An Introduction to Computer Science with Java

Last edited January 6, 2014 by C. Herbert

This document is a draft of a chapter from *Computer Science with Java*, written by Charles Herbert with the assistance of Craig Nelson. It is available free of charge for students in Computer Science at Community College of Philadelphia during the Spring 2014 semester. It may not be reproduced or distributed for any other purposes without proper prior permission.

Contents

Introduction to Computer Science with Java

Chapter 13 – Software Testing

This chapter introduces the concepts of software testing, including software validation and process verification; the development of testing plan; creating software test cases; and creating stubs and drivers for software testing.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the concepts of *software validation* and *process verification.*
- describe the importance of independent software validation and verification and three different parameters for independence.
- Describe what unit tests and integration tests are and how they are related to one another.
- describe what a software testing plan is, and the components of a simplified software testing plan including what should be included in the introduction, how to determine what methods need to be tested and what methods do not need to be tested, what should be included in an annotated list of test to be conducted, and typical software test deliverables.
- describe how IEEE standards 1012 and 829 are related to software testing.
- describe the nature of a test for correctness and the criteria for such tests, and what the criteria are based and when a software testing plan based on the criteria is usually developed.
- Describe the concept of internal software validation and when it occurs.
- Create a simplified software testing plan for a given software development project.
- Describe what a test case is and the nature of test case criteria.
- Create test cases for a given software development project.
- describe the difference between quantitative test criteria and qualitative test criteria and how to develop clear testing criteria from qualitative test criteria.
- Describe the nature and importance of requirements traceability.
- Describe the nature and use of stubs and drivers for software testing.
- Create stubs and drivers to test methods for a given software development project.

6.1 Independent Validation and Verification

Software development can be viewed as an engineering process in which we design and build new products using math, science, and technology. Engineers actually design both *products* and the *processes* that produce those products. To ensure the quality of their work, they *validate* products to make sure that products meet their specified design requirements and *verify* processes to make sure that the processes are producing the correct results and following industry standards. These standards include such things as health and safety guidelines, financial regulations, reporting requirements, and so on.

There is some overlap between validation and verification, but generally, **validation** is the testing of a product to make sure it meets specified design requirements, while **verification** is the examination of a process to make sure that it produces the correct results according to industry standards. Dr. Steve Easterbknight, a former Lead Computer Scientist for NASA's *Independent Validation and Verification Program*, describes it this way:

> *Validation: Are we building the right system?*
> *Verification: Are we building the system right?*
>
> See an entry in Dr. Easterbknight's blog *Serendipity* about this on the Web at:
> http://www.easterbknight.ca/steve/2010/11/the-difference-between-verification-and-validation

Validation of software is reasonably straightforward – does the software meet its specified requirements? Validation is based on specified software requirements and is as simple or as complicated as the specified requirements.

Of course, the validation process should make sure the software meets the needs of the customer, which should be a primary concern at the time software specifications are developed. This falls into the realm of systems analysis and design, which is not part of this course. This course focuses on developing software that meets requirements rather than on developing requirements, although in practice it is sometimes difficult to separate the two.

Verification is a bit more complicated, because it requires an examination of software development processes. Most often this is done by examining software development documents and source code to look for evidence that software developers are using industry standard practices.

Validation and Verification are best done by software development experts other than those who design and build the products themselves – experts who are independent of the development and production team. The term **independent validation and verification (IV&V)** is often used in engineering to highlight this. NASA, for example, has established the *Independent Validation and Verification Program* whose mission is to examine NASA's software engineering and:

> *"…provide our customers assurance that their safety and mission-critical software will operate reliably and safely and to advance the systems and software engineering disciplines. In doing so, we work to standards of excellence, provide professional engineers, provide national and global leadership, focus on customer satisfaction, and adhere to and demonstrate our core set of values: safety, integrity, respect, teamwork, balance, innovation, and excellence."*

NASA IV&V Program Mission Statement
on the Web at: http://www.nasa.gov/centers/ivv/about/visionmission.html

To help ensure its independence, NASA's *Independent Validation and Verification Program* is even located apart from any other NASA facilities in Fairmont, WV. They report directly to NASA's Office of Safety and Mission Assurance (OSMA) at NASA Headquarters in Washington, DC.

The **IEEE 1012** *Standard for Software Verification and Validation*[1], describes in detail processes for validating and verifying software in all stages of software systems development. It describes three parameters for independence:

- **technical independence**

  "… requires the V&V effort to utilize personnel who are not involved in the development of the software. The IV&V effort should formulate its own understanding of the problem and how the proposed system is solving the problem. Technical independence ("fresh viewpoint") is an important method to detect subtle errors overlooked by those too close to the solution."

- **managerial independence**

  "… requires that the responsibility for the IV&V effort be vested in an organization separate from the development and program management organizations. Managerial independence also means that the IV&V effort independently selects the segments of the software and system to

[1] See: *IEEE 1012-2012 - IEEE Standard for System and Software Verification and Validation* on the Web at: https://standards.ieee.org/findstds/standard/1012-2012.html

analyze and test, chooses the IV&V techniques, defines the schedule of IV&V activities, and selects the specific technical issues and problems to act upon. "

- **financial independence**

  "... requires that control of the IV&V budget be vested in an organization independent of the development organization. This independence prevents situations where the IV&V effort cannot complete its analysis or test or deliver timely results because funds have been diverted or adverse financial pressures or influences have been exerted."

The details of software validation and verification are quite extensive and form a subject itself beyond the scope of this course, but we do need to examine how software specifications, software design, and testing are related to one another in the overall process of software development to ensure the correctness and quality of the software we produce.

In the rest of this chapter we will briefly discuss the software development process, then look at software testing as part of internal testing and independent validation. The details of formal software validation and verifying software development processes are the scope of this course. It is enough for now to know generally what they are about and that they are important areas meriting further study for students who wish to become software development professionals.

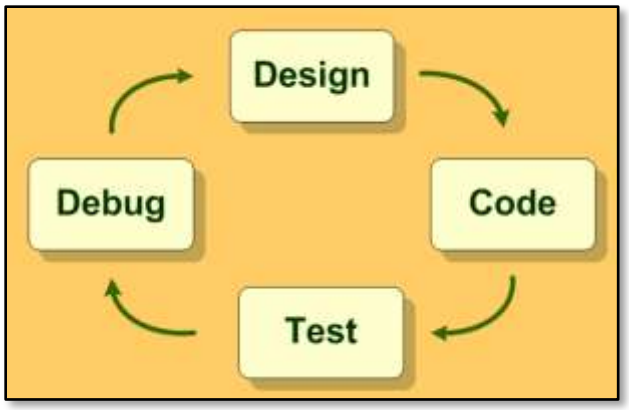
6.2 The Software Development Process

In the language of software engineering, the process of software development can be described in several steps[2]:

- develop requirements for the software
- analyze the requirements for the software
- design software that meets the requirements
- develop software as code that implements the design
- test the software
- debug the software
- install and supporting the software

We can simplify the central part of this process as a four phase **Software Development Cycle**, in which we design, code, test, and debug software. This is the Software Development Cycle discussed back in chapter 2.

[2] These step are based on the descriptions in *IEEE 1012*

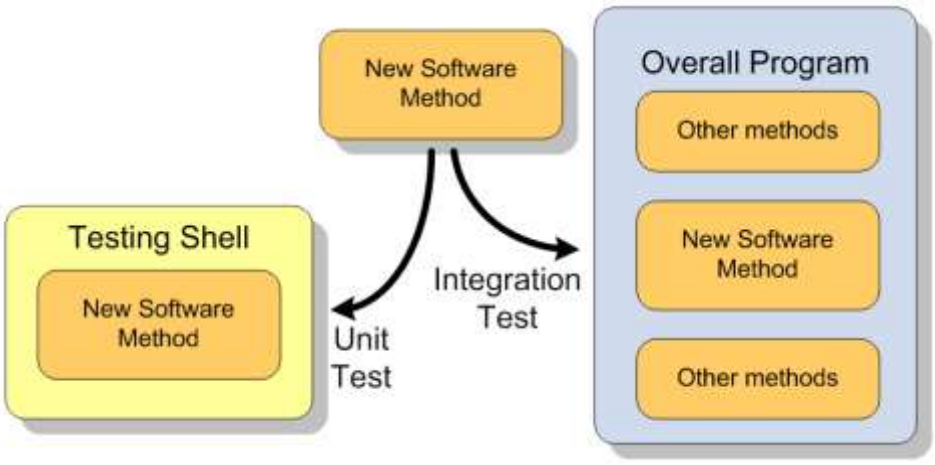


Thorough software testing and debugging can lead us back to revising any flaws in our design and fixing any errors in our code. This is what makes the process a cycle.

The design phase of the simplified software development cycle includes analyzing requirements and designing software that meets the requirements. It also includes designing a testing plan. A software testing plan describes the tests to be conducted to determine if software meets it requirements. The details of a testing plan depend on how the software is organized, but generally it is composed of unit tests and integration tests checking specific software features to make sure they work according to the software requirements.

In chapter 5 we saw two techniques for organizing the design of complex software that are commonly used together in the software development process – top-down design and modular programming, in which a complex program is broken down into simpler components and the simpler components are developed as individual modules that are then integrated to form the desired product. Software module are most often coded and tested as individual software methods in many object-oriented programming languages, including Java.

Each of the methods needs to be designed, developed and validated individually. The complete software package needs to be validated as each method is integrated into the package. A **unit test** checks to see if a method works according to specified requirements all by itself. An **integration test** checks to see if a method works in combination with other methods.



All of the methods in a new class should be individually validated when the class is developed. In a professional environment, existing classes and methods from other sources do not need to be re-validated, provided that their specifications match those needed for the current project and that they were validated by a trusted source. Modifications to existing classes and methods require that they be re-validated after the modification.

Putting all of this together, software is often developed as a series of individual methods, each with their own specifications. The methods need to be individually validated to see if they meet specifications (unit tests), and the overall software product needs to be validated as the methods are integrated into the software (integrations test). Developers should design a testing plan describing how the software will be validated throughout this process.

6.3 Software Testing Plans

A **software testing plan** describes the tests to be performed to validate software. It includes a list of the items being tested, the features of those items that are being tested, the pass/fail criteria for each test, and the software and hardware environment for each test.

The **IEEE 829** *Standard for Software and System Test Documentation*[3] lists test reports that should be produced to validate software. The standard is very detailed – too detailed for the purposes of this course – but it does give us some guidance about what should be included in a testing plan.

The description below outlines the parts of a simplified software testing plan based the more detailed descriptions of the IEEE 829 standard. It can serve as a template for simple software testing plans.

Simplified Software Testing Plan

I. **Introduction** – a brief overview of the software development project and the tests to be conducted. This is often referred to as an "*executive summary*". It briefly describes the purpose and nature of the project, the purpose and nature of the tests to be conducted, and any issues that deserve special mention. The introduction or the details of the tests should tell us if any professional standards are being used.

II. **Items to Be Tested** – a list of the items to be tested. In object-oriented programming this is often a list of classes and methods to be tested. For each item, there should be a description of the specific features to be tested, such as testing a method to see what input it accepts, or testing a method to see if it catches exceptions in accordance with design specifications.

III. **Items Not Being Tested** – a list of the items that do not need to be tested and the reason why each does not need to be tested. This is often a list of classes and methods being reused in this project that have previously been properly validated.

IV. Tests to Be Conducted – an annotated list of the tests to be conducted, including for each test:

A. the purpose of the test – What is being tested and why?

[3] See https://standards.ieee.org/findstds/standard/829-2008.html See

B. a description of how the test will be conducted

C. the pass/fail criteria for the test

V. The Test Environment – what hardware, software, data sets, and so on are needed for the test?

V. The Test Deliverables – what data and documents will be available as a result of the test?

Vi. Responsibility for the Test – who designed the test? Who will conduct the test?

Software validation involves **tests for correctness**, which test a software feature to determine if the software feature is working according to specified requirements. Documentation for each test for correctness should clearly state the feature being tested and the pass/fail criteria being used to determine if the software passes the test.

The nature and scope of tests for correctness, and their pass/fail criteria, should be determined by the specified requirements for the software. However, even though the tests are based on the specifications, a testing plan is usually developed after a general design for the software exists and we know what methods need to be tested. This means the development of a testing plan is the last part of software design.

In a professional environment, independent experts will examine the software design and create a list of the items that need to be validated and the features of those items that need to be tested.

Internal software validation is the testing and validating of software by the same group of experts who developed the software. Often this occurs in a classroom environment where students test their own software. In a professional environment, it often occurs when developers test their own software as it is developed, before turning it over to independent experts for formal validation once they are confident the software correctly meets its specifications.

The following example shows us the specifications for a software package that creates a weekly payroll report, and a testing plan for the software.

Example: Testing Plan for Payroll Report Software

Description:

This program should calculate and print a weekly payroll report for hourly employees.

A text data file contains sets of data for employees, each set on five lines with employee number on the first line, first name on the second line, last name on the third line, hourly rate on the fourth line, and hours worked on the fifth line. For each employee, the software should read the data; calculate the gross pay according to the following "time and a half" rule: *pay = hours x rate for hours up to 40 hours, then 1.5 x hours x rate for all hours over 40 hours*; then print a payroll report with employee number,

first name, last name, hours, rate, overtime hours, and gross pay on each line. The report should be well formatted, with headings for each column and all numbers accurate to two decimal places.

The program should work no matter how many employees are included in the data file. It should work with hours up to 100.00 hours per week and pay rates up to 100.00 dollars per hour. Only positive numbers should be accepted as input.

Software Requirements

1. The software should input from a text file of employees with employee number, first name, last name, hourly rate and hours worked on consecutive lines; no limit on the length of the file.
2. Employee numbers are text with four numeric characters
3. All numeric values are accurate to two decimal places.
4. 0 < hours < 100.00 and 0 < rate < 100.00
5. For each employee, calculate the overtime hours and gross pay according to the following rules:
   if hours <= 40 then gross = hours * rate
   and overtime hours = 0
   if hours > 40 then overtime hours = (hours-40)
   and gross = (40 * rate) + (overtime hours * 1.5 * rate)
6. Print a well-formatted report with columns for employee number, first name, last name, hours, rate, overtime hours, and gross pay on each line.

Software Design

The software developers have decided to use two methods – an existing *PayRecord* class, described by the UML diagram listed below, and a new *PayReport* class, described in the pseudocode following the *PayRecord* UML diagram. The *PayRecord* class was validated when it was created.

PayRecord (weekly payroll record for each employee)	
- empNumber: String	Employee Number
- firstName: String	first name of employee
- lastName: String	last name of employee
- rate: double	hourly pay rate
- hours: double	hours worked in current week
- otHours: double	overtime hours worked
- gross: double	gross pay for the current week
+ PayRecord()	null constructor
+ PayRecord(employe number, firstname, lastName)	initializing constructor 1
+ PayRecord(employe number, firstname, lastName, rate, hours)	initializing constructor 2
+ setFirstName(String): void	
+ setLastName(String): void	
+ setRate(double): void	

+ setHours(double): void	
+ CalcPay():void	calculates and sets otHours and gross using the "time and a half" rule
+ getFirstName(): String	
+ getLastName() String	
+ getRate(): double	returns an unformatted double value
+ getHours(): double	returns an unformatted double value
+ getOtHours(): double	returns an unformatted double value
+ getGross(): double	returns an unformatted double value
+toString(): String	returns a string with empNumber, firstName lastName, rate, hours, otHours, and gross on one line separated by commas.

The following pseudocode describes the methods in the *PayReportmain()* class for the software.:

```
void main() // the main has a loop that calls other methods
{
PayRecord current;

  declare Java File object  // for example -- File empFile = new File("payroll.txt");
  declare Java Scanner object // for example -- Scanner infile = new Scanner(empFile);

  while (next line exists)
  {
       call readRecord(infile, current);
       call calculatePay(current);
       call printHeadings();
       call printRecord(current);
  ) // end while (next line exists)

}  // end main()
/*********************************************************************/

void readRecord(Scanner infile, PayRecord current)
{
  current.empNumber = infile.nextLine();
  current.firstName  = infile.nextLine();
  current.lastName = infile.nextLine();
  current.rate  = Double.parseDouble( infile.nextLine() );
  current.hours = Dopuble.parseDouble( infile.nextLine() );
) // end readRecord()
/*********************************************************************/
void calculatePay(PayRecord current)
{
  if (current.hours <= 40)
    {
     current.gross =  current.hours * current.rate;
     current.otHours  = 0;
    } // end if
  if (current.hours > 40)
   {
    current.otHours  = 40 - hours;
    current.gross = (40 * current.rate) + (current.otHours * 1.5 * current.rate)
```

```
   ) // end if
} // end calculatePay()
/*********************************************************************/
void printHeadings()
{
  // code here to print column headings
} // end printHeadings()
/*********************************************************************/
void printRecord(PayRecord current)
{
// code here to format and print a line in the report for the current record
) // end printRecord()
/*********************************************************************/
```

Based on this design, the methods in the *PayRecord* class do not need to be tested because they were previously validated. The details of the *calcPay()* method in the *PayRecord* class are not provided, but the developers have used their own *caculatePay* method instead, so that will need to be tested.

The following tests will be needed:

A unit test of the *readRecord()* method needs see if it properly reads data from the data file into the payrecord() object and if it creates necessary exceptions when the data file is missing or incorrectly formatted.

A unit test of the *calculatePay()* method needs to see if it performs the correct calculations and returns the proper values. It returns values using *pass by reference* for a payRecord object.

A unit test of the *calculatePay()* method is also needed to make sure it does not accept input values out of range for hours and rate, such a negative values.

The printHeadings() method needs to be tested to make sure it prints correct column headings. The specifications for this are not detailed, but the headings should be well formatted and match the data in the columns. The *printRecord()* method also has similar specifications without detail, but the data must be well-formatted and line up with the column headings. These two methods can be tested together because their specifications are linked to one another. This is similar to a unit test of the two methods together.

The *main()* method needs to be tested to see if it performs all tasks as specified. Basically, criteria similar to the criteria for the unit test of the methods needs to be applied to a test of the overall program. This is an integration test of the software package.

Weekly Gross Payroll Report
Testing Plan

I. Introduction

The software package creates a weekly gross payroll report based on data input from a data file. The software is being written and tested internally the developer, a single programmer assigned to the project. Formal independent validation will follow later, once the software passes the internal testing.

The software reads a set of data for an employee, calculates the overtime hours and gross pay according to given formula, then prints a payroll record for that employee as part of the payroll report. The report should be neatly formatted with correct headings.

II. **Items to Be Tested**

The *readRecord()* method – Does it properly reads data from the data file into a PayrRecord() object? Does it halt the program and create necessary exceptions and warning messages when the data file is missing? Does it Does it halt the program and create necessary exceptions and warning messages when the data file is incorrectly formatted?

The *calculatePay()* method – Does it perform the correct calculations and returns the proper values? Does it reject input values out of range for hours and rate, such a negative values?

The printHeadings()and *printRecord()* methods – Do they produce a well-formatted report with headings that match the columns of data and numeric values accurate to two decimal places?

III. **Items Not Being Tested**

The *PayRecord* class is not being tested because it was previously validated and its functioning is expected to match the requirements for the weekly gross payroll report software.

IV. **Tests to Be Conducted**

Seven tests will be conducted.

1. ***readRecord()* input test 1**– The *readRecord()*method will be tested using a correctly formatted data file to see if it properly stores data in a PayRecord() object. The test will be conducted using a driver program to read the data and the PayRecord toString() method to print the data. The program will pass the test if it correctly reads multiple data from the properly formatted data file
2. ***readRecord()* input test 2** – The *readRecord()*method will be tested using an incorrectly formatted data file to see if the program halts and notifies the user. The test will be conducted using a driver program to read the data. The program will pass the test if it halts and displays a message telling the user the file is incorrectly formatted, or if it generates a proper exception telling the user that file cannot be read and displays a message telling the user the file is incorrectly formatted.
3. ***readRecord()* input test 3** – The *readRecord()*method will be tested with the data file missing to see if generates a proper exception. The test will be conducted using a driver program to read the data. The program will pass the test if it generates a proper exception with a meaningful message telling gthe user the file is missing.
4. ***calculatePay()* correctness test –** The *calculatePay()*method will be tested with a driver program and the PayRecord toString() method to see if it performs calculations correctly. The driver program will call the method with a variety of input, including hours less than 40, equal to 40 and over 40. The method will pass the test if all output is correct to two decimal places.
5. ***calculatePay()* data range test –** The *calculatePay()*method will be tested with a driver program and the PayRecord toString() method to see if it rejects data out of range. The driver program will call the

method with hours above and below the accepted data range and pay rates above and below accepted data range. The method will pass the test if it rejects all output out of range.

6. **printHeadings()**and ***printRecord()*** output test – The printHeadings()and *printRecord()* will be teste to gether to see if they produce a well –formatted easy to read report. A driver program with an array of several PayRecord objects populated with data will be used. The driver will call the printHeadings() method, then call the *printRecord()* method for each PayRecord object to be printed. The methods will pass the test if they produce a well-formatted easy to read report, with all numeric values accurate to two decimal places.

7. **integration test –** the *main()* method will be run using a test data file with some records matching data from the *calculatePay()*correctness test and the *calculatePay()* data range tests. The method will pass the test if it produces a well-formatted report, displays properly calculated results, and rejects data out of range.

V. **Environment Needed for the Test**

Hardware - The software will be tested using the same hardware used for development.

Software - The software will be tested using the same hardware used for development. Other software needed includes:

- a correctly formatted payroll data file
- an incorrectly formatted payroll data file
- a driver for the input tests which has a PayRecord object and which calls the *readRecord()*method as needed.
- a driver for the *calculatePay()* correctness test which calls the *calculatePay()*method several times using a proper variety of input including hours less than 40, equal to 40 and over 40.
- a driver for the *calculatePay()* data range test which calls the *calculatePay()*method several times using a proper variety of input including hours above and below the accepted data range and pay rates above and below accepted data range.
- a driver for the printHeadings()and *printRecord()* output test with an array of several PayRecord objects populated with data and which calls the **printHeadings()**method and then the *printRecord()*method for each array element.

V. **Test Deliverables**

A pass/fail report for each test will be provided along with a description of each test the output from the test, and any data files used for the test. If a test is failed, the report will indicate why it failed.

The source code for each test will be also provided.

VI. **Responsibility for the Tests**

All tests will be conducted by the software developer.

6.4 Test Cases

Each test that is to be conducted on software should include a set of test cases. A software **test case** is a well-defined set of input, execution conditions, and expected results that can be used to verify that a

software feature meets a specific requirement. Each test case should have its own clearly defined pass/fail criteria.

Test cases simulate real world conditions in which the software must operate correctly. Designing good test cases is a specialized skill that can be as difficult as designing the software itself. Software failures in the real world often occur under conditions that were not included in the test cases when the software was being validated. Sometimes this is because the requirements were not comprehensive enough, but more often it is because software test designers missed a possible test case.

Here is a simple example from a real world situation. The ABC Company hired a software developer to produce its own custom payroll program. The calculation part of the program needed to calculate overtime pay for workers who worked more than 40 hours, while workers who did not work more than 40 hours were not paid overtime. The software included logic similar to the following:

```
if (hours < 40)
    worker gets regular pay

if (hours >40)
    worker gets regular pay plus overtime pay
```

The logic tells the computer what to do if the hours are less than 40, and if the hours are greater than 40, but not what to do if the hours equal 40. This is how the software was written. It was tested with hours less than 40.0 and with hours greater than 40.0, but it was never tested with exactly 40 hours. The first time the software was run, several workers who worked exactly 40 hours were not paid.

In this example, the software design was flawed, but so was the testing. Good comprehensive test cases based on the software requirements should detect things that software developers may have overlooked.

There is no magic formula for designing test cases, but in general, test cases developers should consider all possible conditions under which a program will run. Nothing should be taken for granted. What if a data file contains a String where an integer should be? What does the software do if a data file is missing, or a required Internet connection goes down?

Test cases that require user input are especially problematic. Thirty years ago, in the 1980's when personal computers were fairly new, the Franklin Institute Science Museum in Philadelphia introduced an interactive exhibit named *SeaTrader*, in which museum visitors could try a simulation on a computer that involved buying a sailing ship in 1769, hiring a crew, and sailing between ports to try to make a profit buying and selling different cargo. The simulation required users to select items from menus, such as selecting cargo from of a list of ten items –deerskins, lumber, wine, iron goods, etc. The simulation was first tested using a computer with a standard QWERTY keyboard. The researchers discovered that 30 percent of the time when users were asked to pick a number between 0 to 9, they pressed a key that was not in the list, and that in some cases the same user repeatedly pressed keys not in the list.

Imagine how this would work in a guessing game between people.

"I am thinking of an integer between 0 and 9. Try to guess what it is."

"W"
"Try again. I am thinking of an integer between 0 and 9. Try to guess what it is."
"Ctrl-Q"
"Try again. I am thinking of an integer between 0 and 9. Try to guess what it is."
"Backspace"
"Try again. I am thinking of an integer between 0 and 9. Try to guess what it is."
"Ctrl-Alt-Del"
"Program terminated."

In some cases users simply hit the wrong key – such as a "Q" when reaching for a "1". In some cases, people used a lowercase "L" instead of a "1", which was common for people trained on typewriters. In several cases, the user was convinced that there was a "secret button" and was trying to find it.

In the end, the Franklin institute developed a custom touch sensitive keyboard for their exhibit with only the keys needed for the simulation.

This example highlights that fact that it is often difficult to anticipate what happens when a program runs, especially if user input is involved. The developers of test cases need to use their imagination to try to anticipate what might happen when a program runs. This ability is developed with experience, and by reading available literature on software test cases. Three popular books about software testing that include discussions of test cases are:

A Practitioner's Guide to Software Test Design, by Lee Copeland; ArtechPress, 2004; ISBN: 978-1580537919;

Software Testing: A Craftsman's Approach (3rd Ed.), by Paul C. Jorgensen; Auerbach Publications; 2008; ISBN: 978-0849374753

Systematic Software Testing, by Rick D Craig and Stefan P Jaskeil; ArtechPress, 2002; ISBN: 978-1580535083

Test Case Criteria and Traceability

Test cases should have clear pass/fail criteria based on specified software requirements, and documentation should exist to trace test cases back to specific requirements.

Criteria for test cases can be quantitative or qualitative. The terms quantitative and qualitative have different specific meanings in different disciplines, so we need to be careful. In chemistry, for example, quantitative analysis means finding the chemical formula for a substance while qualitative analysis means finding what a substance is composed of without necessarily finding its chemical formula.

For our purposes, **Quantitative test criteria** are criteria that can be specified as specific values for variables or properties of an object. The term *quantitative* is associated with numeric variables, but Strings, Boolean and other data types can also be used for quantitative test data.

Qualitative test criteria are criteria that have more to do with the observed qualities of a software product, such as its look and feel or ease of use, rather than specific numeric measurements. There are several strategies for qualitative test criteria.

A customer must be happy with the look of a program as well as its functionality. Such qualitative criteria need to be reduced as much as possible to a clearly defined quantitative criteria for testing purposes. Often this can be done by turning the qualitative criteria into Boolean conditions under which the software will be acceptable. A person or group of persons can be designated to accept or reject the software, reducing the criteria to a Boolean variable, such as "*the software shall pass the look and feel test if it approved by person x*", "*the software shall pass the look and feel test if a majority of the review board x*" vote that it is acceptable. if possible, qualitative criteria such has "look and feel" should be reduced to clearly identifiable and measurable criteria, such as font, back ground colors, etc.

In any case, if a software package is rejected because of quantitative or qualitative criteria, the rejection should indicate as specifically as possible why it failed in terms of test criteria.

As with many aspects of testing, development of test criteria and turning qualitative criteria into quantitative test is a specialized discipline beyond the scope of this course. It is enough for us to understand the issues and to begin to develop criteria to test the software we create based on the specified software requirements.

The testing criteria must be related to specified software requirements. **Requirement traceability** is the ability to trace criteria for testing software features through the design, implementation, and testing phases of software development back to the original software requirements. It works in two directions – from the requirement through to the finished software and from the finished software back to the original requirements.

According to IBM, one if the world's experts in software requirements[4]:

> *Traceability plays several important roles:*
>
> - *Verify that an implementation fulfills all requirements: Everything that the customer requested was implemented*
> - *Verify that the application does only what was requested: Don't implement something that the customer never asked for*
> - *Help with change management: When some requirements change, we want to know which test cases should be redone to test this change.*

The key to good test case traceability is documentation. Test designers need to identify which requirements are being tested by each test case. This is often a many-to-one relationship, meaning that more than one test case (many) is linked to a single requirement (one). In the example Pay Roll Report testing plan in the previous section, the *calculatePay()* correctness test is linked to the requirements for

[4] from the article *Traceability from Use Cases to Test Cases* on in the IBM *developerWorks* libray, on the Web at: http://www.ibm.com/developerworks/rational/library/04/r-3217/

how regular overtime hours and overtime pay should be calculated. There will be several test cases for this test – the test needs to be run with hours less than 40, hours equal to 40, and hours over 40. Each of these test cases is linked to the original requirement.

Developers need to be able to know what tests need to be changed if the original requirement changes, so they need to able to trace requirements through to specific test cases.

6.5 Stubs and Drivers for Software Testing

Stubs and drivers are small pieces of software used to test other software.

A **test stub** is a program that takes the place of a method being called. It usually prints a message to indicate that it was called and may return a preset value. Test stubs allow us to check how a main program calls method and handles data returned from methods without using the called methods themselves.

A **test driver** is a program that calls or runs a method to be tested. It is sometimes referred to as a testing shell. It usually feeds the method dummy input and either prints the output or sends the output to a data file. Test drivers allow us to test a method without running the entire software package that uses the method.

Stubs and drivers can be used for unit tests of software methods. The unit is most often a single method being tested.

Stubs are also useful for simulating methods that cause things to happen in the real world which we do not necessarily want to happen each time we test the software. For example, a control program for a drawbridge over a river might call a method that tells the bridge to open then notifies the bridge console operator when the bridge is open. It would be time consuming and expensive to repeatedly test the software during the development process if the bridge had to really open for each time. A stub could accept the call to the method that opens the bridge then pass dummy output to the console program. Of course, once the software method works correctly, the entire system – bridge hardware and control software – would need to be further tested to make sure the bridge really does open, but the point here is that the method to do so can be tested with a test stub without opening the real bridge.

A list of any stubs and drivers that are needed to test the software should be included in the part of a testing plan that identifies the environment necessary to conduct tests.

Test drivers and stubs are examples of encapsulated software – we do not need to know how a method works to test it, only the input and output parameters for the method. If, according to the testing criteria, a method accepts the proper input and produces the proper output – and does not have any undesirable side effects, such as deleting data, erasing a hard drive, or shutting down the Internet – then it passes the test.

In situations where a method needs to be tested with different input values, several similar drivers may be used one at a time to test a method, each with different input. The same is true for stubs; several stubs that generate different values may be used one at a time to test how the calling method handles the return value in different test cases.

We could also place sets of values for different test cases in an array, and a test driver could be created with a loop that calls the unit being tested for each version of the test. In the example above, the *calculatePay()* method needs to be tested with hours less than 40, and hours equal to 40 and hours greater than 40. It also needs to be tested to make sure it does not try to process negative hours or hours greater than 100. An array of values, such as {-3.00, 27.25, 40.00, 45.00, 400.00) could be used by a driver that runs the *calculatePay()* method for each of these values.

We can see this in the following example.

Example: test driver – sales tax for multiple states

A chain of sporting goods stores has a website that allows customers to purchase items online. Federal law requires them to charge sales tax for online purchases for customers in any state in which one of their stores is located. This becomes complicated because of different sales tax rates in different states and because items such as clothing or food are taxed in some states but not others.

The store has a *taxable* table of data that shows whether each item is taxable in each state. The database also has the sales tax rate for each state. A *salesTax()* method in the software for the website accepts the item number (String), the price of the item (double), and state the customer lives in (String), and then looks up the item and returns the sales tax for that item (double). If the item is not taxable, it returns 0.0 as the sales tax. If that item is taxable for that state, then it returns the amount of sales tax for the item. This method is part of the overall checkout and payment program for the Website.

We need to design a driver to test the *salesTax()* method.

The internal workings of the method seem complicated, but we do not need to know how the method works to test it. In this case, we need to develop a set of test data and criteria for the test. To thoroughly test the method, the data should include combinations of items and several different states – some where the given item is taxable in the given state and some where it is not. We do not need to test every item from every state, because we are checking whether or not the method works with the right input; we are not checking the data tables to see if they are correct – that is a different test.

The test designers would need to work with a subject matter expert –such as an accountant for the chain of stores – to develop a good, comprehensive set of test cases. In our example we will assume that the set includes 12 test cases.

The description of the method tells us that its input includes the following: *String itemNum*, *double price*, *String state*. The output should be the sales tax for the item as a double value. Our driver to test the method will have an array of 12 cases – some combinations of states and items that are taxable and some combinations of states and items that are not taxable.

We will create an array of test objects, each an instance of a class named *TestCase*, that has five properties:

- String itemNum; // the item number being tested in a specific case
- double price; // the price of the item
- String state; // the state of customer in this case

- String correctTax; // the correct tax for this case
- boolean passTest; // true if this case passes the test, otherwise false

The *TestCase* class will have a constructor method that accepts three properties – itemNum, price, state, and correctTax.

Our test driver will create an array of 12 test objects, then call the *salesTax()* method for each test object in a loop. If the value returned from the *salesTax()* method matches the *correctTax* for that case, then the boolean *passTest* property for that case will be set to true. Otherwise, it will be set to false. The driver will also print a message for each case listing whether it passes or fails the test.

The details of the *TestCase* class are not shown. It simply contains constructors and *set()* and *get()* methods for each property. It could be an inner class because it is only intended for use in this instance.

Here is the pseudocode for a *salesTaxTest()* driver that can be used to test the *salesTax()* method:

```
void salesTaxTest()
{

   int i;                  // loop counter
   double calculatedTax;   // the amount of tax calculated by the salesTax() method

   TestCase[]  case = new testCase[12]; // array of test objects

   case[0] = new case(1763, 29.95, PA,  1.80 );
   case[1] = new case(1763, 29.95, DE, 0.0);
   case[2] = new case(1414, 7.95, NY, 0.32);
     . . .  and so on up to case[11]

   for i = 0; i < case.length; i++)
   {
      calculatedTax =
salesTaxTest(case[i].getItemNum(),case[i].getPrice(),case[i].getState() );

      if calculatedTax = case[i].getcorrectTax()
      {
         case[i].setPassTest(true);
         System.out.println("Item " +  case[i].getItemNum()+ " sold in "  +
                            case[i].getState() +" PASS" );
      } // end if
      else
      {
         case[i].setPassTest(false);
         System.out.println("Item " +  case[i].getItemNum()+ "for "  +
                            case[i].getState() +" FAIL ***" );
      } // end else

   } // end for

] // end salesTaxTest()
```

Example: test stub – sounding a burglar alarm

Event drive software for a burglar alarm system needs to sound an alarm and send a message to a control panel if any of the door or window sensors in the system detect an open door or window. The event handler for the system calls a *soundAlarm()* method that accepts a sensor number (int), sounds the burglar alarm, looks up which door or window is open from an array of sensor objects, and returns as a String a message that will go to the control panel displaying the location of the open door or window.

We wish to create a stub to test the overall program without actually using the method that sounds the alarm. Our stub will print a message that say "**** ALARM SOUNDED****" .

The *soundAlarm()* also looks up the name of the open door or window (such as "*second floor bathroom window*") and returns a corresponding String. There is no need for us to know the details about how the *soundAlarm()* method performs its lookup and creates the string to be returned. Our stub will simply return a String for the console message that says "*Sensor number*" + *sn* " *is open.*"

Here is the pseudocode for our stub:

```
String soundAlarmtest(int sn)
{
   String consoleMessage;
   System.out.println(""*** ALARM SOUNDED***")
   consoleMessage = "Sensor number " + sn " is open.");
   Return consoleMessage;
}
```

That's it. Stubs are often very simple to create. They usually just need to accept the required input and return suitable output.

Chapter Review

Section 1 of this chapter introduced the concept of independent validation and verification, described the difference between validation and verification, and listed several parameters for independence. key terms included: validation, verification, independent validation and verification (IV&V), IEEE 1012, technical independence, managerial independence, and financial independence.

Section 2 briefly reviewed a four-phase software development cycle that was introduced in Chapter 2. Key terms included Software Development Cycle, unit test, and integration test.

Section 3 described software testing plans and presented a template for a simplified testing in plan and an example of a software project with a plan based on the template. Key terms included: software testing plan, IEEE 829, test for correctness, and Internal software validation.

Section 4 discussed software test cases, criteria for test cases, and the importance of requirement traceability. Key terms included: test case, quantitative test criteria, qualitative test criteria, requirement traceability.

Section 5 described test stubs and test drivers and how to use them to test software methods. key terms included: test stub and test driver.

Chapter Questions

1. What is the difference between software validation and verification? What is software validation based on? What evidence is examined to verify that software developers are using industry standard practices?
2. Software validation and verification is best done by whom?
3. What does IEEE standard 1012 define? What three parameters for independence does the standard list?
4. What are the four phases of a simplified software development cycle? Which two parts in the process of software development are not included in the cycle?
5. What is included in the design phase of a software development cycle?
6. What two techniques are commonly used for organizing the design of complex software? What are software modules usually coded and tested as in most object-oriented programming?
7. What is a unit test? What is an integration test?
8. Which methods and classes need to be validated and which do not need to be validated in a software development project? When do existing methods need to be re-validated?
9. What is included in a software testing plan?
10. What does IEEE standard 829 define? How does it help us to develop a software testing plan?
11. What should be included in a simple software testing plan?
12. What should be included in a software testing plan's an annotated list of tests to be conducted?
13. What is a test for correctness? What should the documentation for a test for correctness clearly state? What should determine the nature and scope of test for correctness, and their pass/fail criteria? When in the software development cycle should a testing plan be developed?
14. What is internal software validation? What are two situations in which internal software validation often occurs?
15. What is included in a software test case? What do test cases simulate? In general, what should developers consider when designing test cases?
16. What is the difference between quantitative and qualitative test criteria? How should we do to more clearly define qualitative test criteria?
17. What is requirements traceability? According to IBM, what three important roles does traceability play?
18. What is the difference between a test stub and a test driver? What do stubs allow us to do? What else are they useful for?
19. What do test drivers allow us to do? Test driver are also referred to by what other name?
20. How can we use arrays for multiple test cases using similar test drivers?

Chapter Exercises

1. Test Cases – BMI Calculator

   Software to calculate and analyze body mass index (BMI) for adults works as follows:

   - The user inputs name (String), weight in pounds (double) and height in inches (double). Only positive values should be accepted. Weight should be between 50 and 500 pounds. Height should be between 36 and 96 inches. The software should reject input values out of range.
   - the software converts height and weight to metric units (kilograms and meters)
   - the BMI is calculated: $BMI = \frac{weight}{height^2}$
   - output is produced as follows:

```
System.out.println("BMI is " + bmi);
if (bmi < 18.5)
  System.out.println("Underweight");
else if (bmi < 25)
  System.out.println("Normal");
else if (bmi < 30)
  System.out.println("Overweight");
else
  System.out.println("Obese");
```

   The software has a modular design, with methods to:

   - get name
   - get height
   - get weight
   - calculate BMI
   - display the result.

   The main method primarily ties the other methods together.

   Your task is to create a comprehensive set of test cases for the software. They should be submitted as a Word document.

2. Test Case for a method from a chess program

   Create a comprehensive set of test cases for the *knightChecker()* method in exercise 6 below. They should be submitted as a Word document.

3. Test Stub for a method from a Monopoly game

   a GUI based monopoly game draws the Monopoly Board, player tokens, houses, etc. on the screen. One of the methods in the game, the *newLocation()* method, will accept the roll of the dice and the player's old board square, then calculate the new board square. If it is a special square type, such as "GO" it should call the *SpecialSquare()* method. Otherwise, it should then display the square enlarged on the

screen showing the owner, the value of the property, the rent, the number of houses and the number of hotels on the square, and if the property is mortgaged. It gets this information about the square from an array of BoardSquare objects that have properties for the necessary information. The data in these properties is updated by other parts of the game.

There are 40 board squares numbered 0 to 39 The new board square = (the old board square+ the roll of the dice) mod 40. The values are integers.

Your task is to write a test stub to take the place of the *newLocation()* method in a test of the method that calls this method. The test will be conducted before all of the graphics are ready, so your method should simply accept the old board square and print the information about the new board square. You can assume that the array of board squares has been loaded with the proper data before the test.

You should submit the source code for the stub.

4. Test Stub for landing gear software

The *landingGearDown()* method in the software for the controls of a new "*fly-by-wire*" aircraft should lower the landing gear for the aircraft, but only if the plane is below 10,000 feet and travelling less than 250 miles per hour. It should send a specific warning message to the control panel if the plane is too high, if the plane is travelling too fast, or if the landing gear don't lower into position correctly. If everything works correctly and the method does lower the landing gear, it should send a message to the control panel saying that "The landing gears is lowered."

Your task is to create a testing stub for the software that accept the correct input, prints a message saying "landing gear lowered" instead of actually lowering the landing gear, and return the message for the control panel as a String. The method header is:

String landingGearDown(double speed, double altitude)

You should submit the source code for the stub.

5. Test Driver for a palindrome checker

A palindrome is a String that is the same frontwards as backwards. Character palindromes are exactly the same, character=by character, forward and backward. For example "otto", "radar", and "able was I ere I saw elba" are character palindromes. The PalindromeChecker() method will accept a String and then print a message telling us whether or not the string is a palindrome. The method should only accept Strings that use alphabetic characters and blank spaces. It should not accept Strings that have numeric characters or punctuation marks.

The method header is:

String PalindromeChecker(String inString)

the method will return one of three messages:

if *inString* contains numeric digits or other characters that are not alphabetic or blank space characters the message will be "*The input string is not alphabetic.*"

if *inString* is alphabetic but not a palindrome, the message will be "*The input string is not a palindrome.*"

if *inString* is alphabetic and is a palindrome, the message will be "The input string is a palindrome."

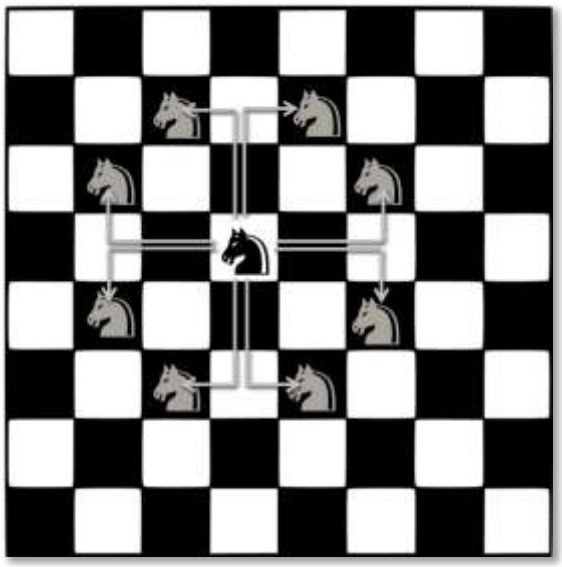
Your task is to construct a test driver for the *PalindromeChecker* () method. The driver will have an array of strings hardcoded into the program with test data. There are four test cases – an alphabetic palindrome, a palindrome that has numeric characters (illegal input), a string with numeric characters that is not a palindrome, and a string that is a palindrome. The array for the test driver should have at least two examples of each.

The driver should contain a loop that will call the PalindromeChecker () method for each string from the array, then print the string being checked and the output message. The results will be checked visually to see if they are correct. Your driver simply needs to display the results neatly.

You should submit only the source code for the test driver method.

6. Test Driver for a Method from a Chess Game

The *knightCheck()* method in a chess program needs to check the movement of a knight to see if the proposed move is a legal move. A knight may move two squares in any one direction and then one square in a perpendicular direction, as shown below.



The method header is:

Boolean knightCheck(int rowBegin , int colBegin, int rowEnd, int colEnd)

rowBegin and *colBegin* indicate the starting location of the knight. *rowEnd* and *colEnd* indicate the ending location of the knight. All values must be between 1 and 8 (1 <= n <= 8).

The method returns true if the move is legal and false if it is not. A move is legal if all values are in the correct range (1<= n <=8) and:

absolute value of (rowBegin – rowEnd) is 2 and absolute value of (colBegin – colEnd) is 1
or
absolute value of (rowBegin – rowEnd) is 1 and absolute value of (colBegin – colEnd) is 2

Your task is to construct a test driver for the *knightCheck*() method. The driver will have an array of TestCase objects, hardcoded into the program with test data. A TestCase object will contain six properties:

int rowBegin , int colBegin, int rowEnd, int colEnd, Boolean correctResult, and Boolean actualResult

The TestCase class should have *get* and *set* methods for each property, a null constructor, and a constructor that accepts five parameters:

int rowBegin , int colBegin, int rowEnd, int colEnd, and Boolean correctResult

Your driver should contain a loop that calls the *knightCheck*() method for each case in the array, then compares the value the method returns to the *correctResult* for that case. It should update the *actualResult* property and print a line for the test case showing us the data for the test case and telling us whether or not the *actualResult* matches the *correctResult* for that case.

You should create several good test cases, several test cases with bad results but values in bounds, and at least one test case that starts in bounds and ends out of bounds.

You should submit the source code for the test driver method, and a UML diagram of the TestCase class.

7. Test Plan for a BMI Calculator

   Create a test plan for the BMI calculator software described in exercise 1, based on the simplified testing plan template in the chapter.

   Submit your plan as a Word document.

8. Test Plan for course grading software

   Software to calculate and display grades for a high school course works as follows:

   The software has a *Student* class with the following properties:

   - student number (String)
   - last name (String)
   - first name (String)
   - an array of scores on tests 1 through 10 (array of integers)
   - final grade (double)

   The Student class has constructors, get and set methods for each property, and the following:

   - a method to let the teacher input a grade for a single test
   - a method to calculate the final grade (an average of all 10 tests)
   - a method to display a histogram (bar chart) of all 10 test scores grades for any student

   The method that prints a histogram calls the *histogram()* method from the exisiting DataGraph Class which has been professionally validated.

All test score and the final grade are integers between 0 and 100 (0 <= n <= 100).

The main method for the software ties everything together. The class that contains the main method, the *Grading* class, has an array of student objects and the following methods:

- a method with a menu for the teacher that calls other methods in the Grading class or Student class as needed. It also has an option to close the program
- a method that loads the array of Student objects from a data file
- a method to save the array of student objects to a data file
- a method to display results for any test or the final grade as a table, sorted by last name
- a method to display the grade distribution for any one test as a pie chart

The method that prints a pie chart calls the *pieChart()* method from the exisiting DataGraph Class which has been professionally validated.

Your task is to develop a simplified testing plan for the software based on the simplified testing plan template and sample in the chapter.

You should submit a word document with the testing plan.

9. Testing Standards

The following IEEE standards all have some relationship to software design and testing:

- *IEEE 730 Standard for Software Quality Assurance Plans*
- *IEEE 828 Standard for Configuration Management in Systems and Software Engineering*
- *IEEE 829 Standard for Software and System Test Documentation*
- *IEEE 1012 Standard for System and Software Verification and Validation*
- *IEEE 1016 IEEE Standard for Information Technology-- Software Design Descriptions*

In October of 2013 a new joint international standard *ISO/IEC/IEEE 29119 Software Testing,* which directly addresses issues of software testing, was published.

How are each of the IEEE related to software testing? The trend is toward new joint international standards, such as *ISO/IEC/IEEE 29119.* How are these new standards different from the IEEE standards? What other joint international standards are related to software design and testing? What other joint international standards are related to other aspects of software development?

10. Software testing certifications

Villanova University recently announced a course to prepare software development professionals for ISTQB (International Software Testing Certification Board) Certification. What is the ITQSB? What certifications do they have available? What are the pre-requisites for taking their certification exams?