Structural Programming and Data Structures

Winter 2000

CMPUT 102: Testing and Debugging

Dr. Osmar R. Zaïane

University of Alberta

Course Content

• Introduction	• Vectors
• Objects	• **Testing/Debugging**
• Methods	• Arrays
• Tracing Programs	• Searching
• Object State	• Files I/O
• Sharing resources	• Sorting
• Selection	• Inheritance
• Repetition	• Recursion

Objectives of Lecture 18

Black box testing and planned debugging

- Learn how to perform black-box testing on a program unit.
- After finding an error, we will use planned debugging to locate the error.

Outline of Lecture 18

- Kinds of program errors
- Testing
- Testing example, black-box testing of Person
- Planned debugging

Kinds of Errors

- There are four basic kinds of errors that can occur in a program:
  - syntax errors
  - compile-time semantic errors
  - run-time errors
  - logic or semantic errors

Syntax Errors

- A **syntax error** is a grammatical error:

```
name = aString. // period instead of semi-colon
years = today.getYear[]; // wrong kind of brackets
aPerson.setName('Fred'); // wrong String delimiter
```

- Syntax errors:
  - are found by the compiler
  - are often caused by typos
  - can usually be fixed quickly and easily.

Compile-Time Semantic Errors

- A **compile-time semantic error** is a non-syntax error that can be found by the compiler.
- One common class is type errors:

```
String yearString = today.getYear();
// bind a String variable to an int expression
int years = aPerson.getYear();
// A Person doesn't understand the getYear() message
```

- These errors:
  - are often caused by conceptual problems
  - are more difficult to fix than syntax errors

Run-time Errors

- A **run-time error** is an error that causes the program to stop running:

```
int years = birthdate.getYears();
// If birthdate is bound to null this program dies
Integer choice = Keyboard.in.readInteger();
int index = choice.intValue();
// If the user enters an invalid integer, the program dies
```

- Run-time errors
  - are not found by the compiler
  - are often due to uninitialized variables or bad input
  - may not be found until software is deployed

Logic Errors

- A **logic error** or **semantic error** is an error that produces incorrect results:

```
aPerson = new Person("Fred", new Date(69, 11, 25));
System.out.println(aPerson.age());
// If the current date is October, 28, 1999 and this program
outputs 30, there is a logic error.
```

- Logic errors
  - are not found by the compiler
  - do not result in program termination
  - might never be discovered
  - can be difficult to find and fix
  - can cause inconvenience, financial losses or disasters

Outline of Lecture 17

- Kinds of program errors
- Testing
- Testing example, black-box testing of Person
- Planned debugging

Testing versus Verification

- Testing is done to reduce the chances of releasing software that contains errors.
- Testing alone, can never guarantee that a program has no errors left.
- Sometimes, the correctness of critical portions of software are verified using an automatic proof checker.
- Verification is often too expensive for common software or for large software systems.

Time for Testing

- There are four common times when software is tested:
  - When a small unit of software is written, it undergoes a **unit test**.
  - When software units are integrated together, **integration testing** is done.
  - When the entire software system is finished, **system testing** is done.
  - When a unit is modified either to fix a problem or to add new features, **regression testing** is done to make sure no new errors are made.

Black-box & White-box Testing

- There are two kinds of testing, **black-box testing** and **white-box testing**.
- In black-box testing, the tester treats the software as a black box and does not see the implementation code.
- In white-box testing, the tester looks at the implementation code.

Black-box Testing

- Is used for functional testing to see if the software meets the specification and satisfies the user requirements.
- The tester creates a test based on all of the features in the specification.
- The tester checks the outputs for each input against the expected outputs defined by the specification.

White-box Testing

- The tester studies the implementation code.
- The tester chooses inputs that exercise each statement or path in the code.
- The tester also chooses inputs that check boundary conditions of selection and repetition control structures.

Design for Testing

- Testing is not something that should be done to software when it is finished.
- Software should be designed with testing in mind.
- Test suites should be constructed as the software is being specified and designed.
- Test code should be included with the software as it is written and kept for regression testing.

Testing Output

- You must know what output should be produced by the program, so you can tell if the test is successful.
- This output must be computed in an independent way.
- It is helpful to output the correct answers as part of the test routine.

Testing Object-Oriented Software

- For object-oriented software, a class is a good unit for testing.
- Black-box test suites consist of main programs that exercise all of the public methods.
- White-box test suites are created as public static methods.
- This approach is necessary since the state and some methods are usually private.

Outline of Lecture 18

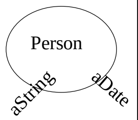
- Kinds of program errors
- Testing
- Testing example, black-box testing of Person
- Planned debugging

Unit Test Example - Person

- To demonstrate testing we will test a class called Person
  - Read specification
  - Construct an external black-box test suite
  - Run test suite
  - Correct errors by Debugging

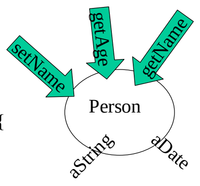
Specification Person(1)

```
public class Person {
/*
   Each instance of this class represents a Person with a name
   and age.
*/

/* Constructors */

   public Person(String aString, Date aDate) {
   /*
         Initialize me to have the given name and
         the given date as my birth date.
   */
```



Specification - Person(2)

```
/* Instance Methods */
   public String getName() {
   /*
         Return my name.
   */

   public void setName(String aString) {
   /*
         Set my name to the given String.
   */

   public int getAge() {
   /*
         Return my current age.
   */
```



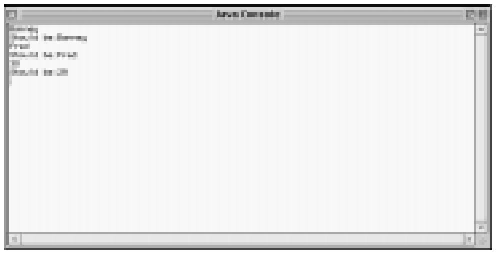
External Test Suite - Person 1

```
public static void main(String args[]) {

   Person     person;
   String     aName;
   // check Constructor for name init and getName()
   person = new Person("Barney", new Date(68, 11, 15));
   aName = person.getName();
   System.out.println(aName);
   System.out.println("Should be:Barney");
```

External Test Suite - Person 2

```
   // check setName(String);
   person.setName("Fred");
   aName = person.getName();
   System.out.println(aName);
   System.out.println("Should be:Fred");

   // check constructor for birthdate and getAge()
   System.out.println(person.getAge());
   System.out.println("Should be:29");
   // assume today's date is October 28, 1998
}
```

Output of External Test - Person



```
Barney
should be Barney
Fred
should be Fred
30
should be 29
```

Errors in class - Person

- Assume that today’s date is October 28, 1998.
- The age of someone born on November 15, 1968 is being reported as 30 instead of 29.
- We need to correct the program to fix this error.

Outline of Lecture 18

- Kinds of program errors
- Testing
- Testing example, black-box testing of Person
- Planned debugging

Debugging

- Run-time and logic errors are called bugs and fixing these bugs is called **debugging**.
- Debugging starts when an error is discovered during testing.
- Testing only identifies a symptom of the error, usually in an output statement.
- The error itself may be “far away” from the symptom in the code.
- The hardest part of debugging is finding the specific code that caused the error.

Debugging Approaches

- There are two kinds of debugging: ad-hoc and planned.
- In the **planned** approach, the programmer uses a four step process to try to deduce the location of the bug.
- In the **ad-hoc** approach, the programmer tries to examine the state of the program at various points of execution, looking for locations where the state is incorrect to zero in on the error location.

Four-Step Planned Debugging

1. understand the problem
   - make sure there are enough test cases to understand the real problem
2. devise a plan
   - develop one or more theories about the error
   - make a plan to confirm these theories
3. execute the plan
   - write more test cases to confirm one of the theories
4. review the solution
   - Inspect the code to verify that it is causing the error

Debug Example - Person

Understand the problem:

- The age is off by one year.
- Try another birth date to make sure that this is really the error:
  person = new Person("Barney", new Date(68, 6, 15));
- The output is now 30 which is correct.
- The age is not always off by one year.
- Perhaps the age is off by one year if the person has not had a birthday yet this year.

Debug Example - Person

Devise a plan:

- Conjecture: the age is off by one year if the person has not had a birthday yet this year and correct if the birthday has occurred.
- Construct test cases with some birthdays in each category to verify the conjecture:
  - January 1, 1950 - should be: 48
  - March 30, 1960 - should be: 38
  - November 10, 1970 - should be: 27
  - November 30, 1980 - should be: 17

Debug Example - Person

Execute the plan:

```
// check constructor for birthdate and getAge()
// birthdate after date in current year
    person = new Person("Barney", new Date(50, 1, 1));
    System.out.println(person.getAge());
    System.out.println("Should be:48");
    person = new Person("Barney", new Date(60, 3, 30));
    System.out.println(person.getAge());
    System.out.println("Should be:38");
// check constructor for birtdate and getAge()
// birthdate before date in current year
    person = new Person("Barney", new Date(70, 11, 10));
    System.out.println(person.getAge());
    System.out.println("Should be:27");
    person = new Person("Barney", new Date(80, 11, 30));
    System.out.println(person.getAge());
    System.out.println("Should be:17");
```

Output of Revised External Test

- The conjecture appears correct!

Debug Example - Person

Review the solution (1)

```
import java.util.*;
public class Person {
/* Each instance of this class represents a Person with a name
   and age. */
/* Constructors */
   public Person(String aString, Date aDate) {
   /*
        Initialize me to have the given name and
        the given date as my birth date.
   */
        this.name = aString;
        this.birthdate = aDate;
   }
```

Debug Example - Person

Review the solution (2)

```
/* Instance Methods */
   public String getName() {
   /*
        Return my name.
   */
        return this.name;
   }
   public void setName(String aString) {
   /*
        Set my name to the given String.
   */
        this.name = aString;
   }
```

Debug Example - Person

Review the solution (3)

```
public int getAge() {
   /*  Return my current age. */
        Date          today;
        int           todayYear;
        int           currentYear;
        int           age;

        today = new Date();
        todayYear = today.getYear();
        currentYear = this.birthdate.getYear();
        age = todayYear - currentYear;
        return age;
   }
```

Subtracting years is not sufficient if the month and day of the birth date has not passed yet.

Debug Example - Person

Review the solution (4)

```
/* Instance Variables */
private String  name;
private Date    birthdate;

}
```

Debug Example - Person

Fix the code 1

```
public int getAge() {
   /*  Return my current age. */
        Date    today;          int  age;
        int     todayYear;      int  currentYear;

        today = new Date();
        todayYear = today.getYear();
        currentYear = this.birthdate.getYear();
        age = todayYear - currentYear;
        if (! this.hadBirthdayThisYear())
                age = age - 1;
        return age;
   }
```

Debug Example - Person

Fix the code 2

```
private boolean hadBirthdayThisYear() {
   /*  Return true if I have already had my
        birthday this year. */

        Date    today;  int     todayMonth;
        int     myMonth;        int     myDayOfMonth;
        int     todayDayOfMonth;

        today = new Date();
        todayMonth = today.getMonth();
        myMonth = this.birthdate.getMonth();
        todayDayOfMonth = today.getDate();
        myDayOfMonth = birthdate.getDate();
```

Debug Example - Person

Fix the code 3

```
        if (myMonth < todayMonth)
                return true;
        else if (myMonth > todayMonth)
                return false;
        else if (myDayOfMonth <= todayDayOfMonth)
                return true;
        else
                return false;
   }
```

Output of Revised External Test

After fixing the birthday bug

- The birthday bug is fixed.

Course Content

• Introduction	• Vectors
• Objects	• **Testing/Debugging**
• Methods	• Arrays
• Tracing Programs	• Searching
• Object State	• Files I/O
• Sharing resources	• Sorting
• Selection	• Inheritance
• Repetition	• Recursion

Objectives of Lecture 19

White box testing and ad-hoc debugging

- Learn how to perform white-box testing on a program unit.
- After finding an error, we will use ad-hoc debugging to locate the error.

Outline of Lecture 19

- Inspecting an example Class's code for paths.
- Constructing an internal white-box test suite.
- Correcting errors by Ad-hoc Debugging.

Unit Test Example - Person

- To demonstrate testing we will continue testing a class called Person
  – Inspect code
  – Construct an internal white-box test suite
  – Run test suite
  – Correct errors by Debugging

Example - Person

Inspect the code for paths 1

```
public int getAge() {
  /*  Return my current age.  */
        Date    today;          int   age;
        int     todayYear;  int  currentYear;

        today = new Date();
        todayYear = today.getYear();
        currentYear = this.birthdate.getYear();
        age = todayYear - currentYear;
        if (! this.hadBirthdayThisYear())
                age = age - 1;
        return age;
  }
```

Find tests that execute both paths, but also include the boundary. Look in this method to identify the boundary.

Example - Person

Inspect the code for paths 2

```
private boolean hadBirthdayThisYear() {
   /*    Return true if I have already had my
         birthday this year. */

         Date    today;         int todayMonth;
         int     myMonth;   int myDayOfMonth;
         int     todayDayOfMonth;

         today = new Date();
         todayMonth = today.getMonth();
         myMonth = this.birthdate.getMonth();
         todayDayOfMonth = today.getDate();
         myDayOfMonth = this.birthdate.getDate();
```

Example - Person

Inspect the code for paths 3

```
        if (myMonth < todayMonth)
                return true;
        else if (myMonth > todayMonth)
                return false;
        else if (myDayOfMonth <= todayDayOfMonth)
                return true;
        else
                return false;
    }
```

Pick myMonth one less, equal and one greater then todayMonth.

Pick myDayOfMonth one less, equal and one greater than todayDayOfMonth

Outline of Lecture 19

- Inspecting an example Class's code for paths.
- Constructing an internal white-box test suite.
- Correcting errors by Ad-hoc Debugging.

Example - Person

White-box test method

- If today's date is October 28, 1998
- Construct test cases for birth dates:
  - September 28, 1950 - should be: 48
  - October 28, 1950 - should be: 48
  - November 28, 1950 - should be: 47
  - October 27, 1950 - should be: 48
  - October 29, 1950 - should be: 47

Test Example - Person

code for white box test method 1

```
public static void test() {

    Person          person;
    String          aName;

    // check Constructor for name init and getName()
    person = new Person("Barney", new Date(68, 11, 15));
    aName = person.getName();
    System.out.println(aName);
    System.out.println("Should be:Barney");

    // check setName(String);
    person.setName("Fred");
    aName = person.getName();
    System.out.println(aName);
    System.out.println("Should be:Fred");
```

Test Example - Person

code for white box test method 2

```
// check constructor for birtdate and getAge()
    person = new Person("Barney", new Date(50, 9, 28));
    System.out.println(person.getAge());
    System.out.println("Should be:48");
    person = new Person("Barney", new Date(50, 10, 28));
    System.out.println(person.getAge());
    System.out.println("Should be:48");
    person = new Person("Barney", new Date(50, 11, 28));
    System.out.println(person.getAge());
    System.out.println("Should be:47");
    person = new Person("Barney", new Date(50, 10, 27));
    System.out.println(person.getAge());
    System.out.println("Should be:48");
    person = new Person("Barney", new Date(50, 10, 29));
    System.out.println(person.getAge());
    System.out.println("Should be:47");
}
```

Output of Person

White-box test method

- We have found another bug!

Outline of Lecture 19

- Inspecting an example Class's code for paths.
- Constructing an internal white-box test suite.
- Correcting errors by Ad-hoc Debugging.

Ad-hoc Debugging

- Assume that testing uncovered an error when the value of a variable called myVariable was output.
- The error occurred sometime between when the program started and when the output statement was performed.
- By examining the values that myVariable was bound to at various points of program execution, the location of the bug can be narrowed.

Examining Program State

- There are three common techniques for examining program state:
  - tracing the code by hand and recording the state when it changes
  - putting output statements into the code whenever a variable is rebound
  - using an automated debugger to step through the execution and examine the program state

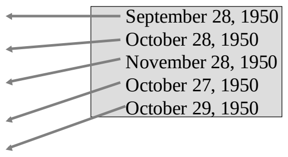
Example - Person

Ad-hoc debugging - output statements: (1)

```
private boolean hadBirthdayThisYear() {
   /*  Return true if I have already had my
       birthday this year. */
       Date    today;          int  todayMonth;
       int     myMonth;   int  myDayOfMonth;
       int     todayDayOfMonth;

       today = new Date();
       System.out.print("today: ");
       System.out.println(today.toString());
       System.out.print("birthdate: ");
       System.out.println(this.birthdate.toString());
       ...
```

Output of Person

Ad-hoc debugging - output statements



```
September 28, 1950
October 28, 1950
November 28, 1950
October 27, 1950
October 29, 1950
```

- The months of the birthdates are wrong!

Output of Person

Ad-hoc debugging - output statements

- Find all places in the code, before the incorrect output, where the variable birthdate is rebound.
- Find all places in the code, before the incorrect output, where a message is sent to the birthdate object that could change its state.
- Put output statements after these places.

Example - Person

references to birthdate

```
import java.util.*;
public class Person {
/* Each instance ... */
                                   birthdate reference

/* Constructors */
   public Person(String aString, Date aDate) {
   /*    Initialize me ... */
         this.name = aString;
         this.birthdate = aDate;
         System.out.print(“birthdate: “);
         System.out.println(this.birthdate);
   }
```

Example - Person

references to birthdate

```
public int getAge() {
   /*… Return my current age. … */
         Date   today;        int  age;
         int    todayYear;  int  currentYear;

         today = new Date();                    birthdate reference
         todayYear = today.getYear();
         currentYear = this.birthdate.getYear();
         System.out.print(“birthdate: “);
         System.out.println(this.birthdate);
         age = todayYear - currentYear;
         if (! this.hadBirthdayThisYear())     age = age - 1;
         return age;
   }
```

Output of Person

Ad-hoc debugging - output statements

```
September 28, 1950
October 28, 1950
November 28, 1950
October 27, 1950
October 29, 1950
```

- The months are wrong in the constructor!

Example - Person

Ad-hoc debugging - output statements

- The output statements indicate that the birthdate is incorrect when it is first passed to the constructor:

```
/* Constructors */
   public Person(String aString, Date aDate) {
   /* Initialize me … */
         this.name = aString;
         this.birthdate = aDate;
   }
   person = new Person("Barney", new Date(50, 9, 28));
```

- Check the spec of the Date constructor to see why Date(50, 9, 28) seems to return October 28 instead of September 28

Example - Person

Date Constructor Specification:

public Date()

The month must be from 0 - 11 instead of from 1 - 12!

Allocates a Date object and initializes it so that it represents midnight, local time, at the beginning of the day specified by the year, month, and date arguments.

Parameters:

year - the year minus 1900.

month - the month between 0-11.

date - the day of the month between 1-31.

Example - Person

Fix the test method:

- The bug was actually in the test method, not the class itself.
- The test method must be modified.
- Also include documentation in the test method that indicates how to use the Date constructor properly.
- Re-run the test.
- Remove the debugging code and re-run it again.

Example - Person

final code for white box test method 1

```
public static void test() {

    Person      person;
    String      aName;

    // check Constructor for name init and getName()
    person = new Person("Barney", new Date(68, 11, 15));
    aName = person.getName();
    System.out.println(aName);
    System.out.println(“Should be:Barney”);
```

Example - Person

final code for white box test method 2

```
// check setName(String);
    person.setName("Fred");
    aName = person.getName();
    System.out.println(aName);
    System.out.println(“Should be:Fred”);

// check constructor for birtdate and getAge()
    person = new Person("Barney", new Date(50, 8, 28));
// September 28, 1950,  since month: 0-11, day: 1-31
    System.out.println(person.getAge());
    System.out.println(“Should be:48”);
    person = new Person("Barney", new Date(50, 9, 28));
    System.out.println(person.getAge());
    System.out.println(“Should be:48”);
```

Example - Person

final code for white box test method 3

```
    person = new Person("Barney", new Date(50, 10, 28));
    System.out.println(person.getAge());
    System.out.println(“Should be:47”);
    person = new Person("Barney", new Date(50, 9, 27));
    System.out.println(person.getAge());
    System.out.println(“Should be:48”);
    person = new Person("Barney", new Date(50, 9, 29));
    System.out.println(person.getAge());
    System.out.println(“Should be:47”);
}
```

Output of Person

White-box testing