SWEN 6301 Software Construction

Lecture 8: Developer Testing and Debugging

Testing

Testing

- Some programmers use the terms “testing” and “debugging” interchangeably, but not right.
- Testing is a means of detecting errors.
- Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected.

Testing can only show the presence of errors, not their absence

Testing

- The testing process has two distinct goals:
  - To demonstrate to the developer and the customer that the software meets its requirements.
    - For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
  - To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification.
    - These are a consequence of software defects. Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

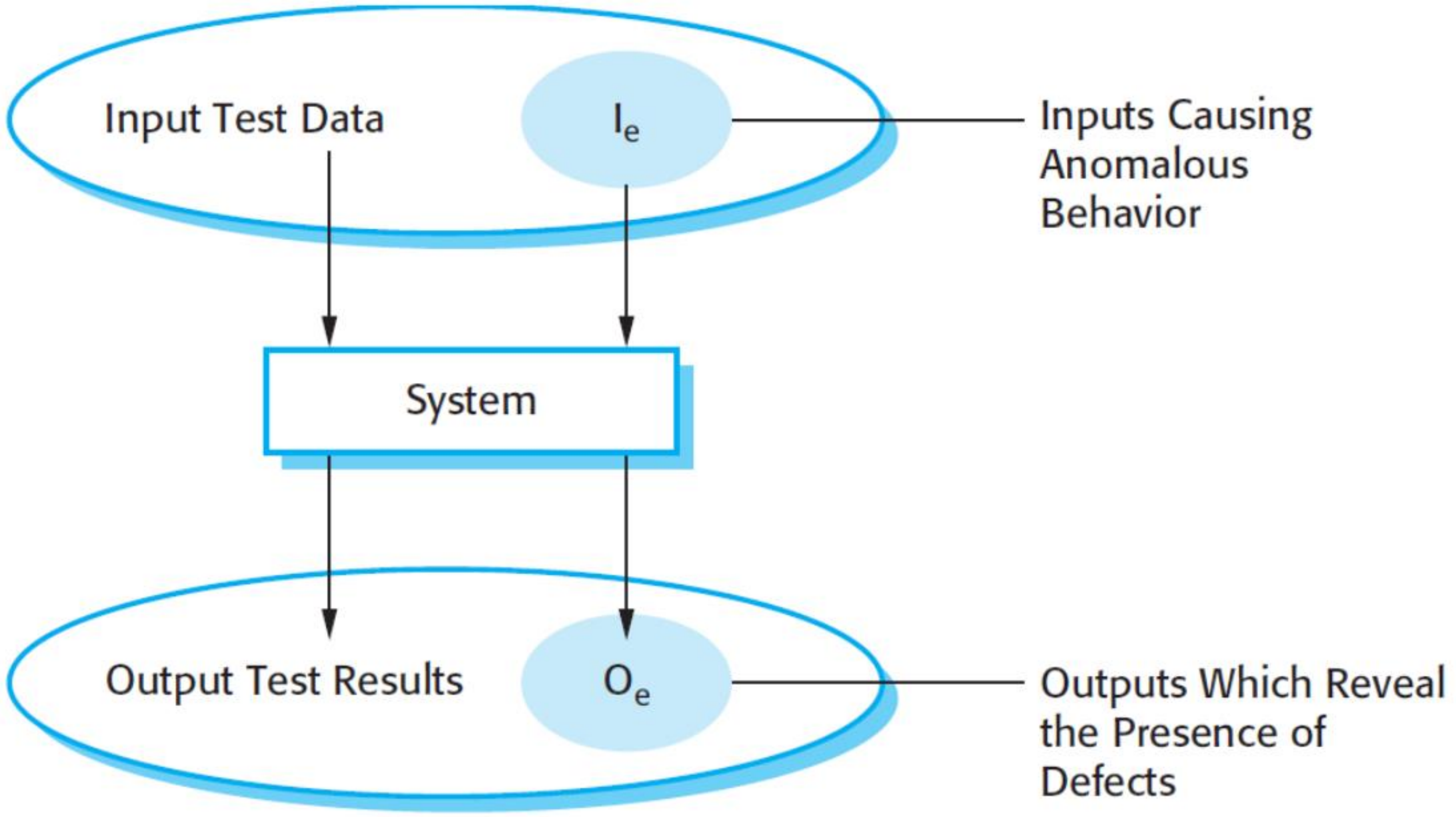
Testing

- The first goal leads to validation testing, where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
- The second goal leads to defect testing, where the *test cases are designed to expose defects*.

Testing

- Defect testing is to find those inputs in the set I_e that reveal problems with the system.

- Validation testing involves testing with correct inputs that are outside I_e, the system to generate the expected correct outputs.


Input Test Data
I_e
Inputs Causing Anomalous Behavior
System
Output Test Results
O_e
Outputs Which Reveal the Presence of Defects

Testing

- The ultimate goal is to establish confidence that the software system is ‘fit for purpose’.
- This means that the system must be good enough for its intended use.
- The level of required confidence depends on the system’s purpose, the expectations of the system users, and the current marketing environment for the system:

Testing

- ***Software purpose***
- The more critical the software, the more important that it is reliable.
- For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.

Testing

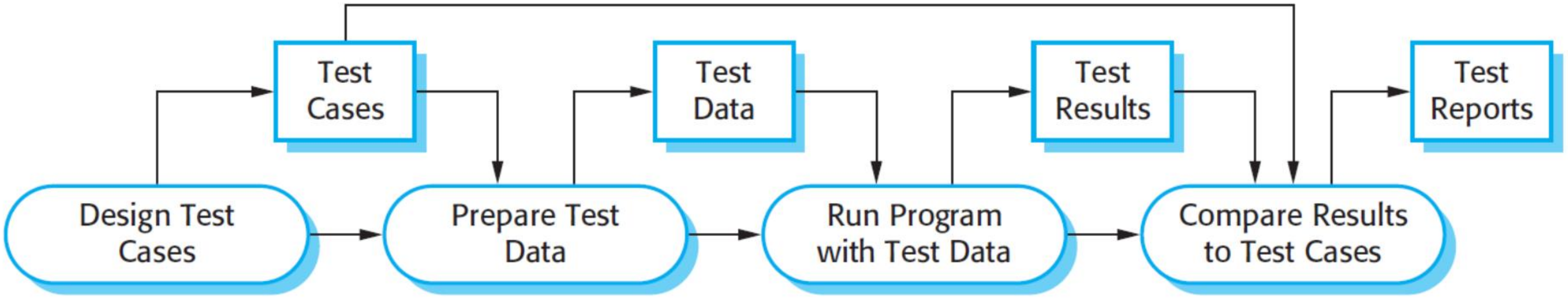
- ***User expectations***
- Because of their experiences with buggy, unreliable software, many users have low expectations of software quality. They are not surprised when their software fails.
- When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery.
- In these situations, you may not need to devote as much time to testing the software.
- However, as software matures, users expect it to become more reliable so more thorough testing of later versions may be required.

Testing

- ***Marketing environment***
- When a system is marketed, the sellers of the system must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system.
- In a competitive environment, a software company may decide to release a program before it has been fully tested and debugged because they want to be the first into the market.
- *If a software product is very cheap, users may be willing to tolerate a lower level of reliability*.

Testing

- Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested.
- Test data are the inputs that have been devised to test a system.

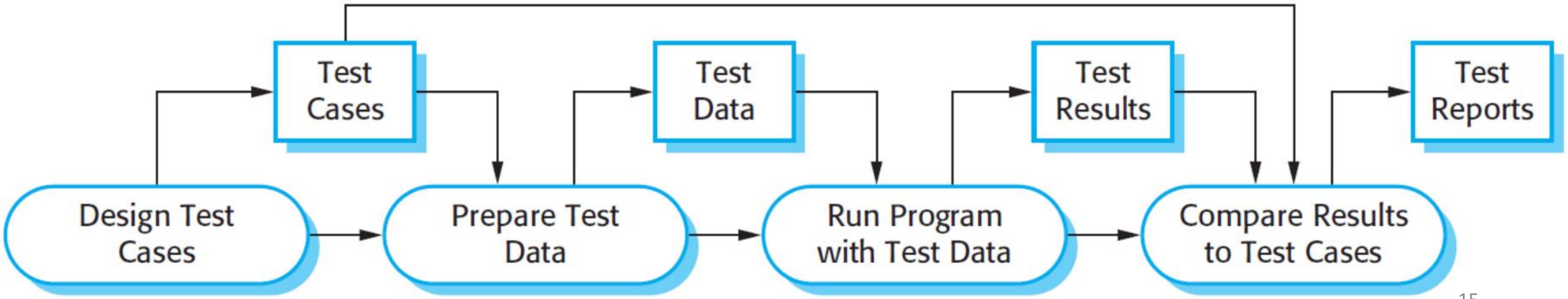


Testing

- Test data can be generated automatically,
- but automatic test case generation is impossible?

What about mutation testing?

- However, test execution can be automated.



Testing

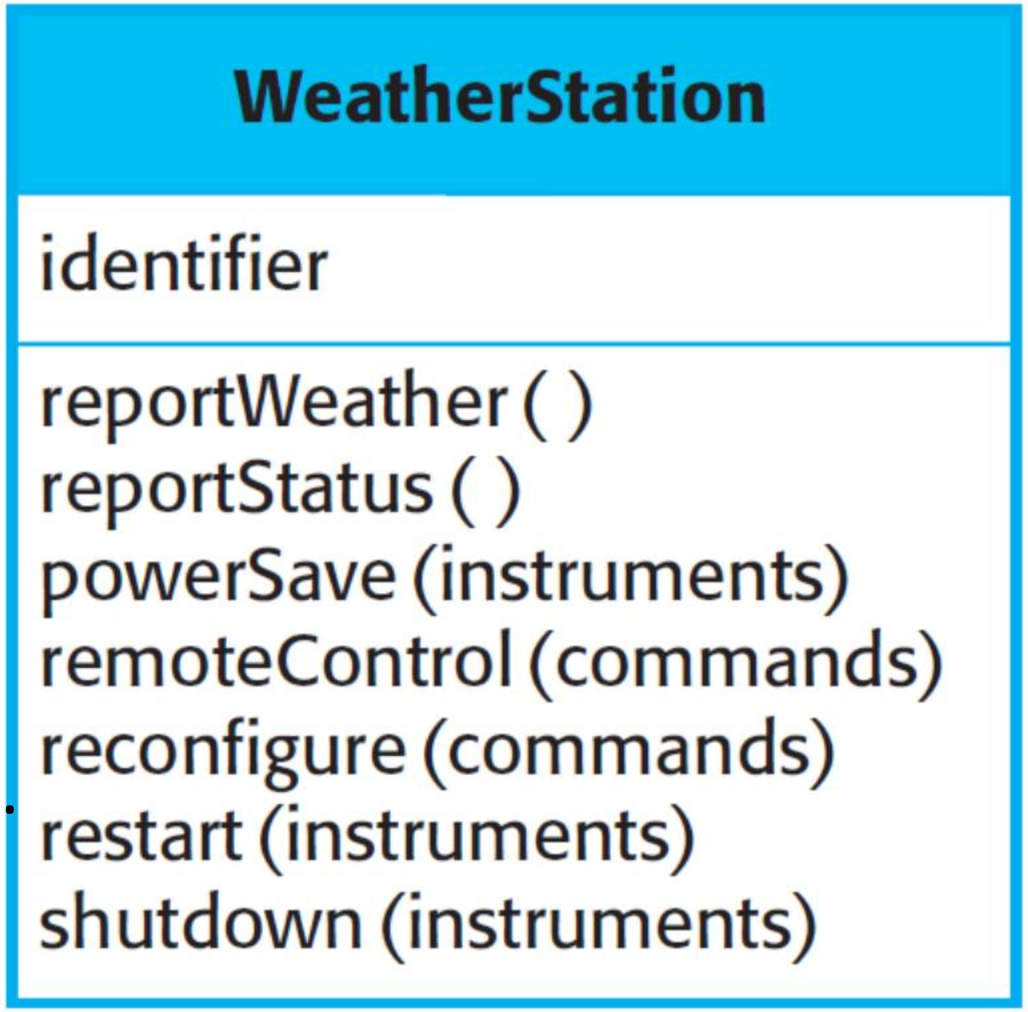
- Unit testing is the execution of a complete class, routine, or small program that has been written by a single programmer or team of programmers, which is tested in isolation from the more complete system.
  - **Unit tests are basically written and executed by software developers** to make sure that code behaves as expected.

Testing – *Unit Testing*

- When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. This means that you should:
  - test all operations associated with the object;
  - set and check the value of all attributes associated with the object;
  - put the object into all possible states. This means that you should simulate all events that cause a state change.

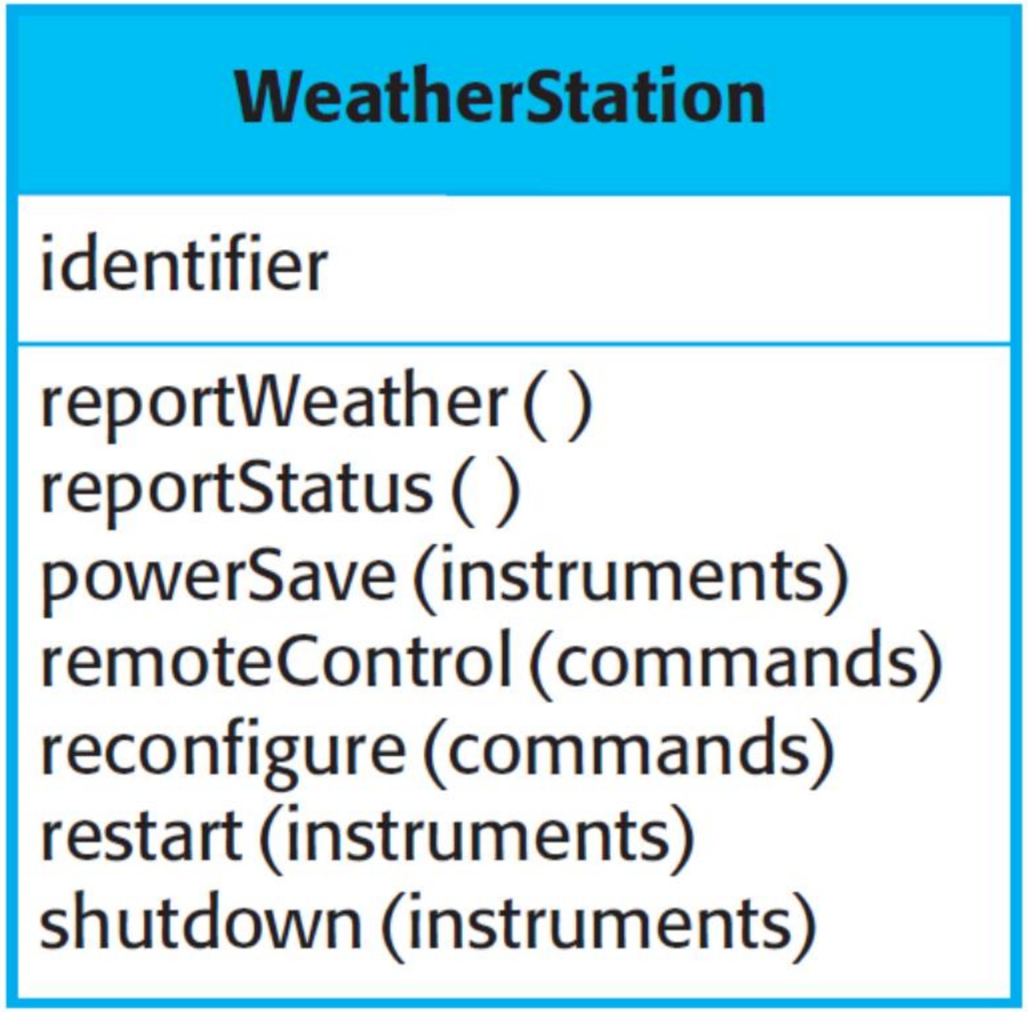
Testing – *Unit Testing*

- For example, **WeatherStation** object
- It has a single attribute, which is its **identifier**. This is a constant that is set when the weather station is installed.
- You therefore only need a test that checks if it has been properly set up.
- You need to define test cases for all of the methods associated with the object such as **reportWeather**, **reportStatus**, etc.



Testing – *Unit Testing*

- Ideally, you should test methods in isolation but, in some cases, some test sequences are necessary.
- For example, to test the method that shuts down the weather station instruments (**shutdown**), you need to have executed the **restart** method.

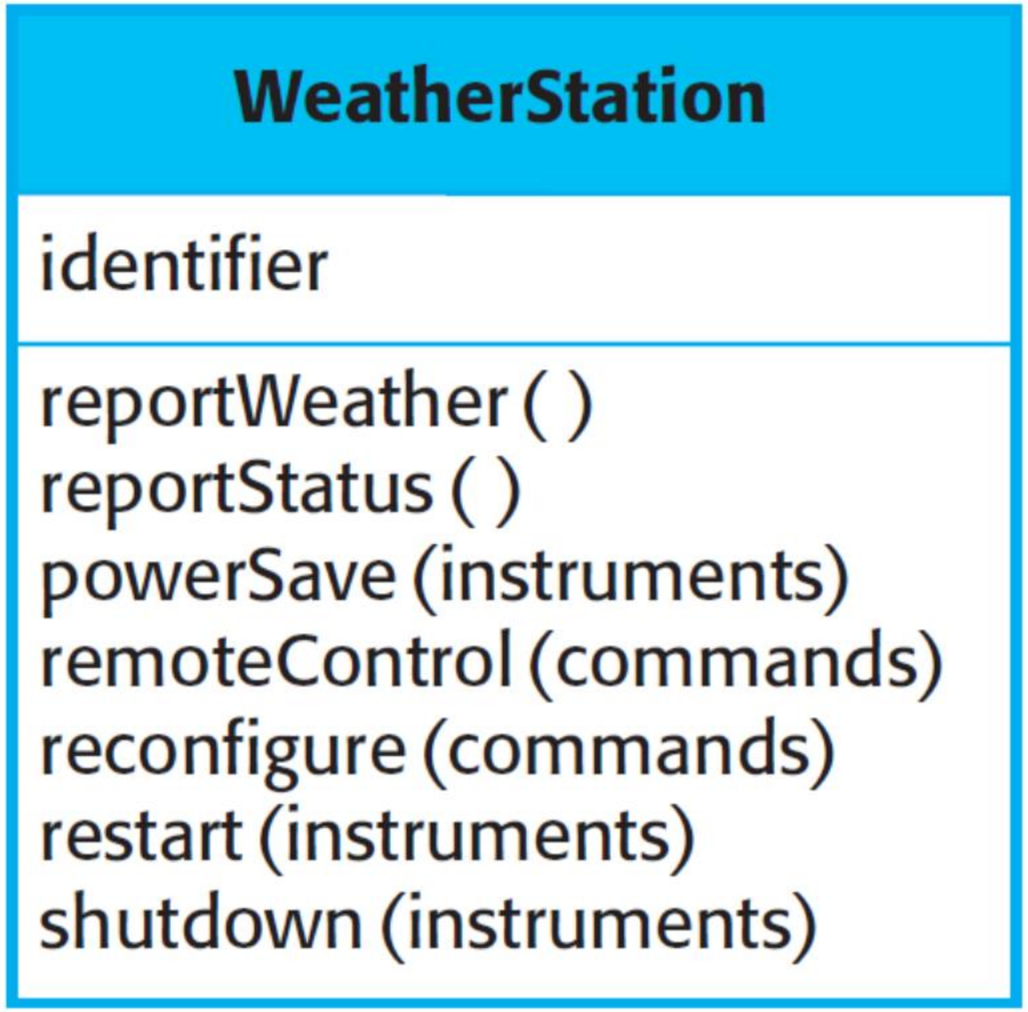


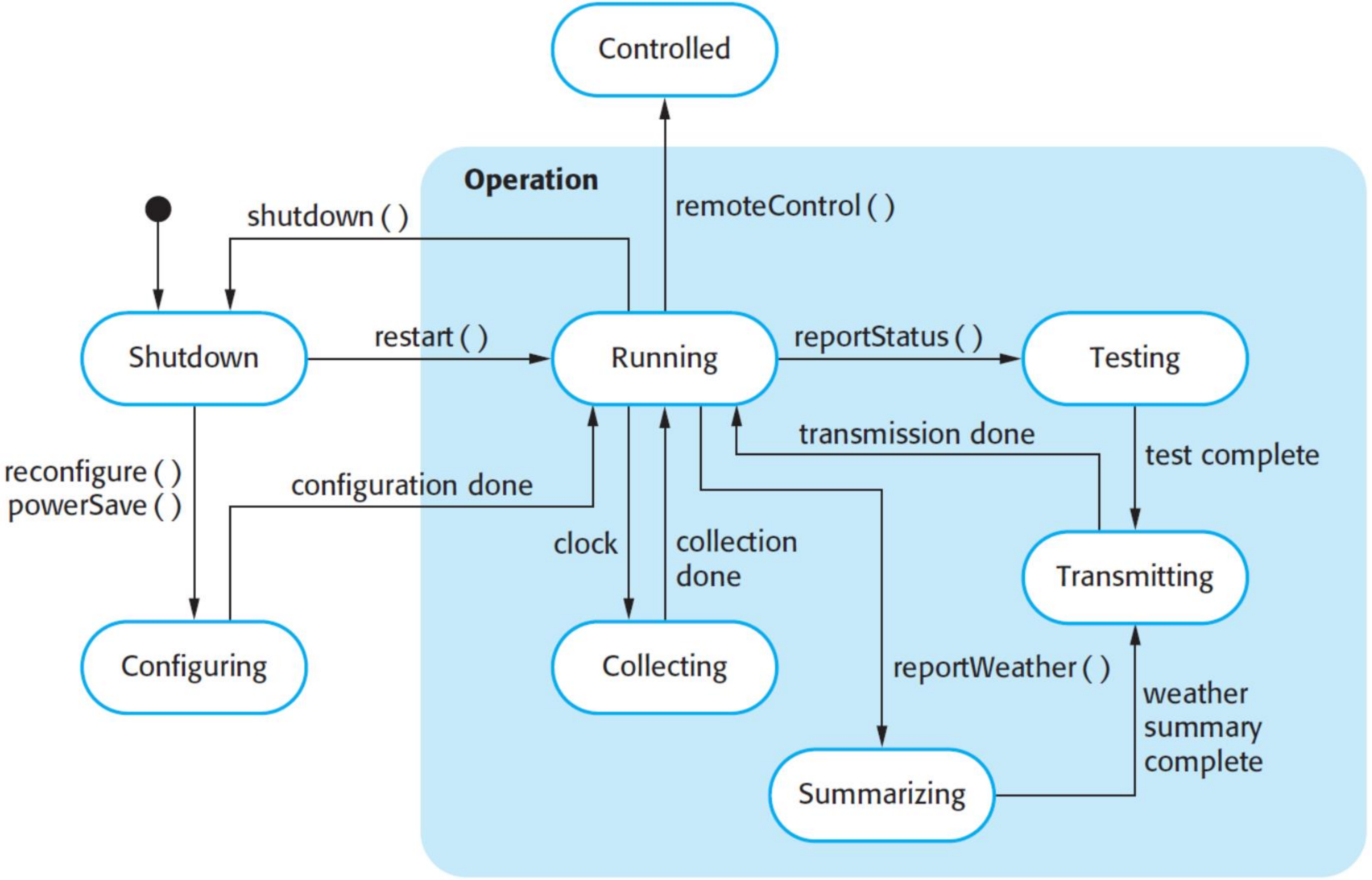
Testing – *Unit Testing*

- ***Inheritance*** makes object class testing more complicated.
- You can’t simply test an operation in the class where it is defined and assume that it will work as expected in the subclasses that inherit the operation.
- The operation that is inherited may make assumptions about other operations and attributes.
- These may not be valid in some subclasses that inherit the operation. You therefore have to test the inherited operation in all of the contexts where it is used.

Testing – *Unit Testing*

- To test the states of the weather station, you use a state model (as on the next slide)
- Using this model, you can identify sequences of state transitions that have to be tested and define event sequences to force these transitions.
- In principle, you should test every possible state transition sequence, although in practice this may be too expensive.




Controlled
Operation
remoteControl ( )
shutdown ( )
Shutdown
restart ( )
Running
reportStatus ( )
Testing
transmission done
test complete
reconfigure ( )
powerSave ( )
configuration done
clock
collection done
Transmitting
Configuring
Collecting
reportWeather ( )
weather summary complete
Summarizing

Testing – *Unit Testing*

- Whenever possible, you should automate unit testing.
- In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases.
- They can then run all of the tests that you have implemented and report, often through some GUI, on the success or failure of the tests.
- An entire test suite can often be run in a few seconds so it is possible to execute all the tests every time you make a change to the program.

Testing – *Unit Testing*

- An automated test has three parts:
  1. A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
  2. A call part, where you call the object or method to be tested.
  3. An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

Testing – *Unit Testing*

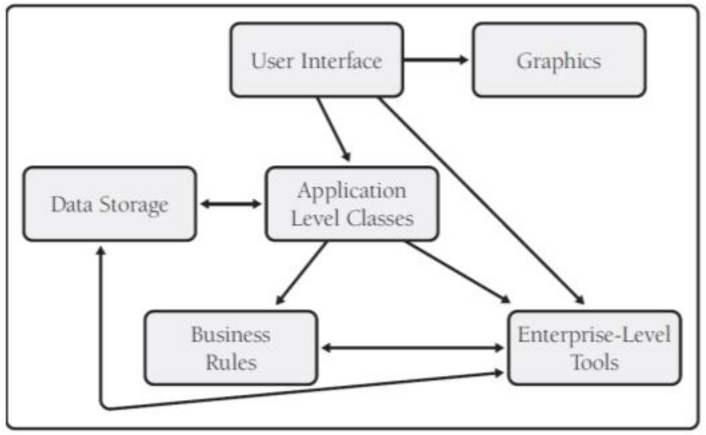
- Sometimes the object that you are testing has dependencies on other objects that may not have been written or which slow down the testing process if they are used.
- For example, if your object calls a database, this may involve a slow setup process before it can be used.
- In these cases, you may decide to use mock objects.

Testing – *Unit Testing*

- Mock objects are objects with the same interface as the external objects being used that simulate its functionality.
- Therefore, a mock object simulating a database may have only a few data items that are organized in an array.
- They can therefore be accessed quickly, without the overheads of calling a database and accessing disks.

Testing

- Component testing is the execution of a class, package, small program, or other program element that involves the work of multiple programmers or programming teams, which is tested in isolation from the more complete system.
  - Like subsystem testing
  - Stubs can be used for testing a subsystem while the connected subsystems aren't finished yet



Testing

- Integration testing is the combined execution of two or more classes, packages, components, or subsystems that have been created by multiple programmers or programming teams.
- This kind of testing typically starts as soon as there are **two classes** to test and continues until the entire system is complete.

Testing

- Regression testing is the repetition of previously executed test cases for the purpose of finding defects in modified software that previously passed the same set of tests.
  - Any new feature is added
  - Any enhancement is done
  - Any bug is fixed
  - Any performance related issue is fixed

Testing

- System testing is the execution of the software in its final configuration, including integration with other software and hardware systems.
- It tests for security, performance, resource loss, timing problems, and other issues that can’t be tested at lower levels of integration.
- It may include tests based on
  - risks and/or requirement specifications, business process, use cases, or other high level descriptions of system behavior, interactions with the operating systems, and system resources.

Testing

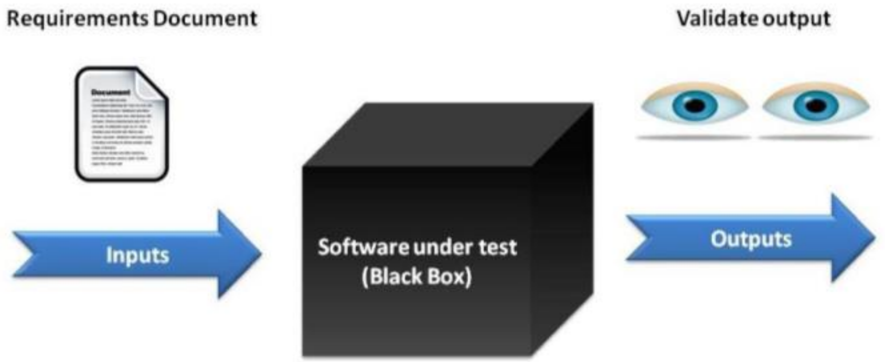
- Acceptance testing is the execution of the software after it is released, by the customer
    - After the system test has corrected all or most defects, the system will be delivered to the user or customer for acceptance testing.

Testing

- Testing is usually broken into two broad categories:
  - black-box (or specification-based) testing
  - white-box (or glass-box) testing

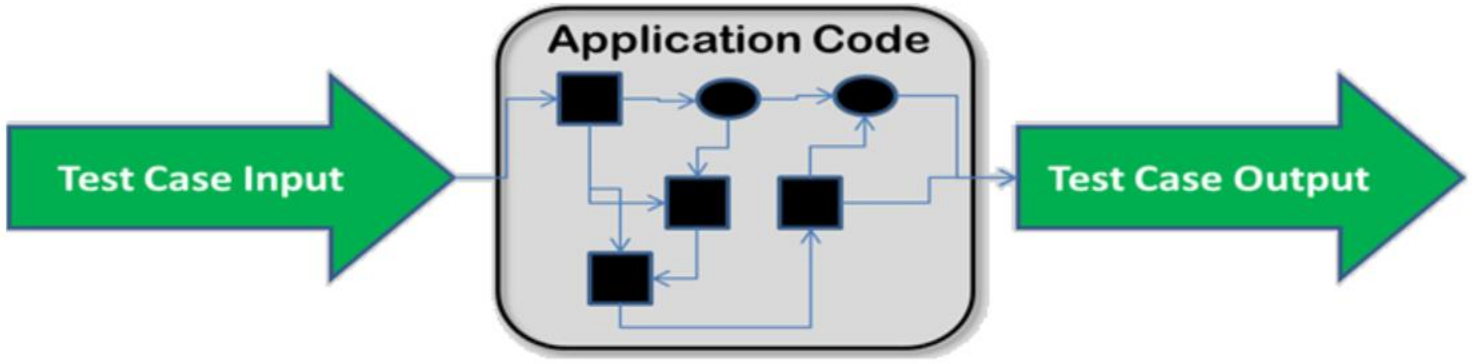
Testing

- Black-box testing refers to tests in which the tester cannot see the inner workings of the item being tested. This obviously does not apply when you test code that you have written!
  - a.k.a. Specification-based testing technique or input/output driven testing techniques because they view the software as a black-box with inputs and outputs.
  - Concentrating on what the software does, not how it does it.



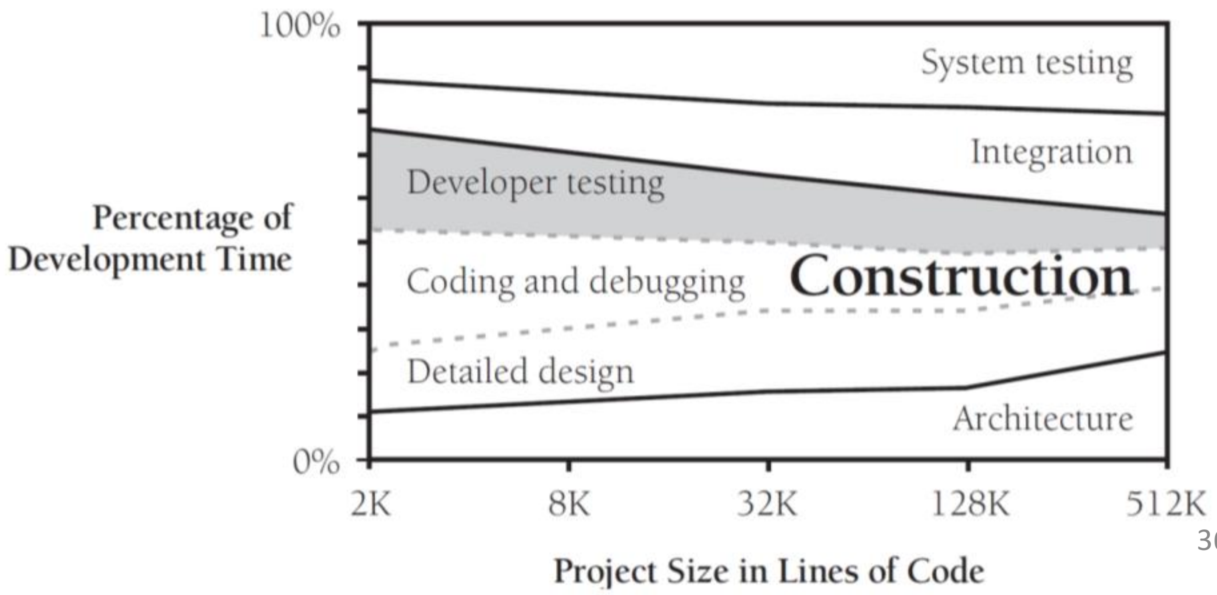
Testing

- White-box testing refers to tests in which the tester is aware of the inner workings of the item being tested. *This is the kind of testing that you as a developer use to test your own code*.
  - a.k.a. Structure-based testing technique is or 'glass-box' testing technique because here the testers require knowledge of how the software is implemented, how it works.



Developer Testing

- Developer testing typically consists of unit tests, component tests, and integration tests
    - but can sometimes include regression tests and system tests.
- A key question is, How much time should be spent in developer testing on a typical project?



Developer Testing

- What do you do with the results of developer testing?
  - To assess the reliability of the product under development. *Even if you never correct the defects that testing finds, testing describes how reliable the software is*.
  - To guide corrections to the software.
  - Finally, over time, the record of defects found through testing helps reveal the kinds of errors that are most common. You can use this information to select appropriate training classes, direct future technical review activities, and design future test cases.

Developer Testing

- During construction, you generally write a routine or class, check it mentally, and then review it or test it.
- Regardless of your integration or system-testing strategy, you should test each unit thoroughly before you combine it with any others.

Developer Testing

- *Test for each relevant requirement to make sure that the requirements have been implemented*.
- Plan the test cases for this step at the requirements stage or as early as possible—preferably before you begin writing the unit to be tested.
- Consider testing for common omissions in requirements.
- The level of security, storage, the installation procedure, and system reliability are all fair game for testing and are often missed at requirements time.

Developer Testing

- *Test for each relevant design concern to make sure that the design has been implemented*.
- Plan the test cases for this step at the design stage or as early as possible-before you begin the detailed coding of the routine or class to be tested.

Developer Testing

- *Use basis testing to add detailed test cases to those that test the requirements and the design*.
- Add data-flow tests, and then add the remaining test cases needed to thoroughly exercise the code.
- At a minimum, you should test every line of code.
- Basis testing and data-flow testing are described later...

Developer Testing

- *Use a checklist of the kinds of errors you've made on the project to date or have made on previous projects*.

Developer Testing – Test Cases First

- Writing test cases before writing the code doesn’t take any more effort than writing test cases after the code;
  - it simply resequences the test-case-writing activity.
- Writing test cases first helps to detect defects earlier and you can correct them more easily.
- Writing test cases first forces you to think at least a little bit about the requirements and design before writing code, which tends to produce better code.
- Writing test cases first exposes requirements problems sooner, before the code is written, because it’s hard to write a test case for a poor requirement.
- If you save your test cases, which you should do, you can still test last, in addition to testing first.

Developer Testing - Limitations

- ***Developer tests tend to be "clean tests"***
- Developers tend to test for whether the code works (clean tests) rather than test for all the ways the code breaks (dirty tests).
- Immature testing organizations tend to have about five clean tests for every dirty test.
- Mature testing organizations tend to have five dirty tests for every clean test. This ratio is not reversed by reducing the clean tests; it's done by creating 25 times as many dirty tests (Boris Beizer in Johnson 1994).

Developer Testing - Limitations

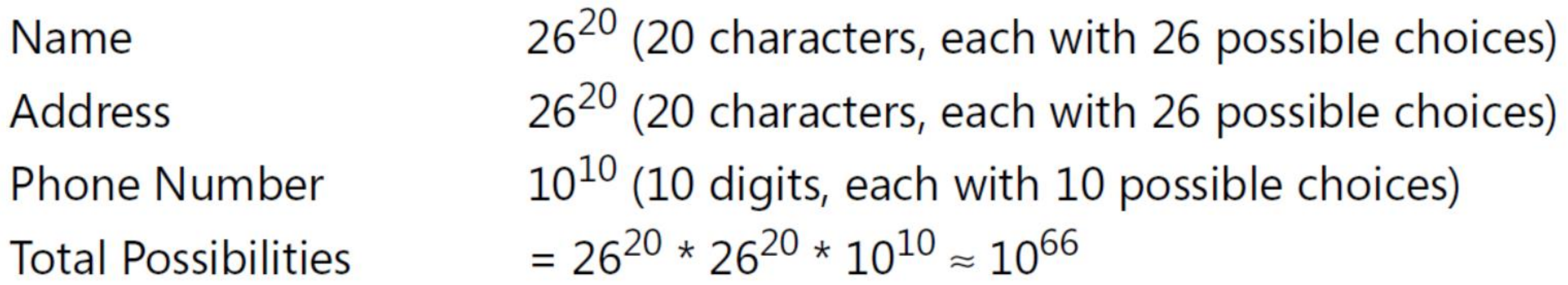
- ***Developer testing tends to have an optimistic view of test coverage***
- Average programmers believe they are achieving 95 percent test coverage, but they’re typically achieving more like 80 percent test coverage in the best case, 30 percent in the worst case, and more like 50-60 percent in the average case (Boris Beizer in Johnson 1994).

Developer Testing - Limitations

- ***Developer testing tends to skip more sophisticated kinds of test coverage***
- Most developers view the kind of test coverage known as “100% statement coverage” as adequate.
- This is a good start, but it’s hardly sufficient.
- A better coverage standard is to meet what’s called “100% branch coverage,” with every predicate term being tested for at least one true and one false value.
  What is the ultimate coverage?

Testing Tricks

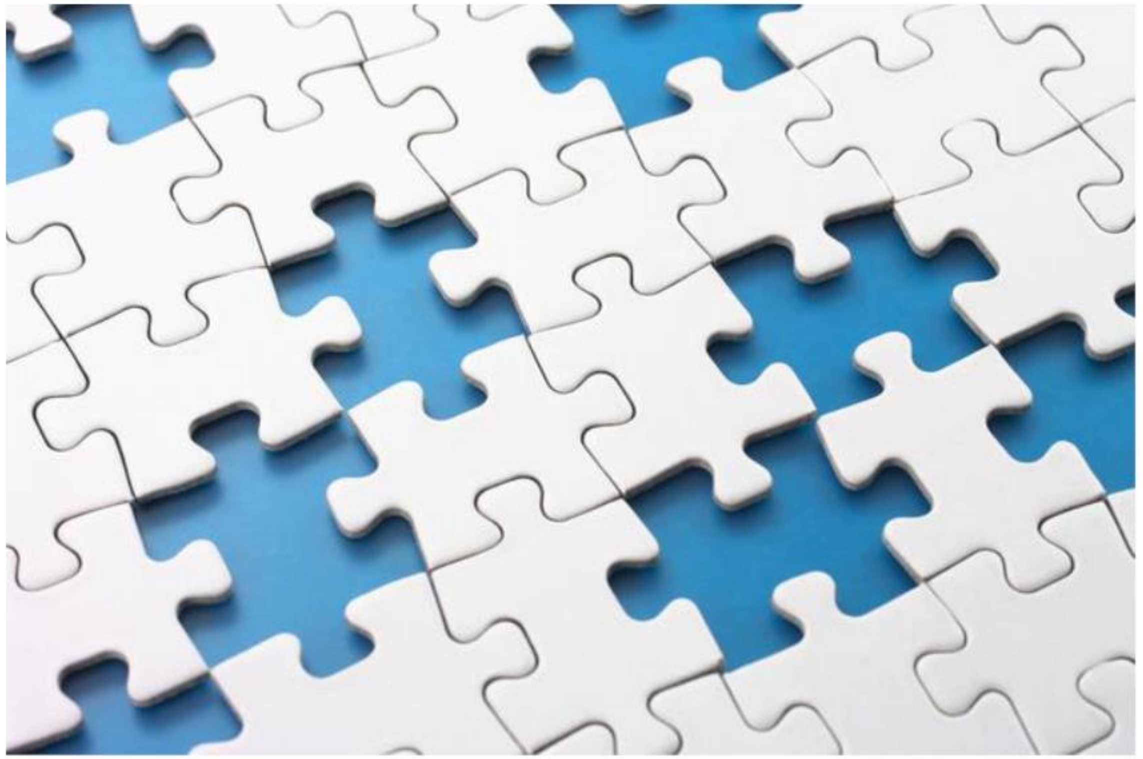
- To use testing to prove that a program works, you'd have to test every conceivable input values.
- Suppose, for example, that you have a program that takes a name, an address, and a phone number and stores them in a file.
- Each of the names and addresses is 20 characters long and that there are 26 possible characters to be used in them. This would be the number of possible inputs:



Name	26^{20} (20 characters, each with 26 possible choices)
Address	26^{20} (20 characters, each with 26 possible choices)
Phone Number	10^{10} (10 digits, each with 10 possible choices)
Total Possibilities	$= 26^{20} * 26^{20} * 10^{10} \approx 10^{66}$

Testing Tricks – *Incomplete Testing*

- Since exhaustive testing is impossible, practically speaking, the art of testing is that of
  - picking the test cases most likely to find errors.
- You need to concentrate on picking a few that tell you different things rather than a set that tells you the same thing over and over.



Testing Tricks – *Structured Basis Testing*

- **Test each statement** in a program **at least once**.
    - If the statement is a logical statement—an if or a while, <u>for example</u>—you need to vary the testing according to how complicated the expression inside the if or while is to make sure that the statement is fully tested.
- The easiest way to make sure that you've gotten all the bases covered is to calculate the number of paths through the program and then develop the <u>*minimum number of test cases that will exercise every path through the program*</u>.
    - Equivalence Behavior Testing?
    - Path Counting Problem?
    - NPATH Paper: https://dl.acm.org/citation.cfm?id=42379
    - Path Counting Toolbox: https://knowledgecentricsoftwarelab.com/PathCounter/

Testing Tricks – *Structured Basis Testing*

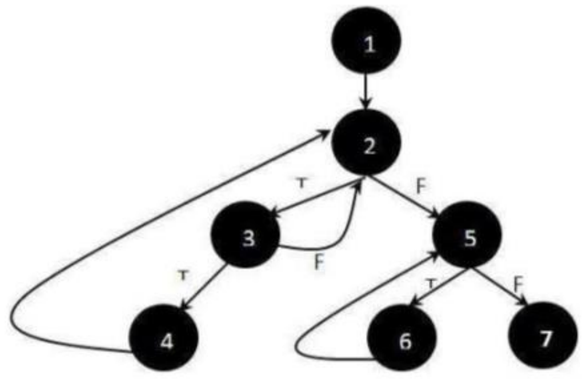
- **Basis path testing (structured testing)** is a white box method for designing test cases.
- The method analyzes the control flow graph of a program to find a set of linearly independent paths of execution.
- The method normally uses McCabe' cyclomatic complexity to determine the number of linearly independent paths and then generates test cases for each path thus obtained.
- Basis path testing guarantees complete branch coverage (all edges of the control flow graph), but achieves that without covering all possible paths of the control flow graph—the latter is usually too costly.
- Basis path testing has been widely used and studied

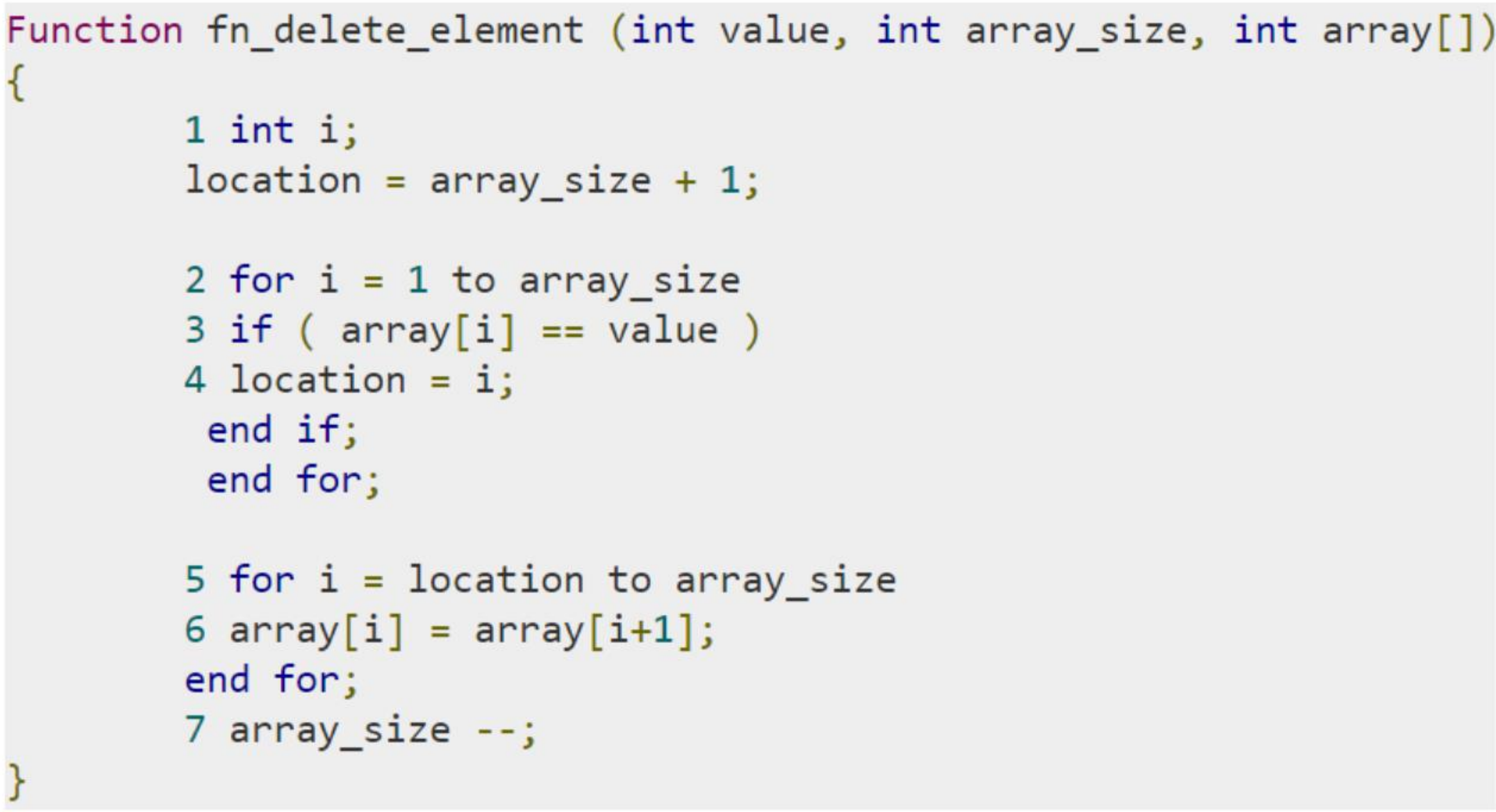
$V(G) = E - N + 2$: Where, E - Number of edges N - Number of Nodes
$V(G) = P + 1$: Where P = Number of predicate nodes (node that contains condition)

https://en.wikipedia.org/wiki/Basis_path_testing

Testing Tricks – *Structured Basis Testing*

- **Step 1 :** Draw the Flow Graph of the Function/Program under consideration as shown below:





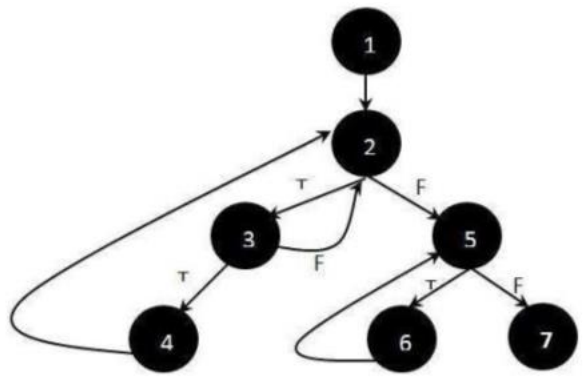
```
Function fn_delete_element (int value, int array_size, int array[])
{
        1 int i;
        location = array_size + 1;

        2 for i = 1 to array_size
        3 if ( array[i] == value )
        4 location = i;
         end if;
         end for;

        5 for i = location to array_size
        6 array[i] = array[i+1];
        end for;
        7 array_size --;
}
```

V(G) = E - N + 2: Where, E - Number of edges N - Number of Nodes
V(G) = P + 1: Where P = Number of predicate nodes (node that contains condition)

Testing Tricks – *Structured Basis Testing*

- **Step 2 :** Determine the independent paths.

```
Path 1:  1 - 2 - 5 - 7
Path 2:  1 - 2 - 5 - 6 - 7
Path 3:  1 - 2 - 3 - 2 - 5 - 6 - 7
Path 4:  1 - 2 - 3 - 4 - 2 - 5 - 6 - 7
```



```
Function fn_delete_element (int value, int array_size, int array[])
{
        1 int i;
        location = array_size + 1;

        2 for i = 1 to array_size
        3 if ( array[i] == value )
        4 location = i;
         end if;
         end for;

        5 for i = location to array_size
        6 array[i] = array[i+1];
        end for;
        7 array_size --;
}
```

V(G) = E - N + 2: Where, E - Number of edges N - Number of Nodes
V(G) = P + 1: Where P = Number of predicate nodes (node that contains condition)

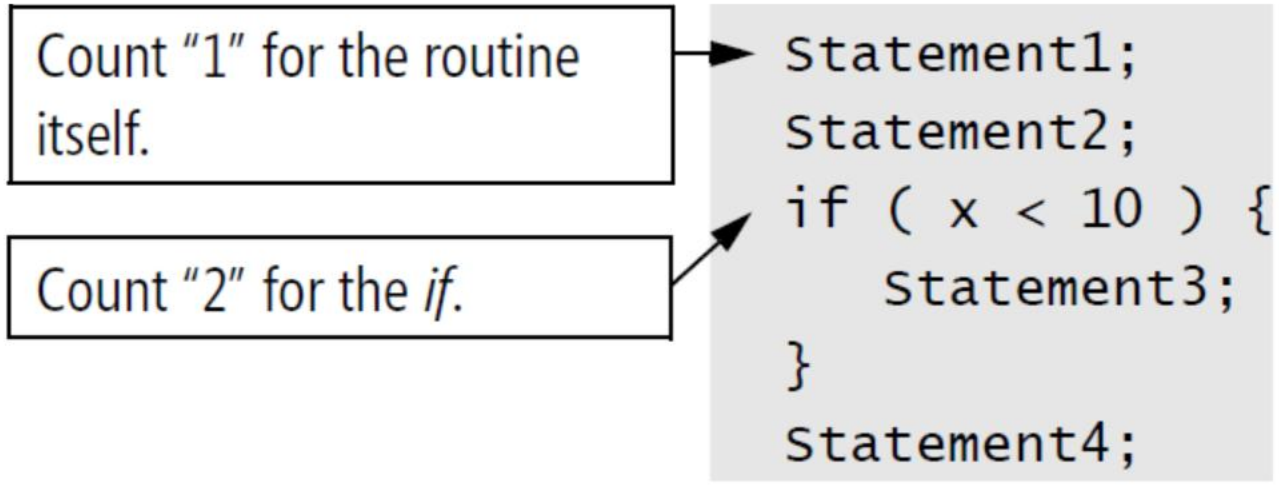
Testing Tricks – *Structured Basis Testing*

- You can compute the minimum number of cases needed for basis testing in this straightforward way:
  1. Start with 1 for the straight path through the routine.
  2. Add 1 for each of the following keywords, or their equivalents: if, case, while, for, and, and or, etc.

V(G) = E - N + 2: Where, E - Number of edges N - Number of Nodes
V(G) = P + 1: Where P = Number of predicate nodes (node that contains condition)

Testing Tricks – *Structured Basis Testing*

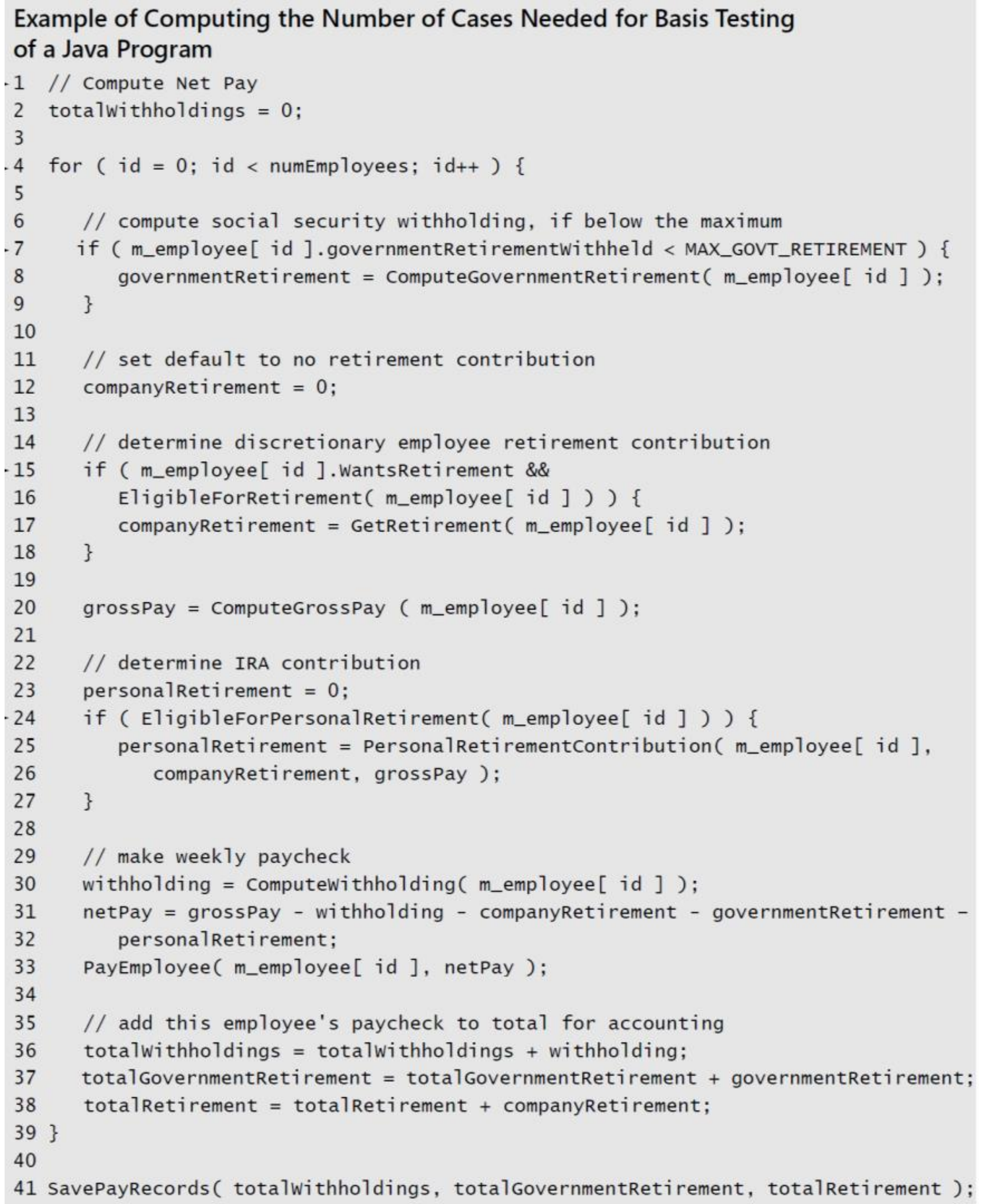
- Start with one and count the if once to make a total of two. So, you need to have at least two test cases to cover all the paths through the program.
- In this example, you'd need to have the following test cases:



```
Statement1;
Statement2;
if ( x < 10 ) {
   Statement3;
}
Statement4;
```

Statements controlled by *if* are executed (x < 10).

Statements controlled by *if* aren't executed (x >= 10)



Example of Computing the Number of Cases Needed for Basis Testing of a Java Program

```java
// Compute Net Pay
totalWithholdings = 0;

for ( id = 0; id < numEmployees; id++ ) {

  // compute social security withholding, if below the maximum
  if ( m_employee[ id ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT ) {
    governmentRetirement = ComputeGovernmentRetirement( m_employee[ id ] );
  }

  // set default to no retirement contribution
  companyRetirement = 0;

  // determine discretionary employee retirement contribution
  if ( m_employee[ id ].WantsRetirement &&
    EligibleForRetirement( m_employee[ id ] ) ) {
    companyRetirement = GetRetirement( m_employee[ id ] );
  }

  grossPay = ComputeGrossPay ( m_employee[ id ] );

  // determine IRA contribution
  personalRetirement = 0;
  if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
    personalRetirement = PersonalRetirementContribution( m_employee[ id ],
      companyRetirement, grossPay );
  }

  // make weekly paycheck
  withholding = ComputeWithholding( m_employee[ id ] );
  netPay = grossPay - withholding - companyRetirement - governmentRetirement –
    personalRetirement;
  PayEmployee( m_employee[ id ], netPay );

  // add this employee's paycheck to total for accounting
  totalWithholdings = totalWithholdings + withholding;
  totalGovernmentRetirement = totalGovernmentRetirement + governmentRetirement;
  totalRetirement = totalRetirement + companyRetirement;
}

SavePayRecords( totalWithholdings, totalGovernmentRetirement, totalRetirement );
```

Example of Computing the Number of Cases Needed for Basis Testing of a Java Program

Count "1" for the routine itself.

Count "2" for the *for*.

Count "3" for the *if*.

Count "4" for the *if* and "5" for the *&&*.

Count "6" for the *if*.

```
// Compute Net Pay
totalWithholdings = 0;

for ( id = 0; id < numEmployees; id++ ) {

   // compute social security withholding, if below the maximum
   if ( m_employee[ id ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT ) {
      governmentRetirement = ComputeGovernmentRetirement( m_employee[ id ] );
   }

   // set default to no retirement contribution
   companyRetirement = 0;

   // determine discretionary employee retirement contribution
   if ( m_employee[ id ].WantsRetirement &&
      EligibleForRetirement( m_employee[ id ] ) ) {
      companyRetirement = GetRetirement( m_employee[ id ] );
   }

   grossPay = ComputeGrossPay ( m_employee[ id ] );

   // determine IRA contribution
   personalRetirement = 0;
   if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
      personalRetirement = PersonalRetirementContribution( m_employee[ id ],
         companyRetirement, grossPay );
   }

   // make weekly paycheck
   withholding = ComputeWithholding( m_employee[ id ] );
   netPay = grossPay - withholding - companyRetirement - governmentRetirement –
      personalRetirement;
   PayEmployee( m_employee[ id ], netPay );

   // add this employee's paycheck to total for accounting
   totalWithholdings = totalWithholdings + withholding;
   totalGovernmentRetirement = totalGovernmentRetirement + governmentRetirement;
   totalRetirement = totalRetirement + companyRetirement;
}

SavePayRecords( totalWithholdings, totalGovernmentRetirement, totalRetirement );
```

Case	Test Description	Test Data
1	Nominal case	All boolean conditions are true
2	The initial *for* condition is false	*numEmployees < 1*
3	The first *if* is false	*m_employee[id].governmentRetirementWithheld >=MAX_GOVT_RETIREMENT*
4	The second *if* is false because the first part of the *and* is false	*not m_employee[id].WantsRetirement*
5	The second *if* is false because the second part of the *and* is false	*not EligibleForRetirement(m_employee[id])*
6	The third *if* is false	*not EligibleForPersonalRetirement(m_employee[id])*

Note: This table will be extended with additional test cases throughout the chapter.

Testing Tricks – Structured Basis Testing

- If the routine were much more complicated than this, the number of test cases you'd have to use just to cover all the paths would increase pretty quickly.
- Shorter routines tend to have fewer paths to test. Boolean expressions without a lot of ands and ors have fewer variations to test. Ease of testing is another good reason to keep your routines short and your boolean expressions simple.
- Now that you've created six test cases for the routine and satisfied the demands of structured basis testing, can you consider the routine to be fully tested? *Probably not*.

Testing Tricks – Data-Flow Testing

- **Data-flow testing** is a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects.
  - focuses on the points at which variables receive values and the points at which these values are used.
- Data Flow testing helps us to pinpoint any of the following issues:
  - A variable that is declared but never used within the program.
  - A variable that is used but never declared.
  - A variable that is defined multiple times before it is used.
  - Deallocating a variable before it is used.

Testing Tricks – Data-Flow Testing

- **Data-flow testing** is based on the idea that data usage is at least as error-prone as control flow.
- Data can exist in one of three states:
  - Defined The data has been initialized, but it hasn't been used yet.
  - Used The data has been used for computation, as an argument to a routine, or for something else.
  - Killed The data was once defined, but it has been undefined in some way.
    - For example, if the data is a pointer, perhaps the pointer has been freed. If it's a for-loop index, perhaps the program is out of the loop and the programming language doesn't define the value of a for-loop index once it's outside the loop.

Testing Tricks – Data-Flow Testing

- In addition to having the terms “defined,” “used,” and “killed,” it’s convenient to have terms that describe entering or exiting a routine immediately before or after doing something to a variable:
  - Entered The control flow enters the routine immediately before the variable is acted upon. <u>For example</u>, a working variable is initialized at the top of a routine.
  - Exited The control flow leaves the routine immediately after the variable is acted upon. <u>For example</u>, a return value is assigned to a status variable at the end of a routine.

Testing Tricks – Data-Flow Testing

- **Combinations of Data States**
- The normal combination of data states is that a variable is defined, used one or more times, and perhaps killed. View the following patterns suspiciously:
  - Defined-Defined
  - Defined-Exited
  - Defined-Killed
  - Entered-Killed
  - Entered-Used
  - Killed-Killed
  - Killed-Used
  - Used-Defined

Testing Tricks – Data-Flow Testing

- **Combinations of Data States**
- **Defined-Defined**
- Define a variable twice before the value is set to it

Testing Tricks – Data-Flow Testing

- **Combinations of Data States**
- **Defined-Exited**
- If the variable is a local variable, it doesn’t make sense to define it and exit without using it.

Testing Tricks – Data-Flow Testing

- **Combinations of Data States**
- **Defined-Killed**
- Defining a variable and then killing it suggests that either the variable is not-essential or the code piece that was supposed to use the variable is missing.

Testing Tricks – Data-Flow Testing

- **Combinations of Data States**
- **Entered-Killed**
- This is a problem if the variable is a local variable. It wouldn’t need to be killed if it hasn’t been defined or used.
- If, on the other hand, it’s a routine parameter or a global variable, this pattern is all right as long as the variable is defined somewhere else before it’s killed.

Testing Tricks – Data-Flow Testing

- **Combinations of Data States**
- **Entered-Used**
- Again, this is a problem if the variable is a local variable.
- The variable needs to be defined before it's used.
- If, on the other hand, it's a routine parameter or a global variable, the pattern is all right if the variable is defined somewhere else before it's used.

Testing Tricks – Data-Flow Testing

- **Combinations of Data States**
- **Killed-Killed**
- A variable shouldn’t need to be killed twice. Variables don’t come back to life.
- A resurrected variable indicates sloppy programming.
- Double kills are also fatal for pointers

Testing Tricks – Data-Flow Testing

- **Combinations of Data States**
- **Killed-Used**
- Using a variable after it has been killed is a logical error.

Testing Tricks – Data-Flow Testing

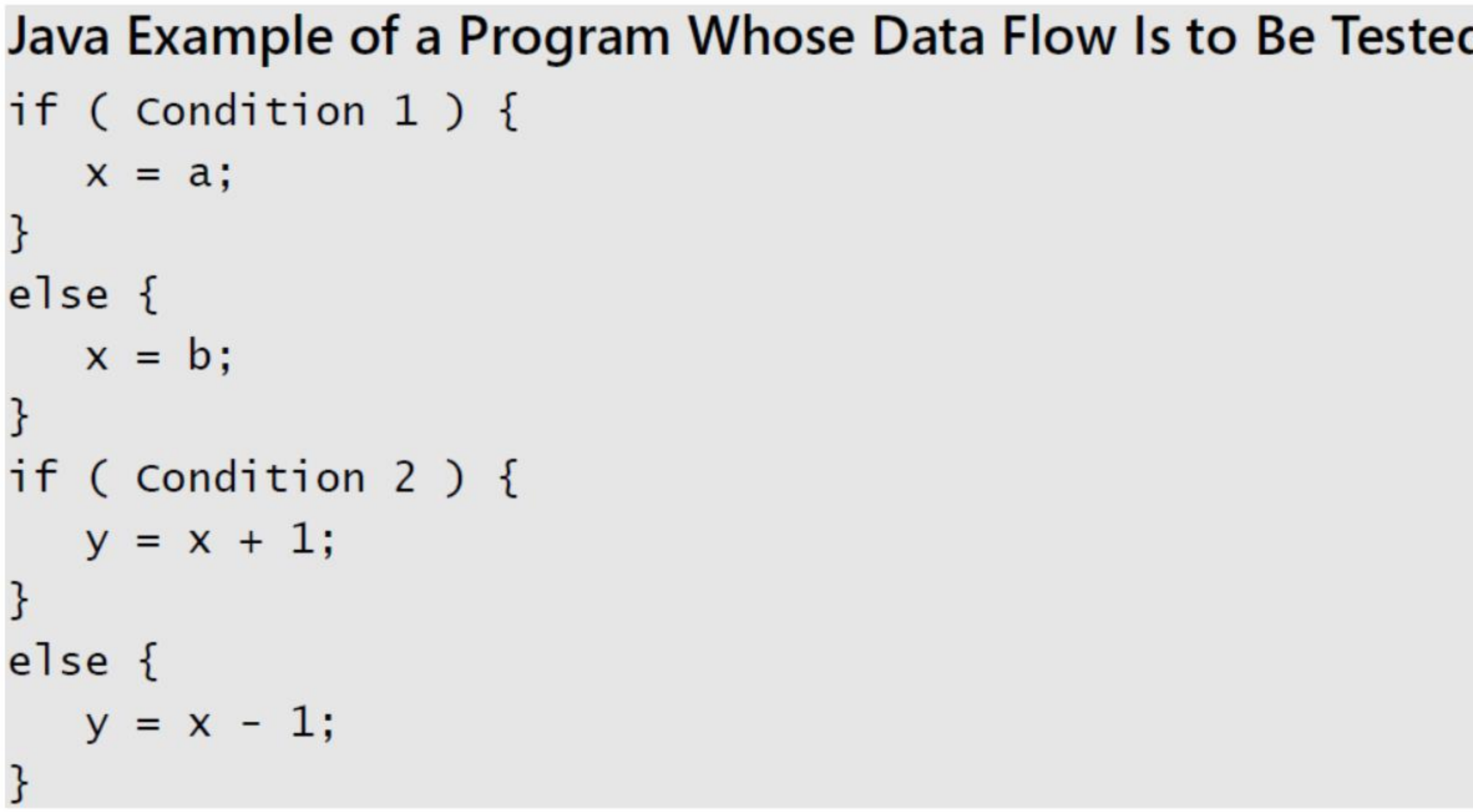
- **Combinations of Data States**
- **Used-Defined**
- Using and then defining a variable might or might not be a problem, depending on whether the variable was also defined before it was used.
- Certainly if you see a used-defined pattern, it’s worthwhile to check for a previous definition.

Testing Tricks – Data-Flow Testing Combinations of Data States

- Check for these anomalous sequences of data states before testing begins.
- After you've checked for the anomalous sequences, the key to writing data-flow test cases is to exercise all possible defined-used paths.
- You can do this to various degrees of thoroughness, including
  - All definitions. Test every definition of every variable—that is, every place at which any variable receives a value. This is a weak strategy because if you try to exercise every line of code, you'll do this by default.
  - All **defined-used** combinations. Test every combination of defining a variable in one place and using it in another. *This is a* stronger strategy *than testing all definitions because merely executing every line of code does not guarantee that every defined-used combination will be tested*.

Testing Tricks – Data-Flow Testing Combinations of Data States

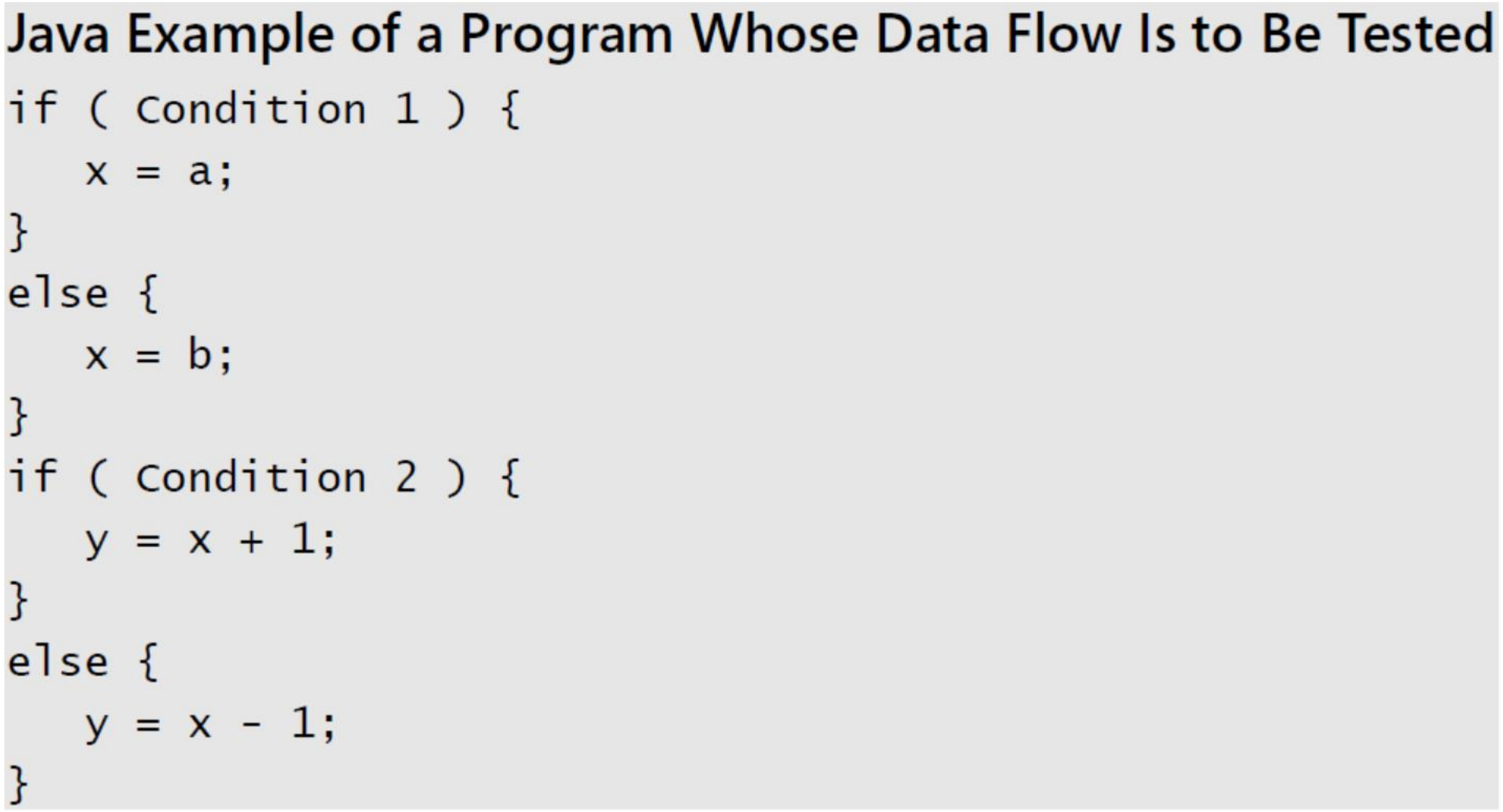
- **What are the test cases for basis testing?**



Java Example of a Program Whose Data Flow Is to Be Testec

```
if ( Condition 1 ) {
   x = a;
}
else {
   x = b;
}
if ( Condition 2 ) {
   y = x + 1;
}
else {
   y = x - 1;
}
```

Testing Tricks – Data-Flow Testing Combinations of Data States



Java Example of a Program Whose Data Flow Is to Be Tested

```
if ( Condition 1 ) {
   x = a;
}
else {
   x = b;
}
if ( Condition 2 ) {
   y = x + 1;
}
else {
   y = x - 1;
}
```

Testing Tricks – Data-Flow Testing Combinations of Data States

- **To cover every path in the program**, you need one test case in which Condition 1 is true and one in which it’s false.
- You also need a test case in which Condition 2 is true and one in which it’s false.
- This can be handled by two test cases: Case 1 (Condition 1=True, Condition 2=True) and Case 2 (Condition 1=False, Condition 2=False).
  - *Those two cases are all you need for structured* **basis testing**.
  - They’re also all you need to exercise every line of code that defines a variable; they give you the weak form of data-flow testing automatically.

Testing Tricks – Data-Flow Testing Combinations of Data States

- To cover every defined-used combination, however, you need to add a few more cases.
- Right now you have the cases created by having Condition 1 and Condition 2 true at the same time and Condition 1 and Condition 2 false at the same time.
- But you need two more cases to test every defined-used combination: (1) x = a and then y = x - 1 and (2) x = b and then y = x + 1. In this example, you can get these combinations by adding two more cases: Case 3 (Condition 1=True, Condition 2=False) and Case 4 (Condition 1=False, Condition 2=True).

```
x = a
...
y = x + 1
```

and

```
x = b
...
y = x - 1
```

Testing Tricks – Data-Flow Testing Combinations of Data States

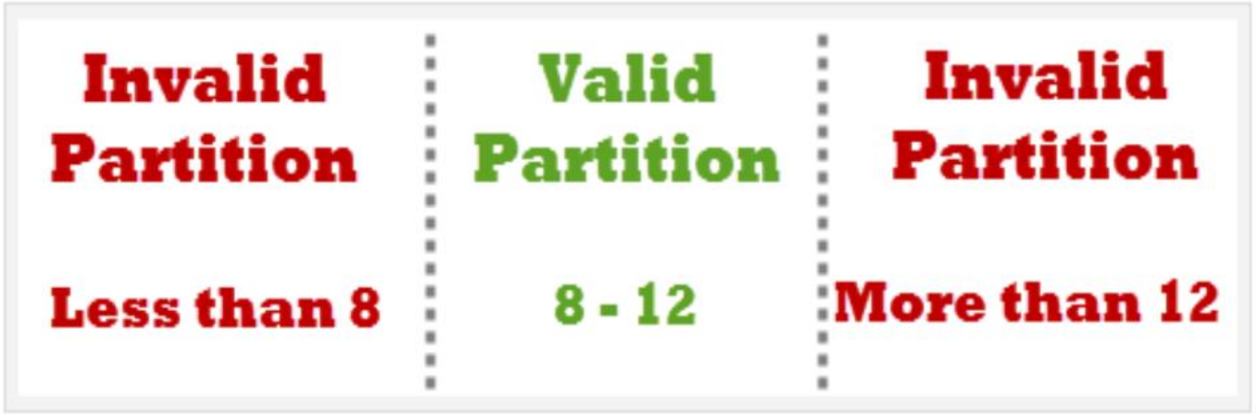
- A good way to develop test cases is to start with structured basis testing, which gives you some if not all of the defined-used data flows.
- Then add the cases you still need to have a complete set of **defined-used data-flow test cases**.

Testing Tricks – Data-Flow Testing Equivalence Partitioning

- A good test case covers a large part of the possible input data.
- *If two test cases flush out exactly the same errors, you need only one of them*.
- The concept of "equivalence partitioning" is a formalization of this idea and helps reduce the number of test cases required.
  - is a software testing technique that divides the input data of a software unit into **partitions** of **equivalent data** from which test cases can be derived.
  - In principle, test cases are designed to cover each partition **at least once**.

Testing Tricks – Data-Flow Testing Equivalence Partitioning

- ***Password***: must be a minimum 8 characters and maximum 12 characters.



Invalid Partition	Valid Partition	Invalid Partition
Less than 8	8 - 12	More than 12

- **What are the test cases?**

Testing Tricks – Data-Flow Testing Equivalence Partitioning

- ***Password***: must be a minimum 8 characters and maximum 12 characters.
    - **Test Case 1**: Consider password length less than 8.
    - **Test Case 2**: Consider password of length exactly 8.
    - **Test Case 3**: Consider password of length between 9 and 11.
    - **Test Case 4**: Consider password of length exactly 12.
    - **Test Case 5**: Consider password of length more than 12.

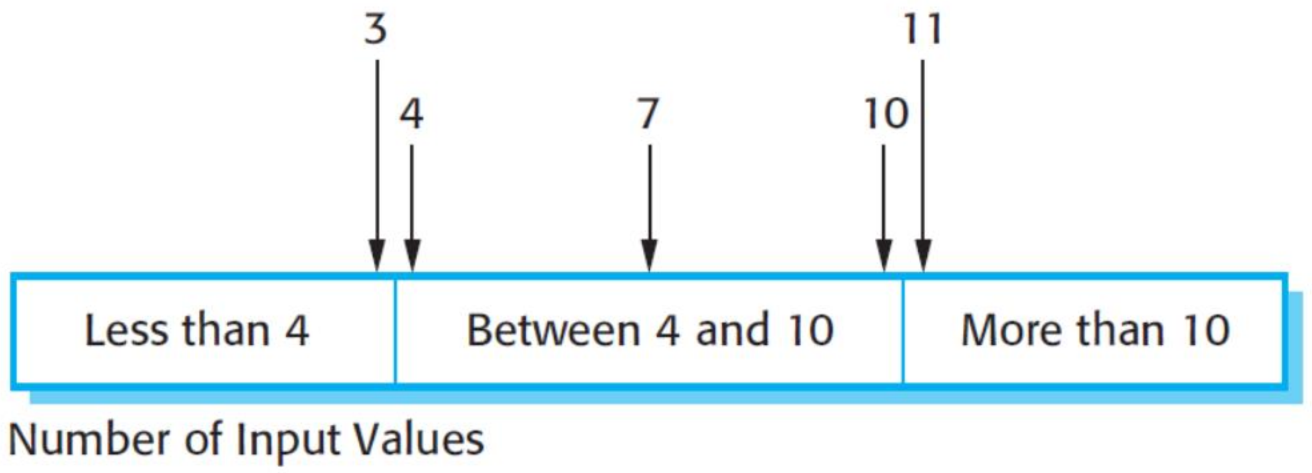
Testing Tricks – Data-Flow Testing Equivalence Partitioning

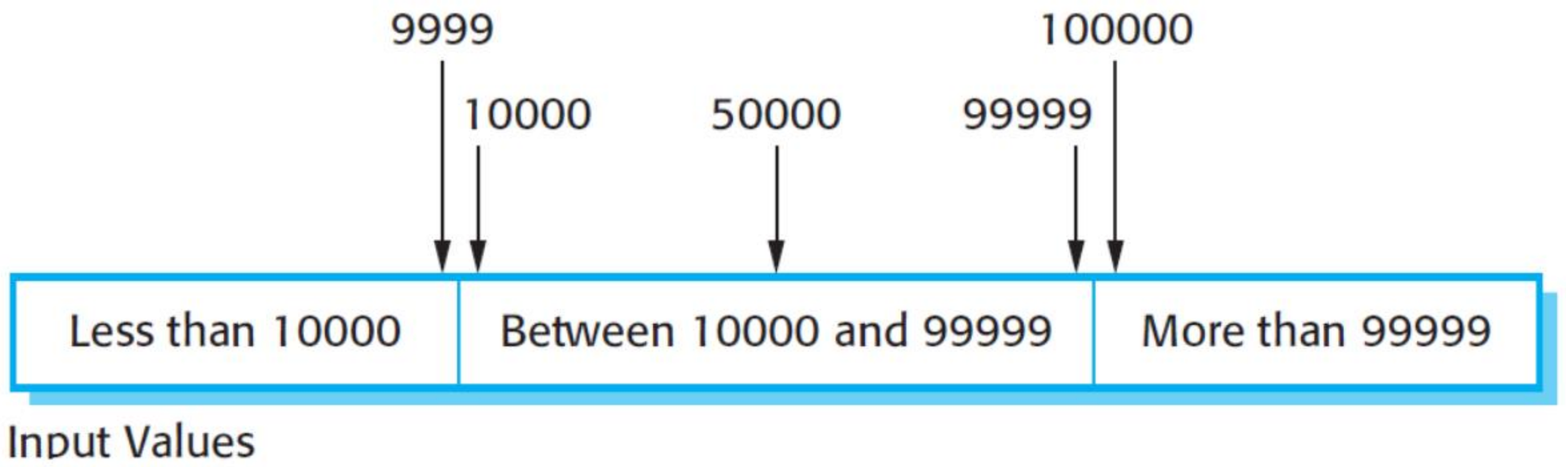
- Once you have identified a set of partitions, you choose test cases from each of these partitions.
- A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition.
- The reason for this is that designers and programmers tend to consider typical values of inputs when developing a system. You test these by choosing the midpoint of the partition.
- Boundary values are often atypical (e.g., zero may behave differently from other non-negative numbers) so are sometimes overlooked by developers.
- Program failures often occur when processing these atypical values.

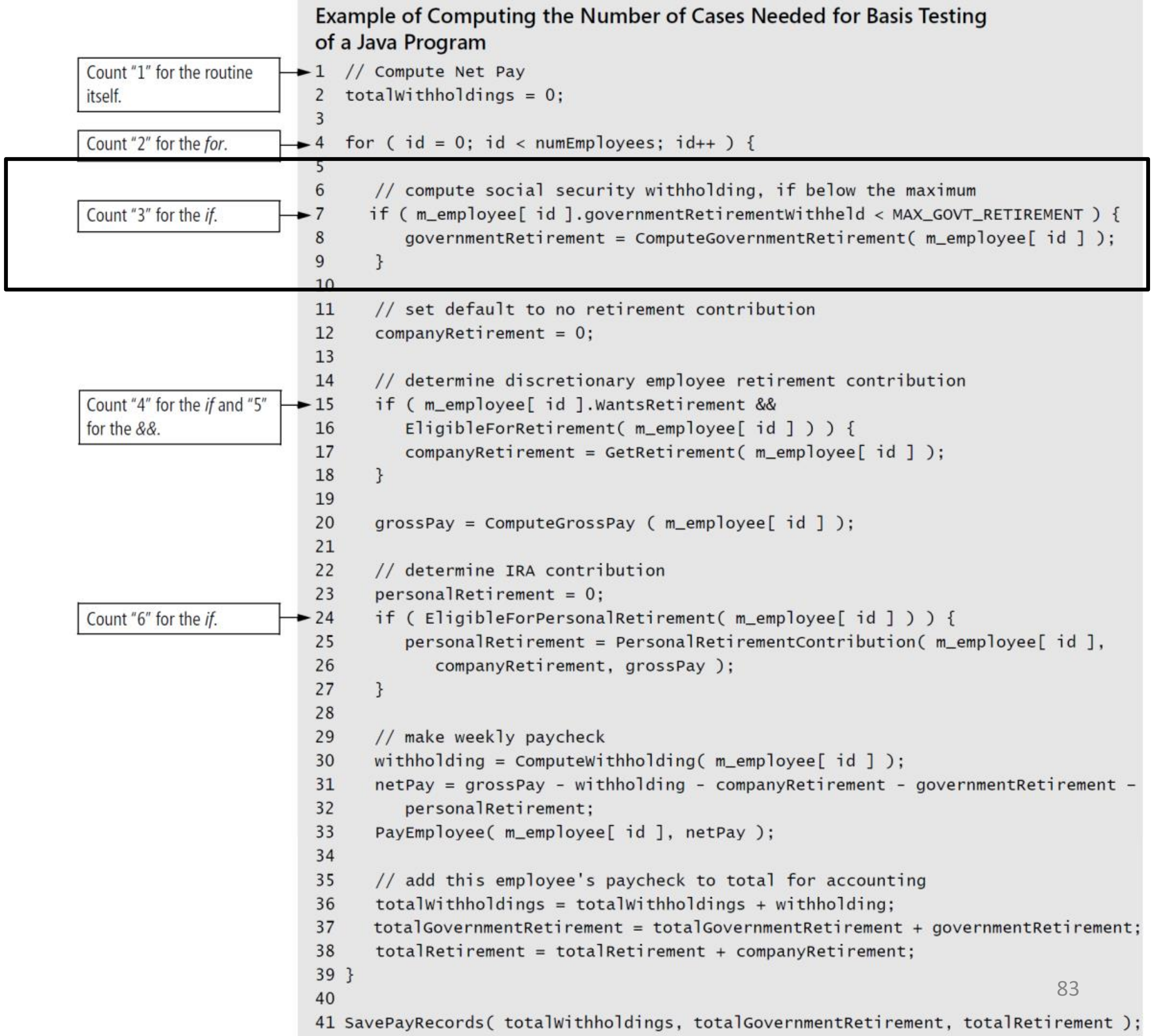
Testing Tricks – Data-Flow Testing Equivalence Partitioning

- ***Equivalence partitions***

as shown here:







Example of Computing the Number of Cases Needed for Basis Testing of a Java Program

Count "1" for the routine itself.

Count "2" for the *for*.

Count "3" for the *if*.

Count "4" for the *if* and "5" for the *&&*.

Count "6" for the *if*.

```
// Compute Net Pay
totalWithholdings = 0;

for ( id = 0; id < numEmployees; id++ ) {

   // compute social security withholding, if below the maximum
   if ( m_employee[ id ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT ) {
      governmentRetirement = ComputeGovernmentRetirement( m_employee[ id ] );
   }

   // set default to no retirement contribution
   companyRetirement = 0;

   // determine discretionary employee retirement contribution
   if ( m_employee[ id ].WantsRetirement &&
      EligibleForRetirement( m_employee[ id ] ) ) {
      companyRetirement = GetRetirement( m_employee[ id ] );
   }

   grossPay = ComputeGrossPay ( m_employee[ id ] );

   // determine IRA contribution
   personalRetirement = 0;
   if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
      personalRetirement = PersonalRetirementContribution( m_employee[ id ],
         companyRetirement, grossPay );
   }

   // make weekly paycheck
   withholding = ComputeWithholding( m_employee[ id ] );
   netPay = grossPay - withholding - companyRetirement - governmentRetirement –
      personalRetirement;
   PayEmployee( m_employee[ id ], netPay );

   // add this employee's paycheck to total for accounting
   totalWithholdings = totalWithholdings + withholding;
   totalGovernmentRetirement = totalGovernmentRetirement + governmentRetirement;
   totalRetirement = totalRetirement + companyRetirement;
}

SavePayRecords( totalWithholdings, totalGovernmentRetirement, totalRetirement );
```

Testing Tricks – Data-Flow Testing Equivalence Partitioning

- The condition to be tested is m_employee[ID].governmentRetirementWithheld < MAX_GOVT_RETIREMENT.
- This case has two equivalence classes:
    - the class in which m_employee[ID].governmentRetirementWithheld is less than MAX_GOVT_RETIREMENT and the class in which it's greater than or equal to MAX_GOVT_RETIREMENT.
- Other parts of the program might have other related equivalence classes that imply that you need to test more than two possible values of m_employee[ID].governmentRetirementWithheld, but as far as this part of the program is concerned, only two are needed.

Testing Tricks – Data-Flow Testing Equivalence Partitioning

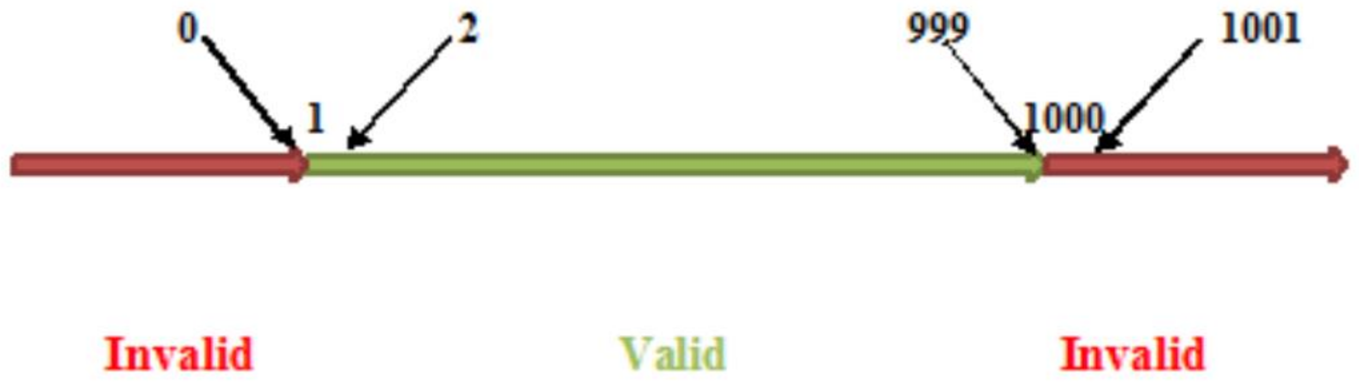
- Thinking about equivalence partitioning won’t give you a lot of new insight into a program when you have already covered the program with basis and data-flow testing.
- It’s especially helpful, however, when you’re looking at a program from the outside (from a specification rather than the source code) or when the data is complicated and the complications aren’t all reflected in the program’s logic.

Testing Tricks – Data-Flow Testing
Error Guessing

- In addition to the formal test techniques, good programmers use a variety of less formal, heuristic techniques to expose errors in their code.
- One heuristic is the technique of error guessing.
- It means creating test cases based upon guesses about where the program might have errors, although it implies a certain amount of sophistication in the guessing.

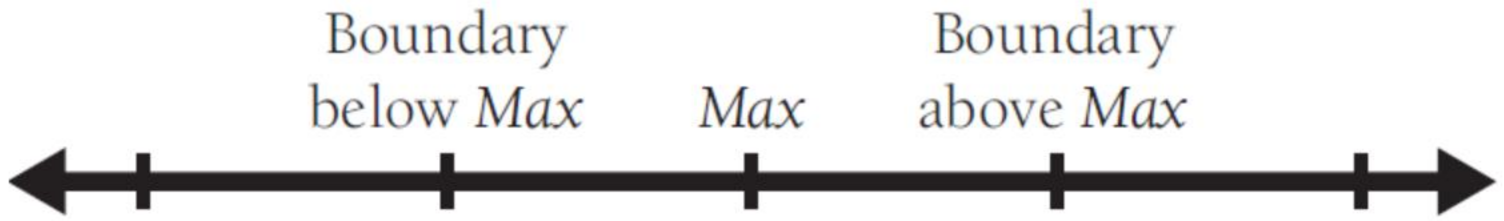
Testing Tricks – Data-Flow Testing Boundary Analysis

- One of the most fruitful areas for testing is boundary conditions—off-by-one errors.
- Saying num – 1 when you mean num and saying >= when you mean > are common mistakes.



Testing Tricks – Data-Flow Testing Boundary Analysis

- The idea of boundary analysis is to write test cases that exercise the boundary conditions.
- If you’re testing for a range of values that are less than max, you have three possible conditions:



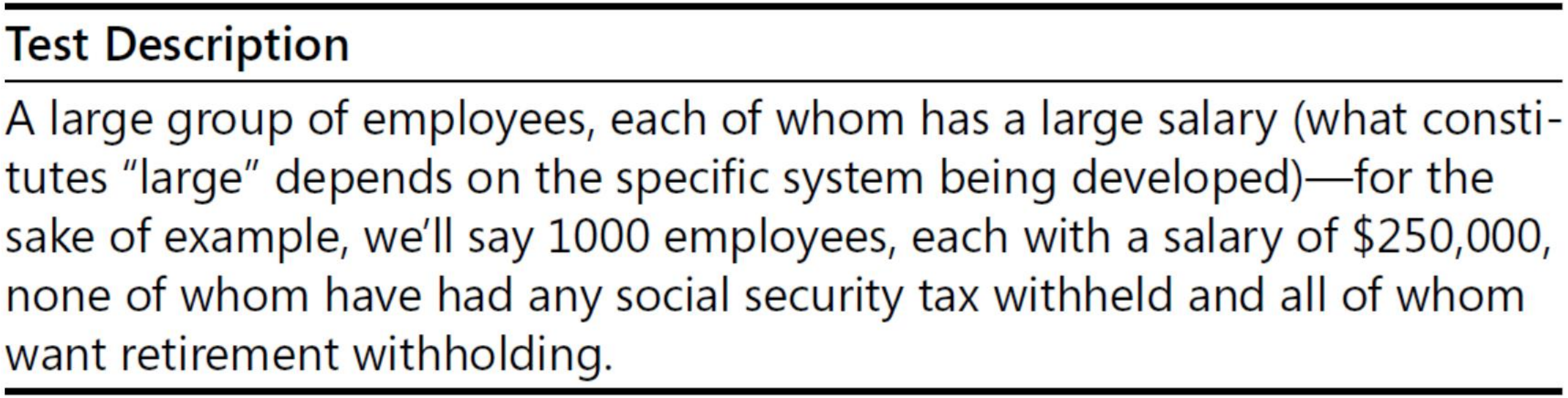
- As shown, there are three boundary cases: just less than max, max itself, and just greater than max. It takes three cases to ensure that none of the common mistakes has been made.

Testing Tricks – Data-Flow Testing Compound Boundaries

- A more complicated kind of boundary condition occurs when the boundary involves a combination of variables.
- For example, if two variables are multiplied together, what happens when both are large positive numbers? Large negative numbers? 0? What if all the strings passed to a routine are uncommonly long?

Testing Tricks – Data-Flow Testing Compound Boundaries

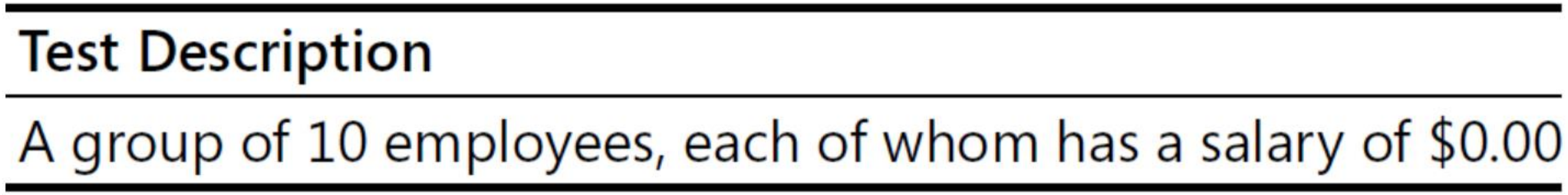
- In the running <u>example</u>, you might want to see what happens to the variables totalWithholdings, totalGovernmentRetirement, and totalRetirement when every member of a large group of employees has a large salary—say, a group of programmers at $250,000 each.



Test Description
A large group of employees, each of whom has a large salary (what constitutes "large" depends on the specific system being developed)—for the sake of example, we'll say 1000 employees, each with a salary of $250,000, none of whom have had any social security tax withheld and all of whom want retirement withholding.

Testing Tricks – Data-Flow Testing Compound Boundaries

- A test case in the same vein but on the opposite side of the looking glass would be a small group of employees, each of whom has a salary of $0.00:



Test Description
A group of 10 employees, each of whom has a salary of $0.00

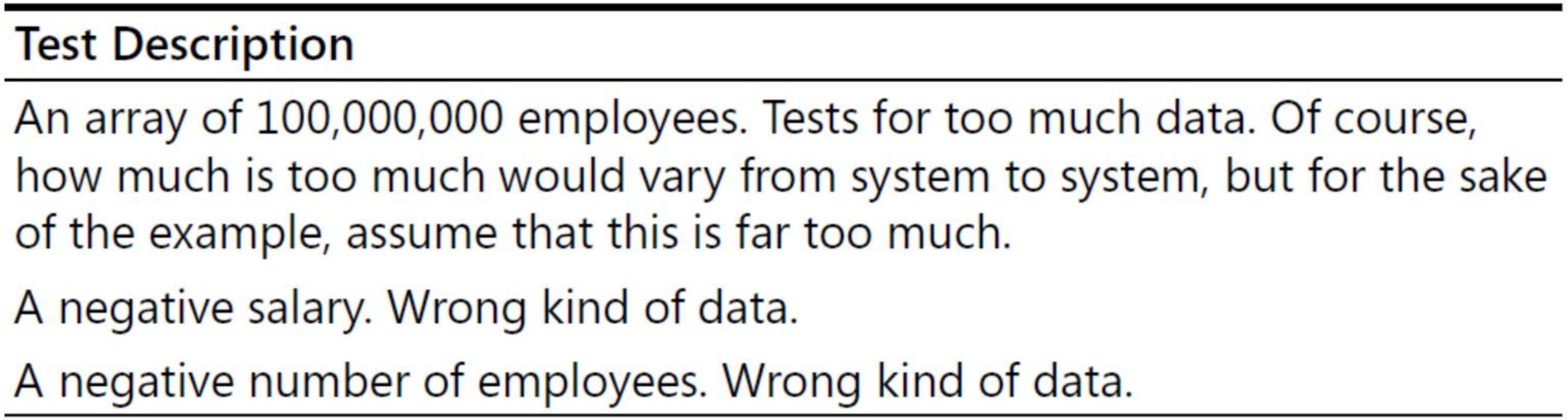
Testing Tricks – Data-Flow Testing Classes of Bad Data

- Aside from guessing that errors show up around boundary conditions, you can guess about and test for several other classes of bad data.
- Typical bad-data test cases include
  - Too little data (or no data)
  - Too much data
  - The wrong kind of data (invalid data)
  - The wrong size data
  - Uninitialized data

Testing Tricks – Data-Flow Testing Classes of Bad Data

- Some of the test cases you would think of if you followed these suggestions have already been covered.

- Classes of bad data nonetheless gives rise to a few more cases:



Test Description
An array of 100,000,000 employees. Tests for too much data. Of course, how much is too much would vary from system to system, but for the sake of the example, assume that this is far too much.
A negative salary. Wrong kind of data.
A negative number of employees. Wrong kind of data.

Testing Tricks – Data-Flow Testing Classes of Good Data

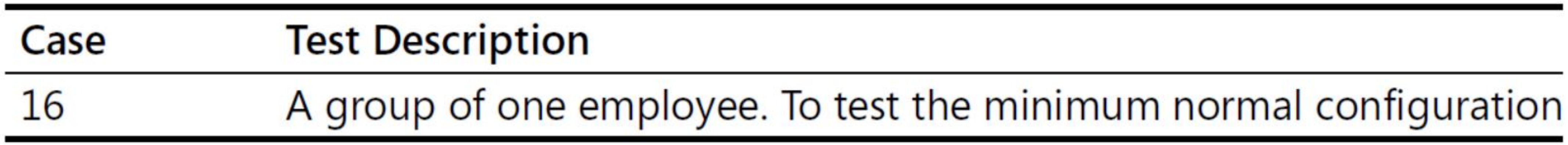
- When you try to find errors in a program, it’s easy to miss the fact that the main case might contain an error.
- Usually the nominal cases described in the basis-testing section represent one kind of good data.
- Following are other kinds of good data that are worth checking.
- Checking each of these kinds of data can reveal errors, depending on the item being tested.

Testing Tricks – Data-Flow Testing Classes of Good Data

- Checking each of these kinds of data can reveal errors, depending on the item being tested.
    - Nominal cases—middle-of-the-road, expected values
    - Minimum normal configuration
    - Maximum normal configuration
    - Compatibility with old data

Testing Tricks – Data-Flow Testing
Classes of Good Data

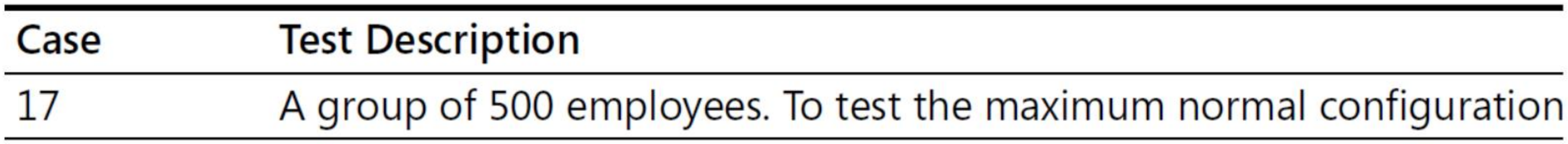
- The minimum normal configuration is useful for testing not just one item, but a group of items.
- *It's similar in spirit to the boundary condition of many minimal values*, but it's different in that it creates the set of minimum values out of the set of what is normally expected.
- One example would be to save an empty spreadsheet when testing a spreadsheet.
- In the case of the running example, testing the minimum normal configuration would add the following test case:



Case	Test Description
16	A group of one employee. To test the minimum normal configuration

Testing Tricks – Data-Flow Testing
Classes of Good Data

- The maximum normal configuration is the opposite of the minimum.
- *It's similar in spirit to boundary testing*, but again, it creates a set of maximum values out of the set of expected values.
- An example of this would be saving a spreadsheet that's as large as the "maximum spreadsheet size" advertised on the product's packaging.
- In the case of the running example, testing the maximum normal configuration depends on the maximum normal number of employees, assuming it's 500, you would add the following test case:



Case	Test Description
17	A group of 500 employees. To test the maximum normal configuration

Testing Tricks – Data-Flow Testing
Use Test Cases That Make Hand-Checks Convenient

- Let’s suppose you’re writing a test case for a nominal salary; you need a nominal salary, and the way you get one is to type in whatever numbers your hands land on.
    - Testing $90,783.82
- Now, further suppose that the test case succeeds—that is, it finds an error. How do you know that it’s found an error? Well, presumably, you know what the answer is and what it should be because you calculated the correct answer by hand.
- Better to use numbers that can be hand checked, e.g. $90,000

Typical Errors - Which Classes Contain the Most Errors?

- It’s natural to assume that defects are distributed evenly throughout your source code.
- If you have an average of 10 defects per 1000 lines of code, you might assume that you’ll have one defect in a class that contains 100 lines of code.
- This is a natural assumption, but it’s **wrong**.

Typical Errors - Which Classes Contain the Most Errors?

- It was reported that a program at IBM identified 31 of 425 classes are error-prone.
- The 31 classes were repaired or completely redeveloped, and, in less than a year, customer-reported defects were reduced ten to one.
- Total maintenance costs were reduced by about 45 percent.
- *Customer satisfaction improved from "unacceptable" to "good"* (Jones 2000).

Typical Errors - Which Classes Contain the Most Errors?

- Most errors tend to be concentrated in a few highly defective routines. Here is the general relationship between errors and code:
    - 80% of the errors are found in 20%of a project’s classes or routines (Endres 1975, Gremillion 1984, Boehm 1987b, Shull et al 2002).
    - 50% of the errors are found in 5% of a project’s classes (Jones 2000).

Typical Errors - Which Classes Contain the Most Errors?

- *These relationships might not seem so important until you recognize a few conclusions*.
- **First**, 20% of a project's routines contribute 80% of the cost of development (Boehm 1987b). *That doesn't necessarily mean that the 20% that cost the most are the same as the 20% with the most defects, but it's pretty suggestive*.
- **Second**, regardless of the exact proportion of the cost contributed by highly defective routines, highly defective routines are extremely expensive.

Typical Errors - Which Classes Contain the Most Errors?

- In a classic study in the 1960s, IBM performed an analysis of its OS/360 operating system and found that errors were not distributed evenly across all routines but were concentrated into a few.
- Those error-prone routines were found to be "the most expensive entities in programming" (Jones 1986a).
- They contained as many as 50 defects per 1000 lines of code, and fixing them often cost 10 times what it took to develop the whole system (The costs included customer support and in-the-field maintenance.)

Typical Errors - Which Classes Contain the Most Errors?

- **Third**, the implication of expensive routines for development is clear.
- If you can cut close to 80% of the cost by avoiding troublesome routines, you can cut a substantial amount of the schedule as well.
- This is a clear illustration of the General Principle of Software Quality: *improving quality improves the development schedule and reduces development costs.*

Typical Errors - Which Classes Contain the Most Errors?

- **Fourth**, the implication of avoiding troublesome routines for maintenance is equally clear.
- Maintenance activities should be focused on identifying, redesigning, and rewriting from the ground up those routines that have been identified as error-prone.
- In the IBM project mentioned earlier, productivity of the product releases improved about 15% after replacement of the error-prone classes (Jones 2000).

Debugging

Debugging Issues

Finding a Defect

Fixing a Defect

Psychological Considerations

Debugging Tools

Overview of Debugging Issues

- **Debugging** is the process of identifying the root cause of an error and correcting it.
- It contrasts with testing, which is the process of detecting the error initially. On some projects, debugging occupies as much as 50 percent of the total development time.
- For many programmers, debugging is the hardest part of programming.

Outline

- Debugging Issues
- Finding a Defect
- Fixing a Defect
- Psychological Considerations
- Debugging Tools

Overview of Debugging Issues

- A **bug** in software means that a programmer made a mistake.
- Like testing, debugging isn’t a way to improve the quality of your software; it’s a way to diagnose defects. Software quality must be built in from the start.
- The best way to build a quality product is to develop requirements carefully, design well, and use high-quality coding practices.
- Debugging is a last resort.

Overview of Debugging Issues - Defects

- Assuming that you don’t want the program to have a defect, it means that you don’t fully understand what the program does.
- If you don’t know exactly what you’re telling the computer to do, you’re only a small step away from merely trying different things until something seems to work—that is, programming by trial and error. And if you’re programming by trial and error, defects are guaranteed.
- You don’t need to learn how to fix defects; you need to learn how to avoid them in the first place.

Example: the GNU Debugger (GDB)

http://www.cprogramming.com/gdb.html

- A good debugger is one of the most important tools in a programmer's toolkit.
- On a UNIX or Linux system, GDB (the GNU debugger) is a powerful and popular debugging tool; it lets you do whatever you like with your program running under GDB.

Overview of Debugging Issues - Defects

- ***Learn about the program you're working on***
- You have something to learn about the program because if you already knew it perfectly, it wouldn't have a defect. You would have corrected it already.

Overview of Debugging Issues - Defects

- ***Learn about the kinds of mistakes you make***
- If you wrote the program, you inserted the defect. It's not every day that a spotlight exposes a weakness with glaring clarity, but such a day is an opportunity, so take advantage of it.
- Once you find the mistake, ask yourself how and why you made it.
    - How could you have found it more quickly?
    - How could you have prevented it?
    - Does the code have other mistakes just like it?
    - Can you correct them before they cause problems of their own?

Overview of Debugging Issues - Defects

- ***Learn about the quality of your code from the point of view of someone who has to read it***
- You'll have to read your code to find the defect.
- This is an opportunity to look critically at the quality of your code.
  - Is it easy to read? How could it be better?
- Use your discoveries to refactor your current code or to improve the code you write next.

Overview of Debugging Issues - Defects

- ***Learn about how you solve problems***
- Does your approach to solving debugging problems give you confidence? Does your approach work? Do you find defects quickly? Or is your approach to debugging weak? Do you feel anguish and frustration? Do you guess randomly? Do you need to improve?
- Considering the amount of time many projects spend on debugging, you definitely won't waste time if you observe how you debug.
- Taking time to analyze and change the way you debug might be the quickest way to decrease the total amount of time it takes you to develop a program.

Overview of Debugging Issues - Defects

- ***Learn about how you fix defects***
- In addition to learning how you find defects, you can learn about how you fix them.
- Do you make the easiest possible correction by applying *goto* bandages and special-case makeup that changes the symptom but not the problem? Or do you make systemic corrections, demanding an accurate diagnosis and prescribing treatment for the heart of the problem?

Overview of Debugging Issues - An Ineffective Approach

- ***Find the defect by guessing***
- To find the defect, scatter print statements randomly throughout a program.
- Examine the output to see where the defect is. If you can't find the defect with print statements, try changing things in the program until something seems to work.
- Don't back up the original version of the program, and don't keep a record of the changes you've made.
- Programming is more exciting when you're not quite sure what the program is doing. Stock up on cola and candy because you're in for a long night in front of the terminal.

Overview of Debugging Issues - An Ineffective Approach

- ***Fix the error with the most obvious fix***
- It’s usually good just to fix the specific problem you see, rather than wasting a lot of time making some big, ambitious correction that’s going to affect the whole program.
- This is a perfect example:

```
x = Compute( y )
if ( y = 17 )
   x = $25.15        -- Compute() doesn't work for y = 17, so fix it
```

- Who needs to dig all the way into *Compute()* for an obscure problem with the value of *17* when you can just write a special case for it in the obvious place?

Overview of Debugging Issues - An Ineffective Approach

- ***Don't waste time trying to understand the problem***
- It's likely that the problem is trivial, and you don't need to understand it completely to fix it.
- Simply finding it is enough.

Overview of Debugging Issues - An Ineffective Approach

- ***Don’t waste time trying to understand the problem***
- It’s likely that the problem is trivial, and you don’t need to understand it completely to fix it.
- Simply finding it is enough.

Outline

- Debugging Issues
- Finding a Defect
- Fixing a Defect
- Psychological Considerations
- Debugging Tools

Finding a Defect - The Scientific Method of Debugging

- Here are the steps you go through when you use the classic scientific method:
    1. Gather data through repeatable experiments.
    2. Form a hypothesis that accounts for the relevant data.
    3. Design an experiment to prove or disprove the hypothesis.
    4. Prove or disprove the hypothesis.
    5. Repeat as needed.

Finding a Defect - The Scientific Method of Debugging

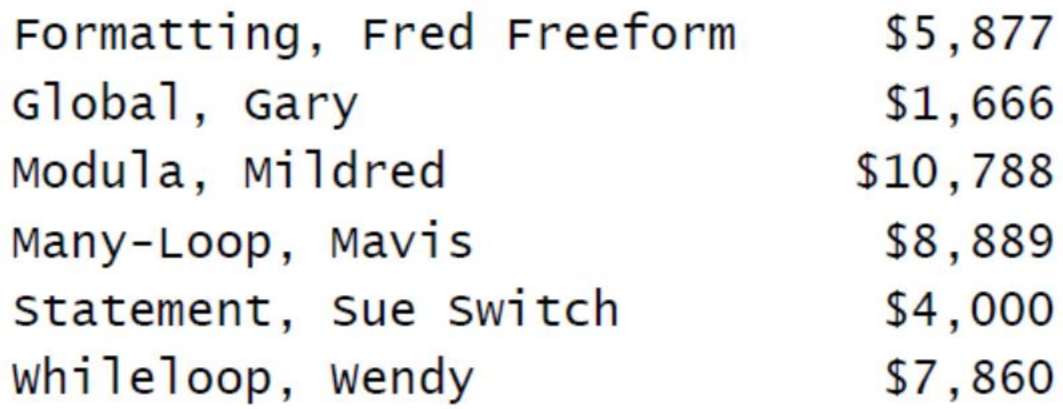
- The scientific method has many parallels in debugging. Here’s an effective approach for finding a defect:
  1. Stabilize the error.
  2. Locate the source of the error (the “fault”).
     - Gather the data that produces the defect.
     - Analyze the data that has been gathered, and form a hypothesis about the defect.
     - Determine how to prove or disprove the hypothesis, either by testing the program or by examining the code.
     - Prove or disprove the hypothesis by using the procedure identified in 2(c).
  3. Fix the defect.
  4. Test the fix.
  5. Look for similar errors.

Finding a Defect - The Scientific Method of Debugging

- The first step is similar to the scientific method's first step in that it relies on repeatability.
  - The defect is easier to diagnose if you can stabilize it—that is, make it occur reliably.
- The second step uses the steps of the scientific method. You gather the test data that divulged the defect, analyze the data that has been produced, and form a hypothesis about the source of the error.
- You then design a test case or an inspection to evaluate the hypothesis, and you either declare success (regarding proving your hypothesis) or renew your efforts, as appropriate.
- When you have proven your hypothesis, you fix the defect, test the fix, and search your code for similar errors.

Finding a Defect - The Scientific Method of Debugging

- Let’s look at each of the steps in conjunction with an example. Assume that you have an employee database program that has an error.
- The program is supposed to print a list of employees and their income-tax withholdings in alphabetical order. Here’s part of the output:
  - The error is that *Many-Loop, Mavis* and *Modula, Mildred* are out of order.



```
Formatting, Fred Freeform     $5,877
Global, Gary                  $1,666
Modula, Mildred              $10,788
Many-Loop, Mavis              $8,889
Statement, Sue Switch         $4,000
Whileloop, Wendy              $7,860
```

Finding a Defect - The Scientific Method of Debugging | Stabilize the Error

- If a defect doesn’t occur reliably, it’s almost impossible to diagnose.
- Making a defect occur predictably is one of the most challenging tasks in debugging.

Finding a Defect - The Scientific Method of Debugging | Stabilize the Error

- An error that doesn’t occur predictably is usually an initialization error, a timing issue, or a dangling-pointer problem.
- If the calculation of a sum is right sometimes and wrong sometimes, a variable involved in the calculation probably isn’t being initialized properly—most of the time it just happens to start at *0*.
- If the problem is a strange and unpredictable phenomenon and you’re using pointers, you almost certainly have an uninitialized pointer or are using a pointer after the memory that it points to has been deallocated.

Finding a Defect - The Scientific Method of Debugging | Stabilize the Error

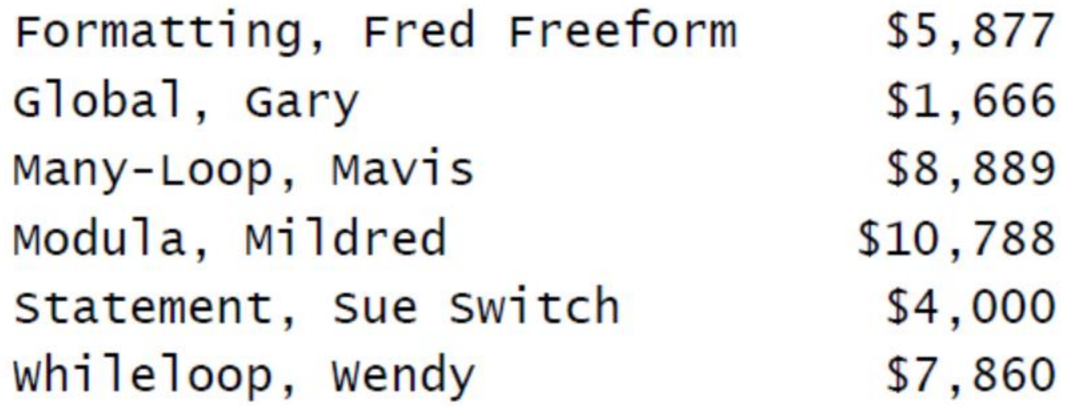
- Stabilizing an error usually requires more than finding a test case that produces the error.
- It includes narrowing the test case to the simplest one that still produces the error.
- The goal of simplifying the test case is to make it so simple that changing any aspect of it changes the behavior of the error. Then, by changing the test case carefully and watching the program's behavior under controlled conditions, you can diagnose the problem.
- If you work in an organization that has an independent test team, sometimes it's the team's job to make the test cases simple. Most of the time, it's your job.

Finding a Defect - The Scientific Method of Debugging | Stabilize the Error

- To simplify the test case, you bring the scientific method into play again. Suppose you have 10 factors that, used in combination, produce the error. Form a hypothesis about which factors were irrelevant to producing the error.
- Change the supposedly irrelevant factors, and rerun the test case. If you still get the error, you can eliminate those factors and you've simplified the test.
- Then you can try to simplify the test further. If you don't get the error, you've disproved that specific hypothesis and you know more than you did before.
- It might be that some subtly different change would still produce the error, but you know at least one specific change that does not.

Finding a Defect - The Scientific Method of Debugging | Stabilize the Error

- In the employee withholdings example, when the program is run initially, *Many-Loop, Mavis* is listed after *Modula, Mildred*.
- When the program is run a second time, however, the list is fine:



```
Formatting, Fred Freeform     $5,877
Global, Gary                  $1,666
Many-Loop, Mavis              $8,889
Modula, Mildred              $10,788
Statement, Sue Switch         $4,000
Whileloop, Wendy              $7,860
```

Finding a Defect - The Scientific Method of Debugging | Stabilize the Error

- It isn't until *Fruit-Loop, Frita* is entered and shows up in an incorrect position that you remember that *Modula, Mildred* had been entered just prior to showing up in the wrong spot too.
- What's odd about both cases is that they were entered singly. Usually, employees are entered in groups.

Finding a Defect - The Scientific Method of Debugging | Stabilize the Error

- You hypothesize: the problem has something to do with entering a single new employee. If this is true, running the program again should put *Fruit-Loop, Frita* in the right position. Here's the result of a second run:

```
Formatting, Fred Freeform      $5,877
Fruit-Loop, Frita              $5,771
Global, Gary                   $1,666
Many-Loop, Mavis               $8,889
Modula, Mildred               $10,788
Statement, Sue Switch          $4,000
Whileloop, Wendy               $7,860
```

- This successful run supports the hypothesis. To confirm it, you want to try adding a few new employees, one at a time, to see whether they show up in the wrong order and whether the order changes on the second run.

Finding a Defect - The Scientific Method of Debugging | Locate the Source of the Error

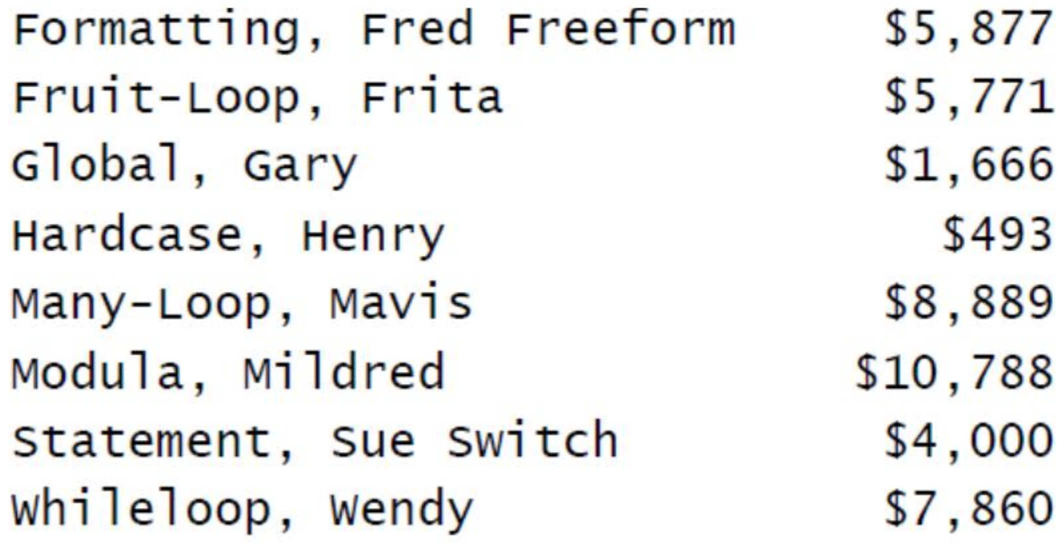
- Locating the source of the error also calls for using the scientific method.
- You might suspect that the defect is a result of a specific problem, say an off-by-one error.
- You could then vary the parameter you suspect is causing the problem—one below the boundary, on the boundary, and one above the boundary—and determine whether your hypothesis is correct.

Finding a Defect - The Scientific Method of Debugging | Locate the Source of the Error

- In the running example, the source of the problem could be an off-by-one defect that occurs when you add one new employee but not when you add two or more.
- Examining the code, you don’t find an obvious off-by-one defect.

Finding a Defect - The Scientific Method of Debugging | Locate the Source of the Error

- Resorting to Plan B, you run a test case with a single new employee to see whether that’s the problem. You add *Hardcase, Henry* as a single employee and hypothesize that his record will be out of order.
- Here’s what you find:



```
Formatting, Fred Freeform     $5,877
Fruit-Loop, Frita             $5,771
Global, Gary                  $1,666
Hardcase, Henry                 $493
Many-Loop, Mavis              $8,889
Modula, Mildred              $10,788
Statement, Sue Switch         $4,000
Whileloop, Wendy              $7,860
```

Finding a Defect - The Scientific Method of Debugging | Locate the Source of the Error

- The line for *Hardcase, Henry* is exactly where it should be, which means that your first hypothesis is false.
- The problem isn't caused simply by adding one employee at a time.
- It's either a more complicated problem or something completely different.

Finding a Defect - The Scientific Method of Debugging | Locate the Source of the Error

- Examining the test-run output again, you notice that *Fruit-Loop, Frita* and *Many-Loop, Mavis* are the only names containing hyphens (-).
- *Fruit-Loop* was out of order when she was first entered, but *Many-Loop* wasn't, was she?
- Although you don't have a printout from the original entry, in the original error *Modula, Mildred* appeared to be out of order, but she was next to *Many-Loop*.
- Maybe *Many-Loop* was out of order and *Modula* was all right.
- You hypothesize again: the problem arises from names with hyphens, not names that are entered singly.

Finding a Defect - The Scientific Method of Debugging | Locate the Source of the Error

- But how does that account for the fact that the problem shows up only the first time an employee is entered? You look at the code and find that two different sorting routines are used.
- One is used when an employee is entered, and another is used when the data is saved. A closer look at the routine used when an employee is first entered shows that it isn't supposed to sort the data completely. It only puts the data in approximate order to speed up the save routine's sorting.
- Thus, the problem is that the data is printed before it's sorted. The problem with hyphenated names arises because the rough-sort routine doesn't handle niceties such as punctuation characters.
- Now, you can refine the hypothesis even further.

Finding a Defect - Tips for Finding Defects

- ***Check for common defects***
- Use code-quality checklists to stimulate your thinking about possible defects.
- If you’re following the inspection practices described in Section 21.3, “Formal Inspections,” you’ll have your own fine-tuned checklist of the common problems in your environment.
- You can also use the checklists that appear throughout this book. See the “List of Checklists” following the book’s table of contents.

Finding a Defect - The Scientific Method of Debugging | Locate the Source of the Error

- You hypothesize one last time: names with punctuation characters aren’t sorted correctly until they’re saved.
- You later confirm this hypothesis with additional test cases.

Finding a Defect - Tips for Finding Defects

- Once you’ve stabilized an error and refined the test case that produces it, finding its source can be either trivial or challenging, depending on how well you’ve written your code.
- If you’re having a hard time finding a defect, it could be because the code isn’t well written.

Finding a Defect - Tips for Finding Defects

- ***Use all the data available to make your hypothesis***
- When creating a hypothesis about the source of a defect, account for as much of the data as you can in your hypothesis.
- In the example, you might have noticed that *Fruit-Loop, Frita* was out of order and created a hypothesis that names beginning with an "F" are sorted incorrectly.
- That's a poor hypothesis because it doesn't account for the fact that *Modula, Mildred* was out of order or that names are sorted correctly the second time around. If the data doesn't fit the hypothesis, don't discard the data—ask why it doesn't fit, and create a new hypothesis.

Finding a Defect - Tips for Finding Defects

- ***Use all the data available to make your hypothesis***
- The second hypothesis in the example—that the problem arises from names with hyphens, not names that are entered singly—didn't seem initially to account for the fact that names were sorted correctly the second time around either.
- In this case, however, the second hypothesis led to a more refined hypothesis that proved to be correct.
- It's all right that the hypothesis doesn't account for all of the data at first as long as you keep refining the hypothesis so that it does eventually.

Finding a Defect - Tips for Finding Defects

- ***Refine the test cases that produce the error***
- If you can’t find the source of an error, try to refine the test cases further than you already have.
- You might be able to vary one parameter more than you had assumed, and focusing on one of the parameters might provide the crucial breakthrough.

Finding a Defect - Tips for Finding Defects

- ***Exercise the code in your unit test suite***
- Defects tend to be easier to find in small fragments of code than in large integrated programs.
- Use your unit tests to test the code in isolation.

Finding a Defect - Tips for Finding Defects

- ***Use available tools***
- Numerous tools are available to support debugging sessions: interactive debuggers, picky compilers, memory checkers, syntax-directed editors, and so on.
- The right tool can make a difficult job easy. With one tough-to-find error, for example, one part of the program was overwriting another part's memory.
- This error was difficult to diagnose using conventional debugging practices because the programmer couldn't determine the specific point at which the program was incorrectly overwriting memory.
- The programmer used a memory breakpoint to set a watch on a specific memory address. When the program wrote to that memory location, the debugger stopped the code and the guilty code was exposed.

Finding a Defect - Tips for Finding Defects

- ***Reproduce the error several different ways***
- Sometimes trying cases that are similar to the error-producing case but not exactly the same is instructive.
- Think of this approach as triangulating the defect.
- If you can get a fix on it from one point and a fix on it from another, you can better determine exactly where it is.

Finding a Defect - Tips for Finding Defects

- ***Reproduce the error several different ways***
- Reproducing an error several different ways helps diagnose the cause of the error.
- Once you think you've identified the defect, run a case that's close to the cases that produce errors but that should not produce an error itself. If it does produce an error, you don't completely understand the problem yet.
- Errors often arise from combinations of factors, and trying to diagnose the problem with only one test case often doesn't diagnose the root problem.

Finding a Defect - Tips for Finding Defects

- ***Reproduce the error several different ways***

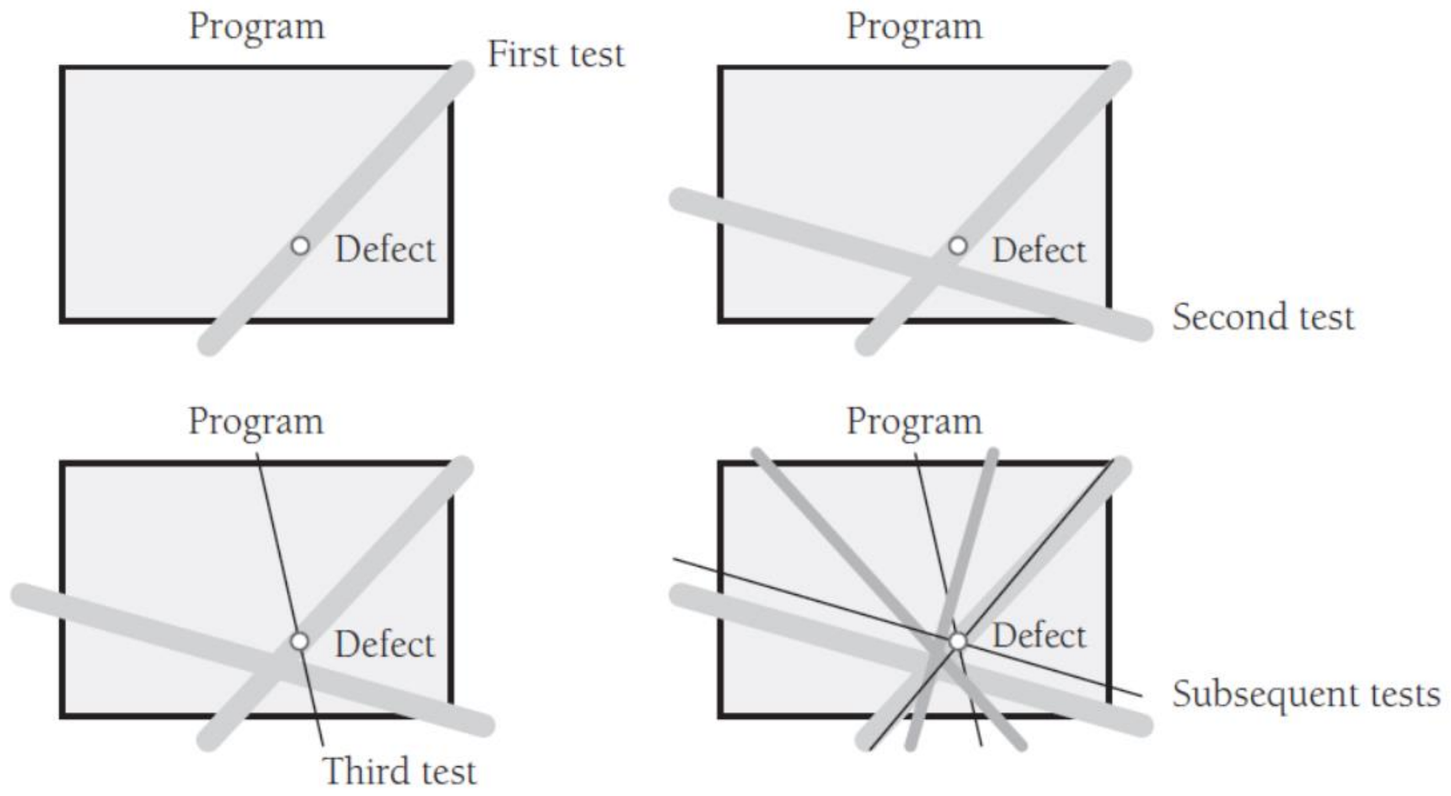


Figure 23-1 Try to reproduce an error several different ways to determine its exact cause

Finding a Defect - Tips for Finding Defects

- ***Generate more data to generate more hypotheses***
- Choose test cases that are different from the test cases you already know to be erroneous or correct.
- Run them to generate more data, and use the new data to add to your list of possible hypotheses.

Finding a Defect - Tips for Finding Defects

- ***Use the results of negative tests***
- Suppose you create a hypothesis and run a test case to prove it. Suppose further that the test case disproves the hypothesis, so you still don’t know the source of the error.
- You do know something you didn’t before—namely, that the defect is not in the area you thought it was.
- That narrows your search field and the set of remaining possible hypotheses.

Finding a Defect - Tips for Finding Defects

- ***Brainstorm for possible hypotheses***
- Rather than limiting yourself to the first hypothesis you think of, try to come up with several.
- Don’t analyze them at first—just come up with as many as you can in a few minutes.
- Then look at each hypothesis and think about test cases that would prove or disprove it.
- This mental exercise is helpful in breaking the debugging logjam that results from concentrating too hard on a single line of reasoning.

Finding a Defect - Tips for Finding Defects

- ***Keep a notepad by your desk, and make a list of things to try***
- One reason programmers get stuck during debugging sessions is that they go too far down dead-end paths.
- Make a list of things to try, and if one approach isn’t working, move on to the next approach.

Finding a Defect - Tips for Finding Defects

- ***Narrow the suspicious region of the code***
- If you’ve been testing the whole program or a whole class or routine, test a smaller part instead.
- Use print statements, logging, or tracing to identify which section of code is producing the error.

Finding a Defect - Tips for Finding Defects

- ***Narrow the suspicious region of the code***
- If you need a more powerful technique to narrow the suspicious region of the code, systematically remove parts of the program and see whether the error still occurs.
- If it doesn’t, you know it’s in the part you took away. If it does, you know it’s in the part you’ve kept.

Finding a Defect - Tips for Finding Defects

- ***Narrow the suspicious region of the code***
- Rather than removing regions haphazardly, divide and conquer. Use a binary search algorithm to focus your search.
- Try to remove about half the code the first time.
- Determine the half the defect is in, and then divide that section. Again, determine which half contains the defect, and again, chop that section in half. Continue until you find the defect.

Finding a Defect - Tips for Finding Defects

- ***Narrow the suspicious region of the code***
- If you use many small routines, you'll be able to chop out sections of code simply by commenting out calls to the routines.
- Otherwise, you can use comments or pre-processor commands to remove code.

Finding a Defect - Tips for Finding Defects

- ***Narrow the suspicious region of the code***
- If you’re using a debugger, you don’t necessarily have to remove pieces of code. You can set a breakpoint partway through the program and check for the defect that way instead.
- If your debugger allows you to skip calls to routines, eliminate suspects by skipping the execution of certain routines and seeing whether the error still occurs.
- The process with a debugger is otherwise similar to the one in which pieces of a program are physically removed.

Finding a Defect - Tips for Finding Defects

- ***Be suspicious of classes and routines that have had defects before***
- Classes that have had defects before are likely to continue to have defects.
- A class that has been troublesome in the past is more likely to contain a new defect than a class that has been defect-free.
- Reexamine error-prone classes and routines.

Finding a Defect - Tips for Finding Defects

- ***Check code that's changed recently***
- If you have a new error that's hard to diagnose, it's usually related to code that's changed recently.
- It could be in completely new code or in changes to old code. If you can't find a defect, run an old version of the program to see whether the error occurs. If it doesn't, you know the error's in the new version or is caused by an interaction with the new version. Scrutinize the differences between the old and new versions.
- Check the version control log to see what code has changed recently. If that's not possible, use a diff tool to compare changes in the old, working source code to the new, broken source code.

Finding a Defect - Tips for Finding Defects

- ***Expand the suspicious region of the code***
- It’s easy to focus on a small section of code, sure that “the defect *must* be in this section.”
- If you don’t find it in the section, consider the possibility that the defect isn’t in the section.
- Expand the area of code you suspect, and then focus on pieces of it by using the binary search technique described earlier.

Finding a Defect - Tips for Finding Defects

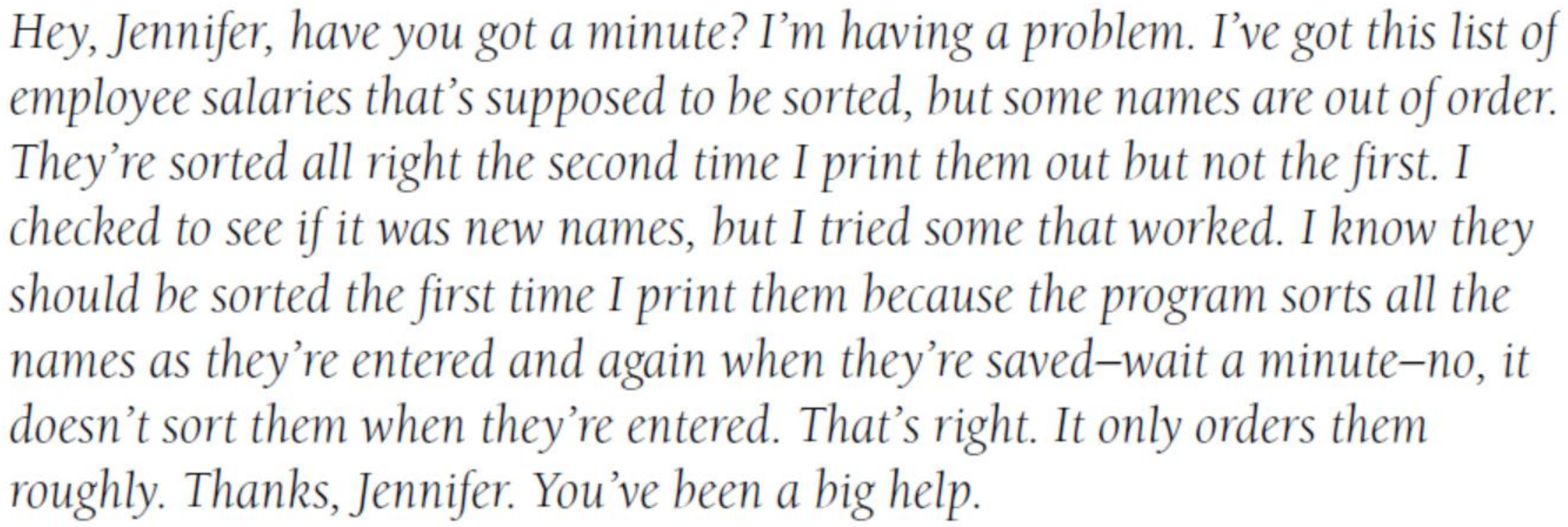
- ***Integrate incrementally***
- Debugging is easy if you add pieces to a system one at a time.
- If you add a piece to a system and encounter a new error, remove the piece and test it separately.

Finding a Defect - Tips for Finding Defects

- ***Talk to someone else about the problem***
- Some people call this “confessional debugging.”
- You often discover your own defect in the act of explaining it to another person.
- For example, if you were explaining the problem in the salary example, you might sound like this:

Finding a Defect - Tips for Finding Defects

- ***Talk to someone else about the problem***



Hey, Jennifer, have you got a minute? I'm having a problem. I've got this list of employee salaries that's supposed to be sorted, but some names are out of order. They're sorted all right the second time I print them out but not the first. I checked to see if it was new names, but I tried some that worked. I know they should be sorted the first time I print them because the program sorts all the names as they're entered and again when they're saved–wait a minute–no, it doesn't sort them when they're entered. That's right. It only orders them roughly. Thanks, Jennifer. You've been a big help.

- Jennifer didn't say a word, and you solved your problem. This result is typical, and this approach is a potent tool for solving difficult defects.

Finding a Defect - Tips for Finding Defects

- ***Take a break from the problem***
- Sometimes you concentrate so hard you can’t think.
- How many times have you paused for a cup of coffee and figured out the problem on your way to the coffee machine? Or in the middle of lunch? Or on the way home? Or in the shower the next morning?
- If you’re debugging and making no progress, once you’ve tried all the options, let it rest.
- Go for a walk. Work on something else. Go home for the day. Let your subconscious mind tease a solution out of the problem.

Outline

- Debugging Issues
- Finding a Defect
- Fixing a Defect
- Psychological Considerations in Debugging
- Debugging Tools—Obvious and Not-So-Obvious

Fixing a Defect

- The hard part of debugging is finding the defect. Fixing the defect is the easy part.
- But as with many easy tasks, the fact that it's easy makes it especially error-prone.
- At least one study found that defect corrections have more than a 50 percent chance of being wrong the first time (Yourdon 1986b).

Fixing a Defect

- ***Understand the problem before you fix it***
- The best way to make your life difficult and corrode the quality of your program is to fix problems without really understanding them.
- Before you fix a problem, make sure you understand it to the core.

Fixing a Defect

- ***Understand the program, not just the problem***
- If you understand the context in which a problem occurs, you're more likely to solve the problem completely rather than only one aspect of it.
- A study done with short programs found that programmers who achieve a global understanding of program behavior have a better chance of modifying it successfully than programmers who focus on local behavior, learning about the program only as they need to (Littman et al. 1986).
- Because the program in this study was small (280 lines), it doesn't prove that you should try to understand a 50,000-line program completely before you fix a defect. It does suggest that you should understand at least the code in the vicinity of the defect correction—the "vicinity" being not a few lines but a few hundred.

Fixing a Defect

- ***Confirm the defect diagnosis***
- Before you rush to fix a defect, make sure that you’ve diagnosed the problem correctly.
- Take the time to run test cases that prove your hypothesis and disprove competing hypotheses.
- If you’ve proven only that the error could be the result of one of several causes, you don’t yet have enough evidence to work on the one cause; rule out the others first.

Fixing a Defect

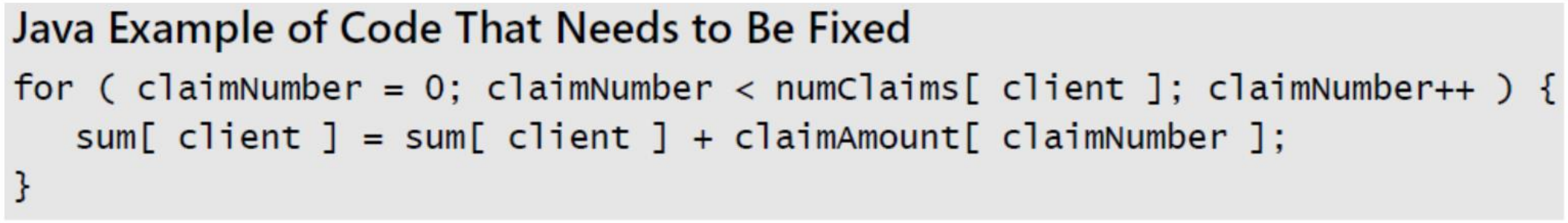
- ***Save the original source code***
- Before you begin fixing the defect, be sure to archive a version of the code that you can return to later.
- It’s easy to forget which change in a group of changes is the significant one.
- If you have the original source code, at least you can compare the old and the new files and see where the changes are.

Fixing a Defect

- ***Fix the problem, not the symptom***
- You should fix the symptom too, but the focus should be on fixing the underlying problem rather than wrapping it in programming duct tape.
- If you don’t thoroughly understand the problem, you’re not fixing the code.
- You’re fixing the symptom and making the code worse.

Fixing a Defect

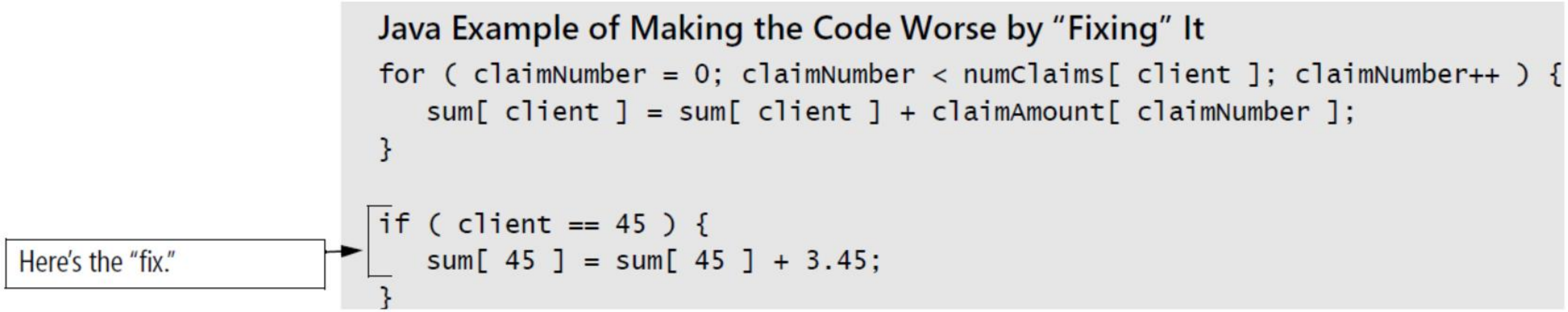
- ***Fix the problem, not the symptom***



Java Example of Code That Needs to Be Fixed

```
for ( claimNumber = 0; claimNumber < numClaims[ client ]; claimNumber++ ) {
   sum[ client ] = sum[ client ] + claimAmount[ claimNumber ];
}
```

- Suppose that when *client* equals *45*, *sum* turns out to be wrong by $3.45. Here’s the wrong way to fix the problem:



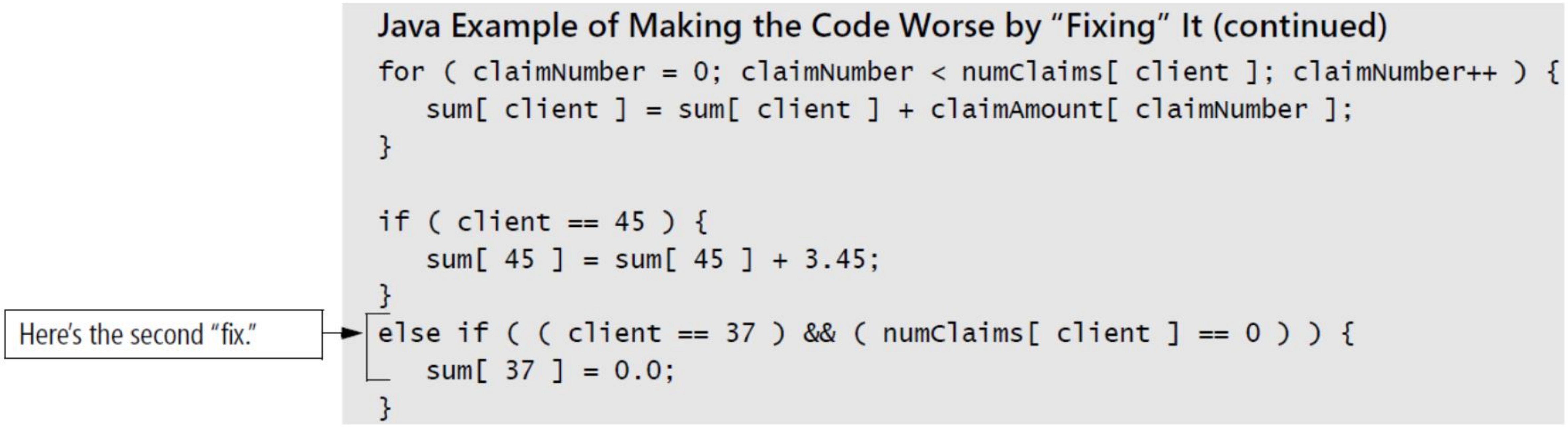
Java Example of Making the Code Worse by “Fixing” It

```
for ( claimNumber = 0; claimNumber < numClaims[ client ]; claimNumber++ ) {
   sum[ client ] = sum[ client ] + claimAmount[ claimNumber ];
}

if ( client == 45 ) {
   sum[ 45 ] = sum[ 45 ] + 3.45;
}
```

Here’s the “fix.”

Fixing a Defect

- ***Fix the problem, not the symptom***
- Now suppose that when *client* equals *37* and the number of claims for the client is *0*, you're not getting *0*. Here's the wrong way to fix the problem:



Java Example of Making the Code Worse by "Fixing" It (continued)

```java
for ( claimNumber = 0; claimNumber < numClaims[ client ]; claimNumber++ ) {
   sum[ client ] = sum[ client ] + claimAmount[ claimNumber ];
}

if ( client == 45 ) {
   sum[ 45 ] = sum[ 45 ] + 3.45;
}
else if ( ( client == 37 ) && ( numClaims[ client ] == 0 ) ) {
   sum[ 37 ] = 0.0;
}
```

Here's the second "fix."

Fixing a Defect

- ***Change the code only for good reason***
- Related to fixing symptoms is the technique of changing code at random until it seems to work.
- The typical line of reasoning goes like this: "This loop seems to contain a defect. It's probably an off-by-one error, so I'll just put a *-1* here and try it. OK. That didn't work, so I'll just put a *+1* in instead. OK. That seems to work. I'll say it's fixed."
- By changing the program randomly, you say in effect, "I don't know what's happening here, but I'll try this change and hope it works."
- Don't change code randomly.

Fixing a Defect

- ***Make one change at a time***
- Changes are tricky enough when they're done one at a time.
- When done two at a time, they can introduce subtle errors that look like the original errors.
- Then you're in the awkward position of not knowing whether you didn't correct the error, whether you corrected the error but introduced a new one that looks similar, or whether you didn't correct the error and you introduced a similar new error.
- Keep it simple: make just one change at a time.

Fixing a Defect

- ***Check your fix***
- Check the program yourself, have someone else check it for you, or walk through it with someone else.
- Run the same triangulation test cases you used to diagnose the problem to make sure that all aspects of the problem have been resolved.
- If you’ve solved only part of the problem, you’ll find out that you still have work to do.
- Rerun the whole program to check for side effects of your changes. The easiest and most effective way to check for side effects is to run the program through an automated suite of regression tests.

Fixing a Defect

- ***Add a unit test that exposes the defect***
- When you encounter an error that wasn’t exposed by your test suite, add a test case to expose the error so that it won’t be reintroduced later.

Fixing a Defect

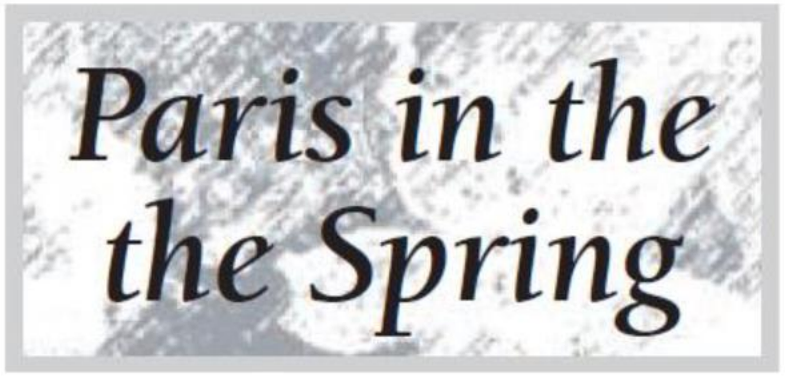
- ***Look for similar defects***
- When you find one defect, look for others that are similar.
- Defects tend to occur in groups, and one of the values of paying attention to the kinds of defects you make is that you can correct all the defects of that kind.
- Looking for similar defects requires you to have a thorough understanding of the problem.
- Watch for the warning sign: if you can't figure out how to look for similar defects, that's a sign that you don't yet completely understand the problem.

Outline

- Debugging Issues
- Finding a Defect
- Fixing a Defect
- Psychological Considerations
- Debugging Tools

Psychological Considerations

- When you see a token in a program that says *Num*, what do you see?
- Do you see a misspelling of the word “Numb”? Or do you see the abbreviation for “Number”? Most likely, you see the abbreviation for “Number.”
- This is the phenomenon of “psychological set”—seeing what you expect to see.



Psychological Considerations

- What was written on the previous image?
- In this classic puzzle, people often see only one “the.” People see what they expect to see.

Psychological Considerations

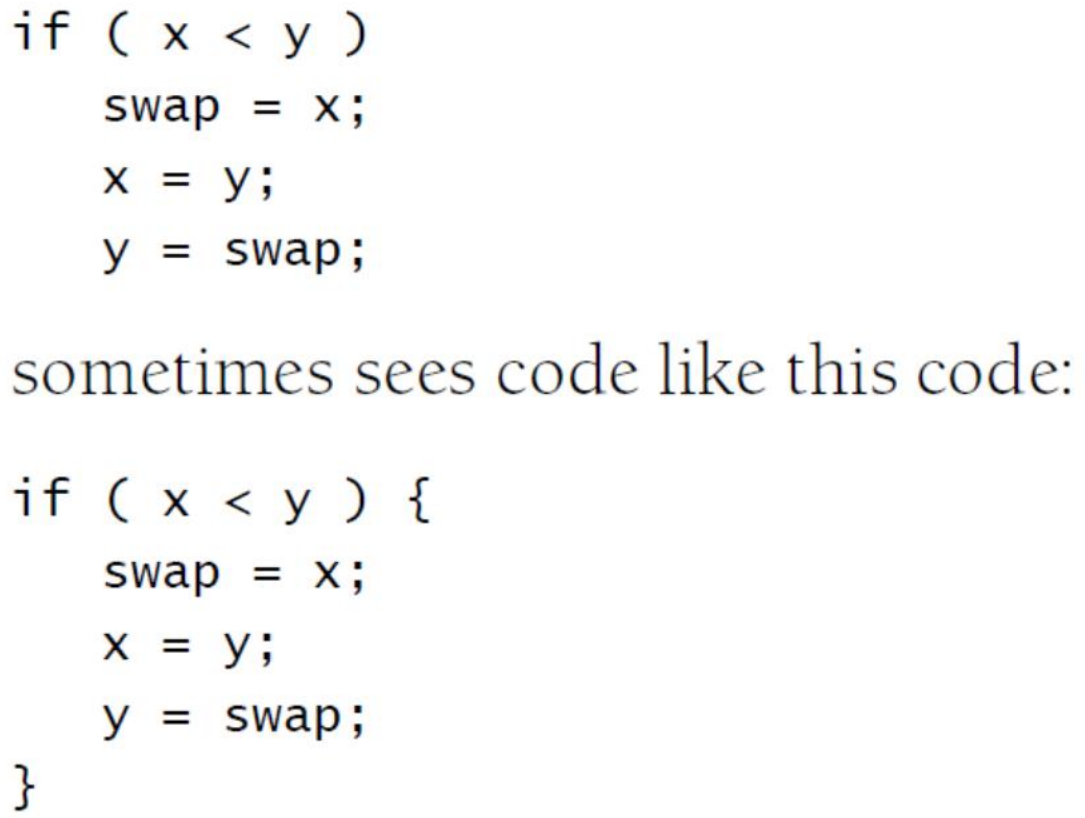
- Students learning *while* loops often expect a loop to be continuously evaluated; that is, they expect the loop to terminate as soon as the *while* condition becomes false, rather than only at the top or bottom (Curtis et al. 1986).
- They expect a *while* loop to act as “while” does in natural language.

Psychological Considerations

- A programmer who unintentionally used both the variable SYSTSTS and the variable SYSSTSTS thought he was using a single variable.
- He didn’t discover the problem until the program had been run hundreds of times and a book was written containing the erroneous results (Weinberg 1998).

Psychological Considerations

- A programmer looking at code like this code:



```
if ( x < y )
   swap = x;
   x = y;
   y = swap;
```

sometimes sees code like this code:

```
if ( x < y ) {
   swap = x;
   x = y;
   y = swap;
}
```

Psychological Considerations

- Psychological distance can be defined as the ease with which two items can be differentiated.
- If you are looking at a long list of words and have been told that they’re all about ducks, you could easily mistake “Queck” for “Quack” because the two words look similar.
- The psychological distance between the words is small.
- You would be much less likely to mistake “Tuack” for “Quack” even though the difference is only one letter again. “Tuack” is less like “Quack” than “Queck” is because the first letter in a word is more prominent than the one in the middle.

Psychological Considerations

- As you construct code, choose names with large differences so that you avoid the problem.

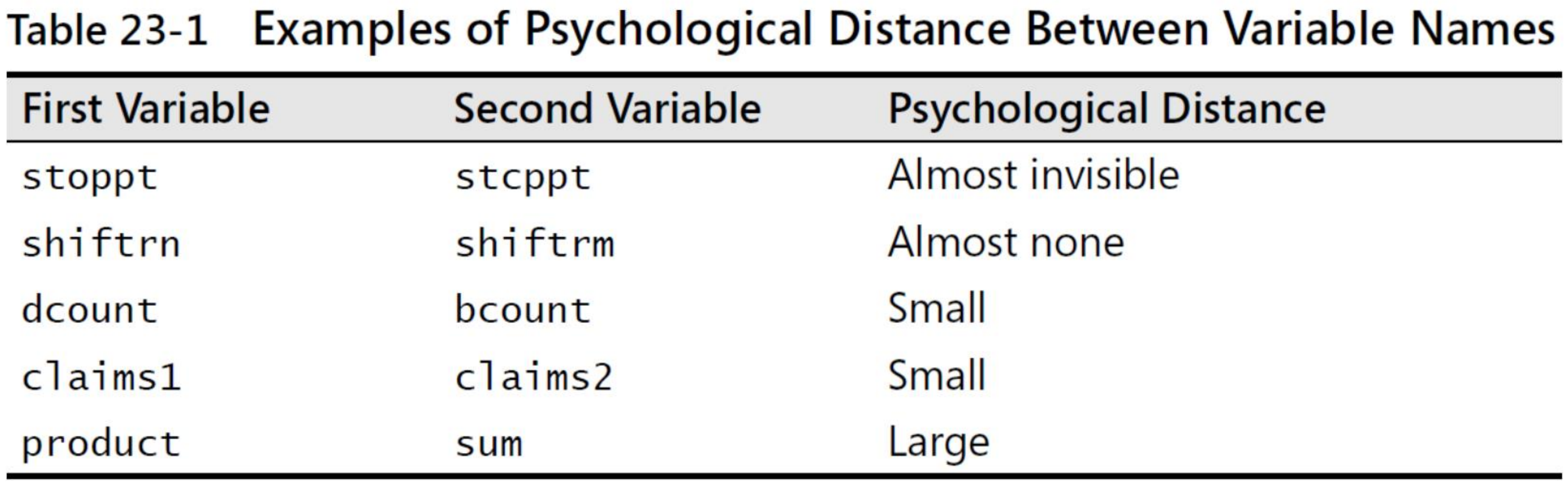


Table 23-1 Examples of Psychological Distance Between Variable Names

First Variable	Second Variable	Psychological Distance
`stoppt`	`stcppt`	Almost invisible
`shiftrn`	`shiftrm`	Almost none
`dcount`	`bcount`	Small
`claims1`	`claims2`	Small
`product`	`sum`	Large

Outline

- Debugging Issues
- Finding a Defect
- Fixing a Defect
- Psychological Considerations
- Debugging Tools

Debugging Tools

- You can do much of the detailed, brain-busting work of debugging with debugging tools that are readily available.
- The tool that will drive the final stake through the heart of the defect vampire isn’t yet available, but each year brings an incremental improvement in available capabilities.

Debugging Tools - Source-Code Comparators

- A source-code comparator such as Diff is useful when you’re modifying a program in response to errors.
- If you make several changes and need to remove some that you can’t quite remember, a comparator can pinpoint the differences and jog your memory.
- If you discover a defect in a new version that you don’t remember in an older version, you can compare the files to determine what changed.

Debugging Tools - Compiler Warning Messages

- ***Set your compiler's warning level to the highest, pickiest level possible, and fix the errors it reports***
- It's sloppy to ignore compiler errors. It's even sloppier to turn off the warnings so that you can't even see them.
- Setting a switch on the compiler to turn off warnings just means you can't see the errors.
- It doesn't make them go away.

Debugging Tools - Compiler Warning Messages

- ***Treat warnings as errors***
- Some compilers let you treat warnings as errors.
- One reason to use the feature is that it elevates the apparent importance of a warning.
- Just as setting your watch five minutes fast tricks you into thinking it's five minutes later than it is, setting your compiler to treat warnings as errors tricks you into taking them more seriously.

Debugging Tools - Extended Syntax and Logic Checking

- You can use additional tools to check your code more thoroughly than your compiler does.
- For example, for C programmers, the lint utility painstakingly checks for use of uninitialized variables (writing = when you mean = =) and similarly subtle problems.

Debugging Tools - Debuggers

- Good debuggers allow you to set breakpoints to break when execution reaches a specific line, or the *n*th time it reaches a specific line, or when a global variable changes, or when a variable is assigned a specific value.
- They allow the program to be executed backwards, stepping back to the point where a defect originated.
- Good debuggers allow full examination of data, including structured and dynamically allocated data. They make it easy to view the contents of a linked list of pointers or a dynamically allocated array.

Debugging Tools - Execution Profilers

- You might not think of an execution profiler as a debugging tool, but a few minutes spent studying a program profile can uncover some surprising (and hidden) defects.
- For example, I had suspected that a memory-management routine in one of my programs was a performance bottleneck.
  - Memory management had originally been a small component using a linearly ordered array of pointers to memory. .
  - I replaced the linearly ordered array with a hash table in the expectation that execution time would drop by at least half. But after profiling the code, I found no change in performance at all. I examined the code more closely and found a defect that was wasting a huge amount of time in the allocation algorithm.

つづく