a leap year.) Most of the time that's right, but . . .
- If it can also be divided by 100 with no remainder (eg 1800, 1900) it's *not* a leap year . . .
- Except that if it can also be divided by 400 with no remainder (eg 1600, 2000) it *is* a leap year.
- Was this done just to confuse us?
- No, it was done to keep the calendar in time with the earth's rotation around the sun, and thus with the seasons.

24

A 400-year cycle

- So if 2000 was a leap year, 2100, 2200, and 2300 won't be, and 2400 will be, 400 years is the minimum possible length of a repeating cycle in the calendar.
- How do we know if the calendar repeats every 400 years? Maybe it takes longer; maybe it never repeats.
- A lot of thought tells us that the calendar repeats if 400 years is a whole number of weeks, with no days left over. For example, if 1 Jan 2001 was a Monday, and 1 Jan 2401 will be a Monday, we have a cycle.
- We could check this just by typing the dates into Excel and formatting them to show the day of the week, but let's solve the problem more convincingly.

25

Number of days in 400 years

- The number of days in 400 years is
- 400×365 (146 000)
- + 97 (one day for each leap year, where one century has 25 leap years and the other three have 24)
- Total days: 146 097
- Total weeks: $146\,097 \div 7 = 20\,871$
- Yes, there is a whole number of weeks in 400 years, so the calendar repeats after that period

26

A program to count the 13ths

- First thoughts . . .
- We can perhaps find what day of the week a 13th falls on;
- we can perhaps jump straight to the 13th of the next month and see what day of the week it falls on;
- and keep doing this for the whole of the 400 years
- If we explore DateTime we find that it has some potentially helpful properties (such as DayOfWeek) and methods (such as AddMonths)
- (We could do this simply by declaring a DateTime variable, typing its name and a dot, and looking at the drop-down list that appears)

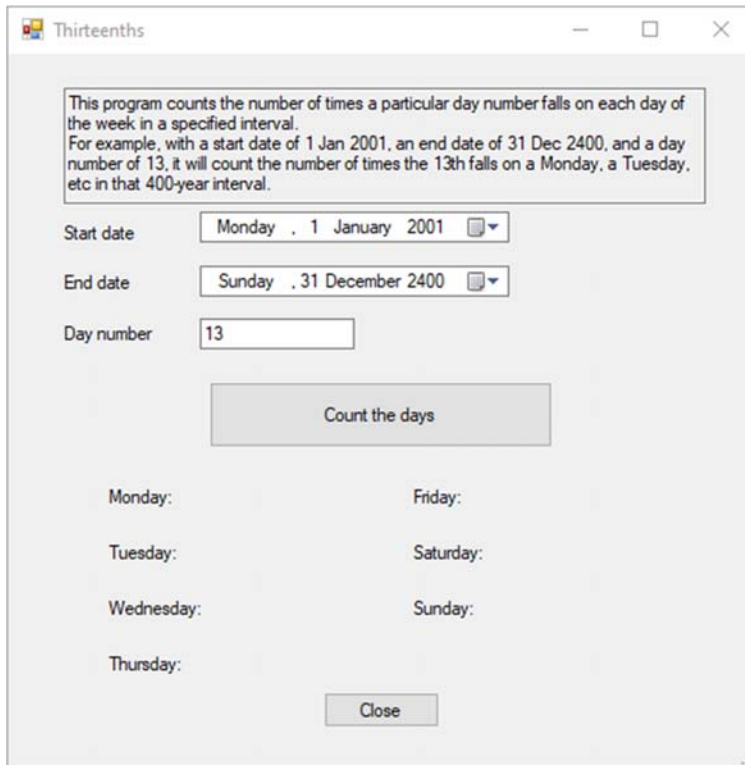
27

Form design

- Let's allow the program to count the incidence of any day number in any interval
- We'll allow the user to input the start date, the end date, and the day number to be counted
- We'll have a button to start the actual counting
- We'll have 7 labels to output the counts for the 7 days of the week
- The purpose of the program isn't obvious, so perhaps we'll also have some simple instructions. (We wouldn't generally do this on a program that's used frequently, as the instructions would add unnecessary clutter.)

28

The form



The instructions are in a textbox that's set to multi-line and read-only. That's a little bit easier than using labels.

29

Getting the data

```
// Collect the data from the form
DateTime dtRunningDate = DtpStartDate.Value; // Will change as we proceed
DateTime dtEndDate = DtpEndDate.Value;
int iDayNum = 13; // A default value in case of poor data
int iMonCount = 0, iTueCount = 0, iWedCount = 0, iThuCount = 0, iFriCount = 0,
    iSatCount = 0, iSunCount = 0;
try
{
    iDayNum = Convert.ToInt32(TbxDayNum.Text);
}
catch
{
    MessageBox.Show("The day number isn't valid.\r\nI'll assume you meant 13.",
        "Thirteenths");
}
```

- Note the separate counters for each day of the week
- Exception handling is needed for iDayNum, where users are expected to enter a number
- It's not needed for the dates, because a DateTimePicker always returns a valid date

30

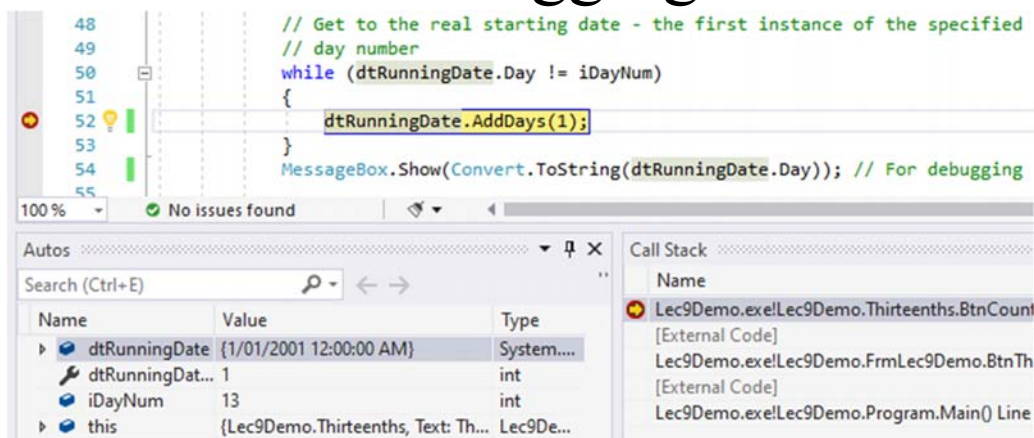
Shifting to the first 13th

```
// Get to the real starting date - the first instance of the specified
// day number
while (dtRunningDate.Day != iDayNum)
{
    dtRunningDate.AddDays(1);
}
MessageBox.Show(Convert.ToString(dtRunningDate.Day)); // For debugging
```

- Incremental development: test this before writing any more
- The MessageBox is just to confirm that we're at the 13th
- We try the program, it hangs
- It shouldn't take long to add 12 days – is it in an infinite loop?
- Let's add a breakpoint on the AddDays line and see what's happening

31

Debugging



- Each time through the loop the program stops, we look at the values in Autos, and tell it to continue . . .
- and each time `dtRunningDate` has the same value of `{01/01/2001 12:00:00 AM}`
- `AddDays` doesn't seem to be working

32

Using Help

- We now look up Help for AddDays
- Unfortunately, we might find the AddDays method of the Calendar class; so we try DateTime, and then find the AddDays method of the DateTime class . . .
- which eventually tells us “This method does not change the value of this DateTime. Instead, a new DateTime is returned whose value is the result of this operation.”
- Ah – it’s a function method, not a void method. It’s unfortunate that C# allows us to use function methods as complete statements; otherwise this would have been flagged as a syntax error
- We can now fix the problem and complete the program

33

Testing the program

- How do we know whether the counts are accurate?
- Test them on a period for which we can do the counts manually
- This year will do, if we have a calendar handy
- Set the start date to January 1 this year and the end date to December 31 this year, press the button, then see if the counts are accurate
- Perhaps test some more with a different year or a different day number. What happens if we specify a day that doesn’t arise in every month, such as 31? Help on DateTime.AddMonths should tell us.

34

So – is it true?

- You're now in a position to confirm what your friend told you
- Is Friday the 13th more common than any other day the 13th?
- Examine the demo program very carefully. Be sure that you understand it all (it's not really very involved).
- Change the AddDays line back to what we had originally, and practice using the breakpoint and debugging tool. Or do the same with the AddMonths line.
- Then consider putting the program aside and writing your own from scratch

35

Continue debugging

- If the user enters a non-numeric day number the exception handler catches the problem and uses the value 13
- If the user enters a numeric day number that isn't a valid day number, the program will still crash when it tries to AddDays(1)
- Fix the input section to disallow any integer outside the range 1-31
- Also add a warning, if the user enters a number more than 28, that the counts won't be accurate

36

You're not the only one

- You probably thought that any year evenly divisible by 4 was a leap year
- So did all the programmers who wrote Microsoft Excel
- In Excel, enter some consecutive dates in February 1900, then autofill them to continue the sequence; you'll find that Excel thinks 1900 was a leap year
- They eventually discovered the problem. Try the same thing in February 2100 and you'll find that it's not a leap year.
- Why didn't they fix the problem for 1900 when they fixed it for the future? Why did they prefer to leave this embarrassing error? There is a good reason.