Parameterized Random Test Data Generation

Bj Rollison
Bj.Rollison@Microsoft.com

Abstract

Testing is the most challenging job in software development. The software tester must select a small number of tests from countless possible tests and perform them within a limited period of time, often with too few resources. Additionally, tests usually employ only a fraction of the possible data that may be used by customers or by malicious users. Whether we are unit testing, testing an API, or executing end-to-end user scenarios or acceptance tests the test data is usually the keystone to many functional tests.

Testers often craft test data representing typical customer inputs, as well as invalid data for a given input control or parameter. But, defining a broad set of test data from all possible inputs for either positive or negative testing is often a non-trivial part of the testing effort. Also, while static test data is useful, the effectiveness of static test data wears out with repeated use in subsequent iterations of a test in which that data is used.

One possible way to increase the breadth of test data coverage is to use random test data. But, random test data is sometimes disregarded because it may not "look like" customer data, or random data may generate false positive results indicating a failure in the system due because of invalid constructs in the random test data. This paper explains the fundamental principles of parameterized random test data generation which can be used to overcome many of the problems associated with random test data. It also demonstrates how parameterized random test data can increase test coverage and expose unexpected issues in software.

Biography

Bj Rollison is a Principal SDET Lead at Microsoft, currently leading a team responsible for testing the foundation services integrating social networking features on the Windows Phone. Bj started his professional career in the computer industry in 1991 building custom hardware solutions for small and medium sized businesses for an OEM company in Japan. In 1994, he joined Microsoft's Windows 95 international team, and later moved into a test management role in the Internet division working on Internet Explorer 3.0 and 4.0 and several web-client products.

As the Director of Test and a Test Architect in Microsoft's Engineering Excellence group Bj has taught thousands of testers and developers around the world. Bj also teaches software testing courses at the University of Washington, and is a frequent speaker at international software testing conference, and is co-author of the book How We Test Software At Microsoft. Bj is currently interested in the application of random test data generation to increase test effectiveness of test designs through variance in test data, and some of his tools can be found at http://www.testingmentor.com.

Introduction

Test data is a keystone of functional tests designed to evaluate specific computational logic, or error handling behavior. Test cases require test data that is representative of the specific input domain for either positive or negative testing. But, unfortunately in many cases the test data commonly used in testing usually represents a relatively small sample of the total population of all possible test data variations for even a single input field.

For example, any non-trivial input variable the number of possible permutations of the data elements in test data is virtually infinite making it impractical to test all possible input variations. For example, a valid computer name for the NETBIOS protocol may be composed of up to 15 of the 82 valid alphanumeric ASCII characters. The number of possible permutations of a NETBIOS name is $82^{15} + 82^{14} + 82^{13}\ldots + 82^{1}$ or a total of 51,586,566,049,662,994,687,009,994,574 possible test data variants.

At 1 test per millisecond it would take over 6000 years to test every possible permutation of allowable inputs for a NETBIOS name. Exhaustive testing is not practical so the tester must decide how to choose a subset of test inputs from the total population of possible inputs that will potentially expose errors or deviations from the expected behavior, and increase confidence that other input values from the same domain would have similar results to the set of inputs used in a given test.

The test data frequently used in tests usually comes from files or databases of static test data, or from direct input via the keyboard. Static test data is usually compiled from various sources including:

- actual customer data
- data that caused errors in the past, or have caused problems in similar situations
- data based on unique or specialized system knowledge
- intuition

Customer data is valuable because it is usually representative of real or real-like customer data. Customer data may come directly from customers, or from domain or business experts. Test data that has exposed problems in the past can reveal problems in other areas, or help verify the robustness of the software in its ability to deal with problematic input values in the right context. Many professional testers may also have specialized knowledge, or understand patterns of attacks that have proven useful in exposing issues in the past in similar context. Even a tester's intuition in defining test data may occasionally reveal previously undiscovered issues.

Static test data provides value in terms of increased confidence and in some cases may improve early defect detection of common data handling errors. However, the effectiveness of static data used over and over tends to diminish in its ability to provide new useful information to the tester. In other words, relying too heavily on static data contributes to the pesticide paradox (Beizer 2009). Once we test the sets of static test data the effectiveness of that data in providing new useful information diminishes in subsequent iterations of the test case in which that data is used.

Randomizing the input variables is one approach to prevent test data from becoming stale or prevent diminishing value gained from reusing static test data. Increasing randomness reduces uncertainty. But, manually choosing a set of possible variables for a given input may not provide a representative sample of the whole population of input variations. Automated random test data generation can effectively improve coverage of the input variations used in a given test. Automated random test data generation can improve the set of test data from all possible variations used to evaluate a program's functional capabilities.

But, simple randomness may not provide an adequate solution to the test data problem in all situations. For example, we cannot simply generate a random number that is composed of 15 numerical digits to test a web form that requires an American Express credit card number. A recklessly generated random number might not satisfy the criteria for a valid American Express card number and result in a false positive outcome of a test.

Automated random test data generation starts with a model of the input variable for the specific domain. Randomly generated test data from a model cannot be predicted exactly. However, we can constrain certain aspects of randomly generated test data by parameterizing the test data attributes. Automatic generation of randomized test data that is a representative sample from all possible permutations is accomplished by selecting specific properties of the modeled test data attributes from equivalent partitions.

Parameterized random test data generation from equivalent partitions is a systematic approach to automated test data generation. By using parameterized equivalent partitions and simple genetic mutation the tester can generate test data that that is an unbiased representative sample from all possible permutations for a given input. Probabilistic random test data can expose unexpected anomalies, significantly reduce redundancy of test data, reduce exposure to risk, and increase overall confidence in test coverage.

1. Modeling Test Data

The key to effective automated random test data generation is the ability to mimic the input domain space. Some input domains are a simple; for example a range of integers between 1 and 100. But, other types of inputs such as credit card numbers are not as simple as a range of sequential values. For complex input domain spaces a model of the input domain helps identify important attributes of input variables. Reliable automated random test data generation is based on a model of the input domain.

1.1 Input Domain Schema

Some input domains such as email addresses, uniform resource locators (URL), and credit card numbers have various constraints in how they are formed. Also some complex data types are composed of 2 or more components combined to formulate a valid (or invalid) input. Modeling the organizational pattern and creating a schema of the data helps to accurately generate random data for complex data. For example, let's define the schema for credit card numbers. Credit card numbers are non-sequential, and not all number permutations are valid credit card numbers. So, we should begin by decomposing the credit card number into its unique components. Figure 1 illustrates the schema for credit card numbers.

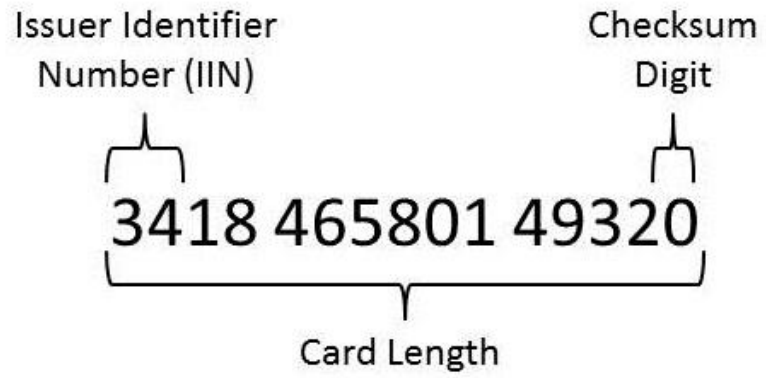

Figure 1. *Credit card number data schema*

The schema for credit card numbers includes:

- Issuer identification number
- Checksum value
- Total number of digits

The first set of digits is the Issuer Identification Number (IIN) used to identify the issuing institution. The IIN values are specific numbers that can be from 1 to 6 digits in length. The last digit of a credit card number is the check sum. All credit cards use the Luhn algorithm (Wikipedia) to validate the credit card number sequence. (The China Union Pay credit card is an exception and the checksum algorithm is unknown at this time.) Finally, credit card numbers can have a length between 12 and 19 digits. For example, Laser, Maestro, Solo, and Switch have card numbers with variable lengths, but, most card issuers use a specific length.

1.2 Data Equivalent Partitioning

Equivalence partitioning based on mathematical set theory has long been used in software testing to categorize input and output variables into equivalent subsets for testing data. The principle assertion of equivalence class partitioning is that any element from a given set has an equal probability of producing the same outcome or behavior as any other element from the same set. Glenford Myers (Myers 1979) stated "…identifying the equivalence classes is largely a heuristic process." This is especially true from a "black box" perspective because the tester doesn't have 'visibility' into the internal structure of the code that parses or manipulates the data.

The heuristics for identifying equivalent partition sets and subsets of input or output variables include:

- **Range** –a linear range of values. For example, integer values from 1 through 100, or upper case ASCII characters A through Z.
- **Unique** – values in a set that may be handled differently. For example, an asterisk character and a forward slash character are both invalid filename characters on the Windows platform, but each character is processed differently by the file I/O APIs. The forward slash will throw an exception resulting in an error message; the asterisk does not throw an exception but changes the state of the listview control in the Save As dialog.
- **Group** – distinctive values in a set. For example, upper case and lower case characters, or the IIN numbers 34 and 37 are a distinctive group of values for the American express card from the larger set of IIN values.
- **Specific** – value that "must be" or "must not be." For example, A password might require at least on upper case character and one number, or some email addresses must not start with a number.

Partitioning data without an explicitly detailed understanding of the algorithm that parses or manipulates the data is not always easy. To be most effective testers should have knowledge of the overall system (hardware, operating system, environment, etc.), the specific domain (client program, programming language used to develop the application, etc.), and the data schema. Limited knowledge or the inability to adequately analyze the data may result in incorrect subsets that ultimately impact the value of randomly generated data from the model of parameterized partitions.

In our example of credit card numbers we should define equivalent partitions of the schema components for each issuing institution. Table 1 illustrates the valid class data partitions for each component of the schema for a small set of credit card types.

Card	IIN	Length	Checksum
American Express	34, 37	15	Luhn
MasterCard	51-55	16	Luhn
Solo	6334, 6767	16, 18, 19	Luhn
VISA	4	16	Luhn

Table 1. *Sample equivalence partitions of some common credit cards*

For each card type in our example, our equivalent partitions include the group of valid values or the value for the IIN, the value or group of values for a valid length, and the specific checksum algorithm. Of course, the only way to validate the equivalence hypothesis is to test the entire population of each set, including all permutations of each number for valid or invalid inputs. But, exhaustive testing is impossible in most situations. So, instead we can generate random samples of the possible set of all data based on our model of that data. Random sampling from the entire data set can increase the breadth of test data coverage, and statistically improve overall confidence that other values in the data set would also produce the same result.

2. Generating Random Test Data

There are numerous random number generation algorithms available in various programming languages. Some programming languages such as Java, Ruby, and C# also have built in class library functions for random number generation. Computer algorithms are actually pseudo-random as compared to “true” random. True randomness is based on some physical phenomenon such as the roll of dice. Pseudo-random generators are initialized with some predetermined value as a starting point such as the system time, and they use mathematical algorithms to select values from a finite set. So, we must ask, are pseudo-random algorithms are reasonably random enough for most generating most types of test data?

The scatter chart in Figure 2 illustrates 5000 random numbers in the range of 0 through 2,147,483,647 using the pseudo random number generator in the System.Random class in the .NET framework. The System.Random class pseudo-random generator uses a subtractive random number generator algorithm introduced by Donald E. Knuth (Knuth 1981).

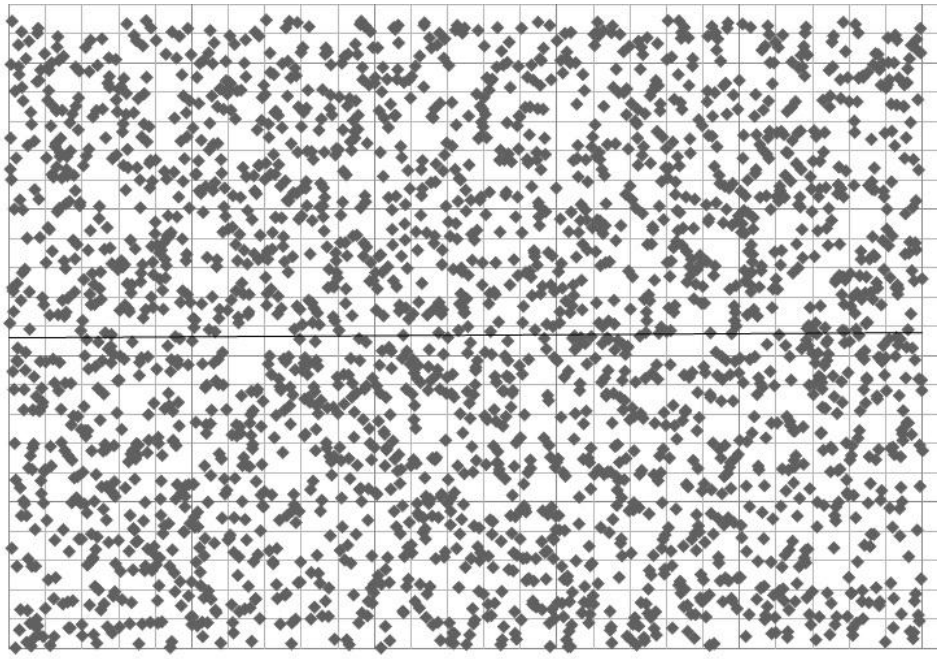

Figure 2. *Scatter chart of 5000 randomly generated integer values*.

The random numbers generated by the System.Random class in Figure 2 appear reasonably random enough for random generation of most common test data. It also appears that this set of 5000 values includes a broad set of data points from the entire set of possible values. Statistically speaking, testing 5000 values out of the total population of 2,147,483,648 would provide a confidence level of greater than 99% with a sampling error of ±3%.

Jeff Atwood (Attwood 2006) compared the 2 pseudo-random generators in the .NET framework; the System.Random class, and the System.Security.Cryptography.RandomNumberGenerator (Attwood 2006). Based on his comparison he also concluded that pseudo-random generators are "perhaps sufficiently random for practical purposes." The Cryptography class pseudo random generator uses multiple environmental factors as compared to the Random class pseudo random generator which uses the system clock. However, we cannot seed the Crypography class pseudo random generator which is critical for random test data generation.

2.1 Seeding Pseudo Random Generators

Software developer and consultant Jonathan Kohl (Kohl 2006) discussed a situation in which his client's excessive use of random data made it virtually impossible to track down failures. A common problem with random data generation is the inability to reconstruct the randomly generated test data in case there is an unexpected outcome. By seeding a pseudo random generator we are able to reproduce randomly generated test data. For example, we can use an integer values as a constructor when we instantiate a new instance of System.Random in C# so we can reliably reproduce randomly generated test data.

Using the same seed value would generate the same random test data each time. So, instead of using a hard-coded seed value, or having to pass a seed value as an argument we can generate a random seed value for each test. The randomly generated seed value is used to seed the pseudo random generator used to generate the test data. The randomly generated seed value should be logged in order to reproduce a random sequence in case of unexpected behavior. The code sample below is one possible implementation of how to instantiate a new instance of Random by either providing a seed value, or by randomly generating a seed value.

```
interface IPseudoRandomGenerator
{
  Random GetPseudoRandomGenerator(out int seedValue);

  Random GetPseudoRandomGenerator(int seedValue);
}

public class PseudoRandomGenerator : IPseudoRandomGenerator
{
  /// <summary>
  /// Generates pseudo random object using a randomly generated seed value
  /// </summary>
  /// <param name="seedValue">Out integer value to log randomly generated
  /// seed value variable for reproducibility of random generated data</param>
  /// <returns>Pseudo random object</returns>
  public Random GetPseudoRandomGenerator(out int seedValue)
  {
    Random randomSeedGenerator = new Random();
    seedValue = randomSeedGenerator.Next();
    return GetSeededPseudoRandomGenerator(seedValue);
  }

  /// <summary>
  /// Generates pseudo random object with a specified seed value
  /// </summary>
  /// <param name="seedValue">Integer value to calculate starting value
```

```
    /// for the pseudo random number generator</param>
    /// <returns>Pseudo random object</returns>
    public Random GetSeededPseudoRandomGenerator(int seedValue)
    {
      Random pseudoRandomObject = new Random(seedValue);
      return pseudoRandomObject;
    }
  }
```

In this example, the GetPseudoRandomGenerator() method instantiates a new instance of a pseudo-random generator and uses the .Next() method to generate a random seed value.It also provides the out parameter so we can log the randomly generated seed value. It then calls the GetSeededPseudoRandomGenerator() method and returns the pseudo-random generator object for use in our test. By directly calling the GetSeededPseudoRandomGenerator() method and passing a given seed value we are able to reproduce randomly generated test data.This approach is an effective way to improve variability of random test data without having to manually seed a random test data generator. It also enables the tester to exactly reproduce specific test data if the randomly generated data exposes an error.

2.2 Generating Random Test Data From Equivalent Partitions

Equivalent partition sets define possible values for use in each component of complex data. Since each possible set of data can be large we can increase variability by randomly selecting values from each equivalent partition. Then by combining randomly selected values from equivalent sets for each component in the data schema we can form random test data based on the model of a specific input domain.

For example, to generate a valid American Express card number a value of 34 or 37 is randomly selected from the equivalent set of valid numbers for an American Express IIN. Next, 13 digits between 1 and 9 are appended to the IIN to satisfy the 15 digit length requirement. Finally, the randomly generated card number must be validated against the Luhn algorithm to produce a valid or invalid card number.

The code sample below provides a simplified implementation for generating a specific credit card based on the creditCardType parameter. This also illustrates how a parameterized value from the larger equivalent set of all valid financial issuing institutions is used to control the random test data to produce a credit card number that begins with either a 34 or a 37 if we pass the appropriate argument for American Express to the creditCardType parameter.

```
  public interface ICreditCardNumberGenerator
  {
    string GetRandomCreditCardNumber(
      Random randomObject, int creditCardType, bool isValidNumber);
  }

  public class CreditCardNumberGenerator
  {
    public string GetRandomCreditCardNumber(
      Random randomObject, int creditCardType, bool isValidNumber)
    {
      // Instantiate class containing credit card data partitions for each card type
      CreditCardDataPartitions cardData = new CreditCardDataPartitions();

      // Get issuer identifier numbers and card lengths by selected card type
      int[][] creditCardData =
        cardData.GetCreditCardPartitionedDataSets(creditCardType);

      // Get a string representing the issuer identifier number if more than 1 IIN
      // for selected card type
```

```
      string issuerIdentiferNumber =
        creditCardData[0][randomObject.Next(cardData[0].Length)];

      // Get the length of the selected credit card type if more than 1 length for the
      // selected card type
      int creditCardLength = creditCardData[1][randomObject.Next(cardData[1].Length)];

      return GenerateCreditCardNumber(
        randomObject, issuerIdentiferNumber, creditCardLength, isValidNumber);
    }

    private string GenerateCreditCardNumber(
      Random randomObject,
      string issuerIdentiferNumber,
      int creditCardLength,
      bool isValidNumber)
    {
      StringBuilder creditCardNumber = new StringBuilder(issuerIdentiferNumber);

      while (creditCardNumber.Length != creditCardLength)
      {
        creditCardNumber.Append(randomObject.Next(9));
      }

      return ValidateNumber(creditCardNumber, isValidNumber);
    }

    private string ValidateNumber(StringBuilder creditCardNumber, bool isValidNumber)
    {
      if (!IsValidLuhnNumber(creditCardNumber) && isValidNumber)
      {
        creditCardNumber = MakeValidCreditCardNumber(creditCardNumber);
      }
      else if (IsValidLuhnNumber(creditCardNumber) && !isValidNumber)
      {
        creditCardNumber = MakeInvalidCreditCardNumber(creditCardNumber);
      }

      return creditCardNumber;
    }
  }
```

In this sample implementation we begin by instantiating the class containing all of our defined equivalent data partitions for credit card numbers. Next we declare a multi-dimensional array to store the equivalent data sets for the IIN numbers and the card number lengths for our selected card type. The pseudo random generator (randomObject) randomly selects an IIN and card length if more than one value are stored in each array. Finally, a call to the GenerateCreditCardNumber() method appends the remaining digits to the IIN number to satisfy the credit card number length argument value. Since this implementation appends a random number to the randomly generated test data, we also need to validate whether or not this credit card number satisfies the isValidNumber parameter.

2.3 Random Test Data Fitness

In the *Encyclopedia of Software Engineering,* Hamlet (Hamlet 1994) points out “Any testing method must rely on an oracle to judge success or failure of each test point.” Reliable oracles are a common problem with random test data generation. Hamlet also notes that random testing is impossible without an effective oracle. So, a method that generates random test data should also ensure that test data satisfies the fitness requirements for a mechanical oracle to be most effective.

In our example, the randomly generated credit card number must pass the Luhn algorithm if the isValidNumber parameter is true. The oracle for our credit card number to validate its fitness is the IsValidLuhnNumber() method. One possible implementation of the Luhn algorithm is illustrated with the code sample below.

```
/// <summary>
/// This method determines whether a number satisfies a Luhn Number
/// </summary>
/// <param name="number">Array of integers used in the number</param>
/// <returns>True if the number satisfies the Mod10 checksum; otherwise
/// false.</returns>
public bool IsValidLuhnNumber(string creditCardNumber)
{
  // Convert string to int array
  int[] number = ConvertStringOfNumbersToIntArray(creditCardNumber);

  int[] temp = new int[number.Length];
  Array.Copy(number, temp, number.Length);

  // The luhn algorithm
  int sum = 0;
  bool checkBit = false;
  for (int i = temp.Length - 1; i >= 0; i--)
  {
    if (checkBit)
    {
      temp[i] *= 2;
      if (temp[i] > 9)
      {
        temp[i] -= 9;
      }
    }

    sum += temp[i];
    checkBit = !checkBit;
  }

  // if the modulus == 0 it is a valid Luhn number; else invalid
  return sum % 10 == 0;
}
```

In this case ‘valid’ implies the card number satisfies the IIN, length, and Luhn algorithm requirements. So, the card number “looks and feels” like a valid number. Each randomly generated credit card number represents one possible number out of the entire population of valid card numbers that can be used by the issuing institution.

For negative testing, or testing for invalid credit cards we would need to add additional parameters. For example, one invalid random credit card number might include an invalid IIN, but the length and checksum requirements are satisfied. Another variation might be a valid IIN for the card type along with a valid length but it checksum digit would not pass the Luhn algorithm. Additional variations for invalid credit card numbers should be tested.

2.4 Random Test Data Fitness

If our desired test data is a valid credit card number and the randomly generated number does not satisfy the requirements for the Luhn algorithm then it calls the MakeValidCreditCardNumber() method. The MakeValidCreditCardNumber() method is an algorithm that mutates the last digit of the randomly

generated credit card number to a valid checksum value to satisfy the Luhn algorithm. Mutation is one way we can further programmatically craft randomly generated test data to contain specific properties.

Mutating test data uses simplified genetic algorithm (GA) principles. A GA is an optimization technique to produce exact (or approximate) solutions to a problem. The key elements of a genetic algorithm include:

- An abstract representation of the population for a given solution
- A fitness function that evaluates the solution for 'correctness'

Basic GA principles can be used to produce or 'evolve' test data that is representative of the population of possible inputs via recombination or mutation. As already demonstrated the tester can use parameterized equivalent partitions to generate an "abstract representation" of test data. But, sometimes we might need to "mutate" test data to satisfy particular requirements; especially for negative or known problematic testing scenarios.

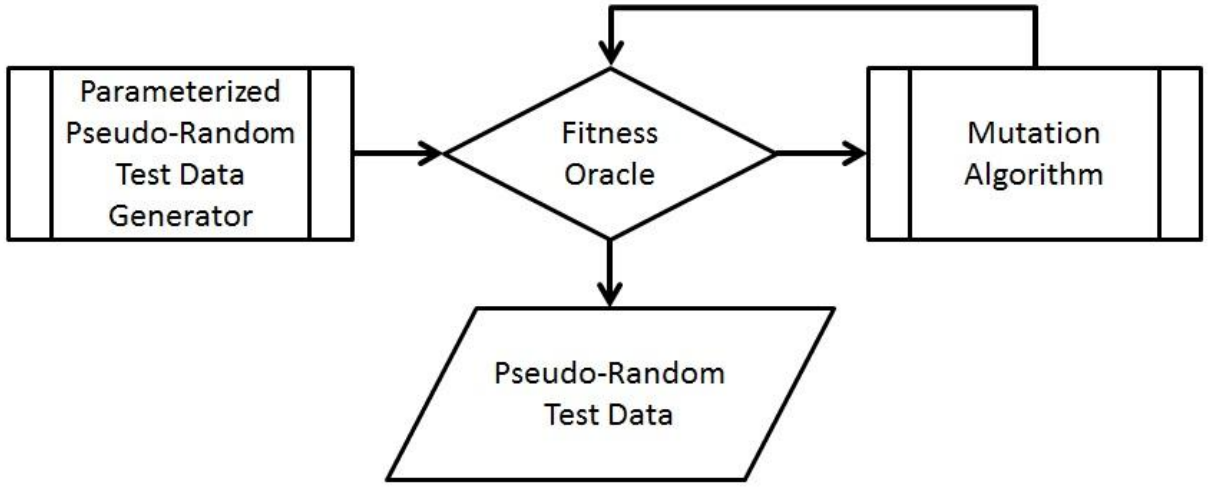

Figure 3. *Simple model of 'genetic' mutation for parameterized random test data generation*

For example let's assume you need a specific permutation of the test data such as a valid checksum value, or you want to generate a random string with a specific Unicode character in a specific location in a string. In these situations parameterized pseudo-random test data generation may not provide the specific outcome. However, we can still use pseudo-random data generation to produce an abstract representation of the data based on the specified properties. The random test data is checked for fitness using a heuristic oracle. If the random test data does not satisfy the desired criteria then the random sequence is passed to a mutation algorithm that mutates the sequence. The mutated test data is passed back to the proto-oracle until the test data satisfies all the criteria for 'correctness' specified by the tester.

This approach uses parameterized equivalence partitions to generate probabilistic random test data, and then use simple 'genetic' engineering principles to mutate the test data (if necessary) until the specified fitness criteria is satisfied for all conditions. Mutating randomly generated test data is especially useful for crafting negative test data variants. For example, we might want to generate a string of random Unicode characters equal to max length, but then replace the last character in the string with a Unicode surrogate pair character which is actually composed of 2 Unicode code point values which may result in a buffer overflow.

3. Pseudo-Random Test Data Generation Library

Pseudo-random test data generators can be used in both automated testing and manual testing. Pseudo-random test data generators are just tools that can help testers increase test data coverage by efficiently generating variable test data that satisfies specific properties. In some cases we might only need one type

of test data such as a random string of characters. But, sometimes we may need test data that resembles a more complete profile of a fictitious person for testing web forms, or sign in to a new social network, a medical records system, or contacts on a phone. In these situations we would need several different types of random test data.

For example, a new contact on your phone might include a name, a phone number, a cell phone number, an address, an email address, a birthdate, notes, and a website URL. Instead of generating random test data for each type of input variable a pseudo-random test data generation library can include a persona abstraction layer. A persona is simply a collection of various types of randomly generated test data.

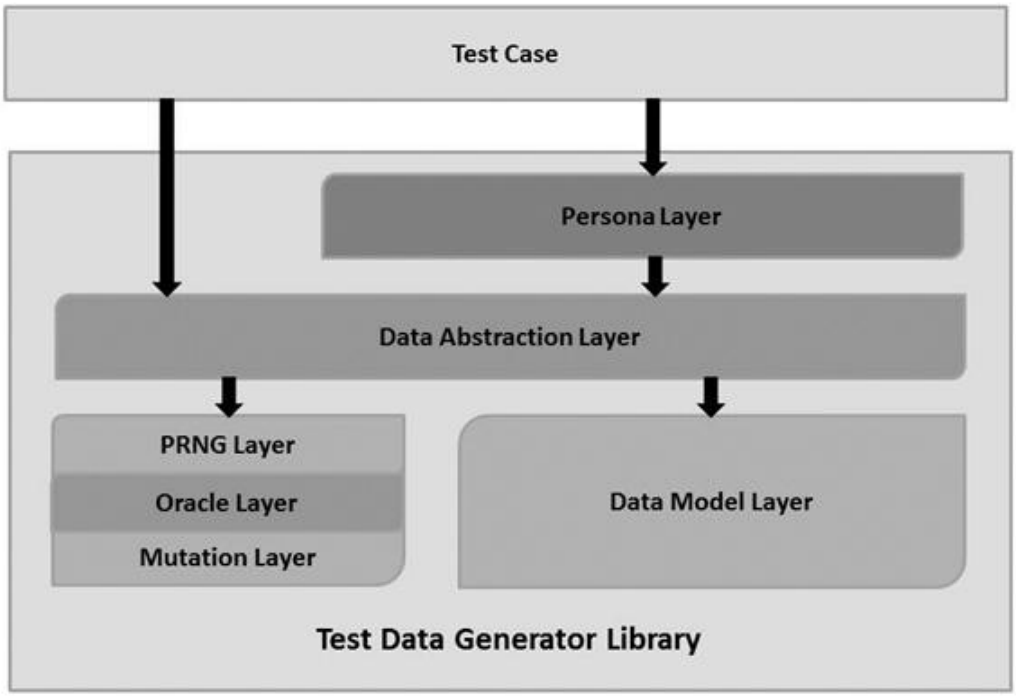

Figure 4. *Architectural layer diagram for a pseudo-random test data generation library*

The Persona layer provides a high level interface that models one or more customer profiles and generates the information or data for that customer profile. For example, a new mobile phone contact may have any or all of the following types of pseudo-random test data:

- Name
- Phone number
- Mobile phone number
- Email address (business)
- Email address (personal)
- Street address
- Website URL
- Birthday (date)
- Anniversary (date)
- Significant other (name)
- Children (names)
- Office location
- Job Title
- Notes
- Ringtone (randomly selected from collection)

The automated test developer can call pre-defined customer personas with any combination of these individual data elements. All these types of test data can be randomly generated and combined in

different patterns to automatically generate any number of new contacts on a mobile device for testing purposes. Of course, different software systems might need different types of information about its customers. Medical records might require social insurance numbers or national identification numbers, and medical insurance numbers.

Occasionally, the automated test developer may not require a complete customer persona, and only require one type of test data such as a date, or a credit card number. The data abstraction layer is the interface for generating various individual types of pseudo-random test data available to the automated test developer and any associated properties that are used to parameterize the attributes of the pseudo-random test data. Table 2 illustrates several attributes that may be found in a pseudo-random Unicode string class in data abstraction layer.

Pseudo-Random Strings
+ Name
+ Address
+ Phone number
+ Date
+ Email address
+ URL
+ String of Unicode characters
+ String 'looks like' sentence
+ Number
+ National identification number
+ Credit card number
+ Driver's license number

Table 2. *Simplified UML class diagram for data abstraction layer.*

The foundation of this pseudo-random test data generation library is the data model layer. The data model layer contains the collections of data partitioned into equivalent subsets for all of the various types of test data. For example, the data model layer contains the collections of all valid issuing identification numbers for credit card institutions. It also contains equivalent partitioned sets of invalid characters in a local email address according to the appropriate RFC and/or various hosting organizations. It may also contain locale specific collections of localized given and surnames.

The pseudo-random number generator (PRNG) layer, the oracle layer, and the mutation layer provide private supporting methods for generating a random object, methods that act as oracles to check the validity of certain types of pseudo-randomly generated test data, and methods to mutate pseudo-randomly generated test data according to specific attributes.

4. Testing with Parameterized Random Test Data

Pseudo-random test data can be used effectively in a variety of situations to increase test data coverage and also expose anomalies that might now otherwise be found with static test data. This section contains examples of issues found using word editor programs to evaluate the effectiveness of parameterized pseudo-random test data generation. This case study used Windows Notepad, and 2 shareware applications; Copywriter and Win32pad. The oracle was a simple string parsing algorithm that compared each byte in the output (the string in each program's rich edit control or the contents of a saved file) with the randomly generated test data applied as input.

4.1 Testing Object Linking and Embedding (OLE)

A common approach to moving data between applications is via OLE, copy and pasting data streams. I used a pseudo-random string generator to automatically generate a string of 1000 Unicode characters

from the Unicode base multilingual plane and set to the Windows clipboard. The string was then copied from the Windows clipboard and pasted it into the rich edit control of each program.

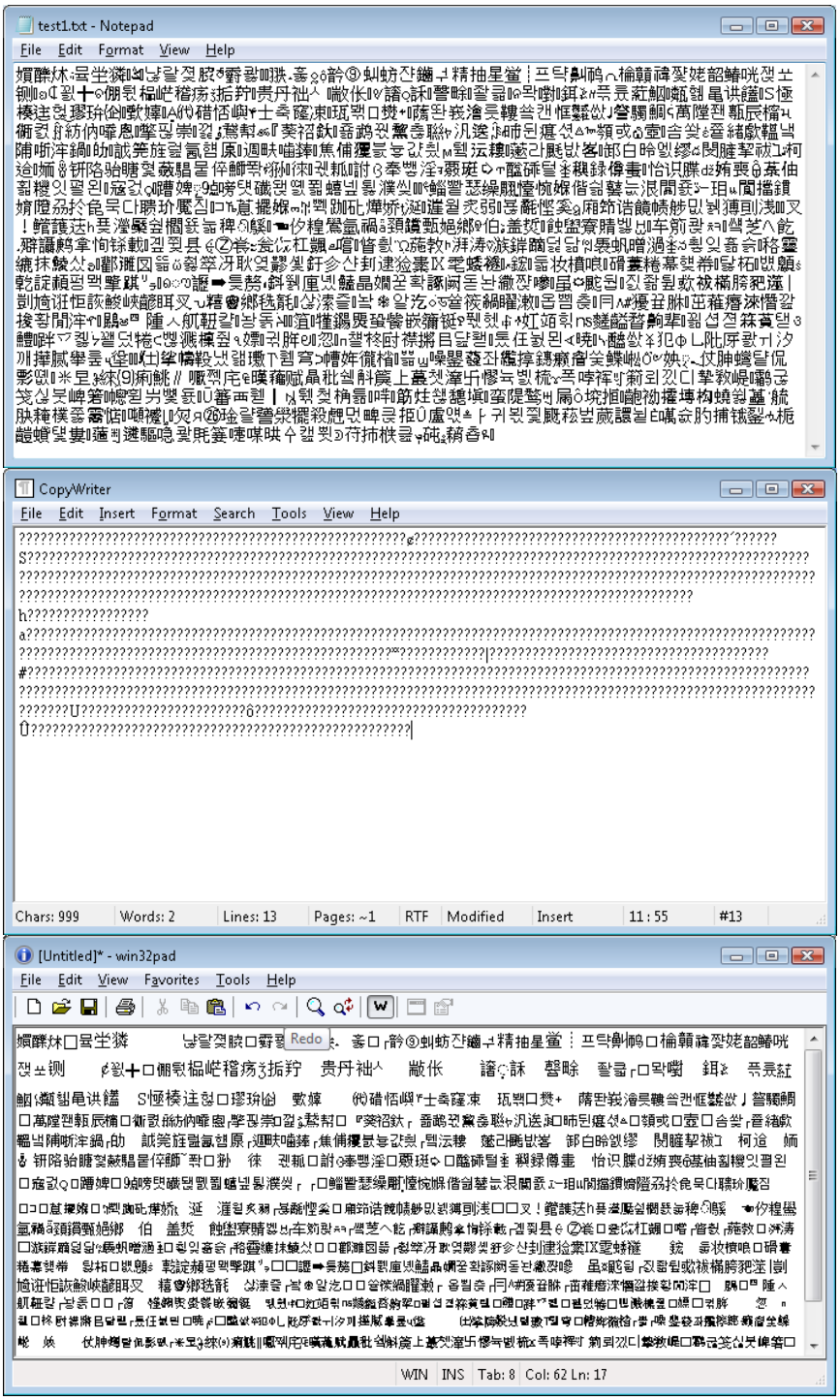

Figure.5: *Results from copy/paste of randomly generated Unicode test data*

As we can see in Figure 2 the randomly generated test data was properly copied from the clipboard into Notepad's rich edit control while the contents of Copywriter's rich edit control was unable to properly handle the incoming stream and converted all characters (except characters in the Unicode Latin 1 range)

to the question mark character (U+003F), which is the Windows default character for unknown character encoding. The Win32pad application visually appears to have preserved the string. Although several characters appear to be missing, a code point is there we simply cannot visually see a glyph. However, when the oracle performs a byte comparison of the string copied from Win32pad the oracle flags an error because all the character code points in the data stream are U+003F; the question mark character. So, even though the random test data might appear to be displayed properly, attempting to save this to a file would result in data loss because the Win32pad converted the stream to the question mark character.

4.2 Testing File I/O Operations

Another common testing scenario involves saving a file, and opening the file. This test saved randomly generated test data to a text file with the Notepad application using the Unicode encoding format. Then the saved file was reopened with the Notepad, Win32pad, and CopyWriter programs.

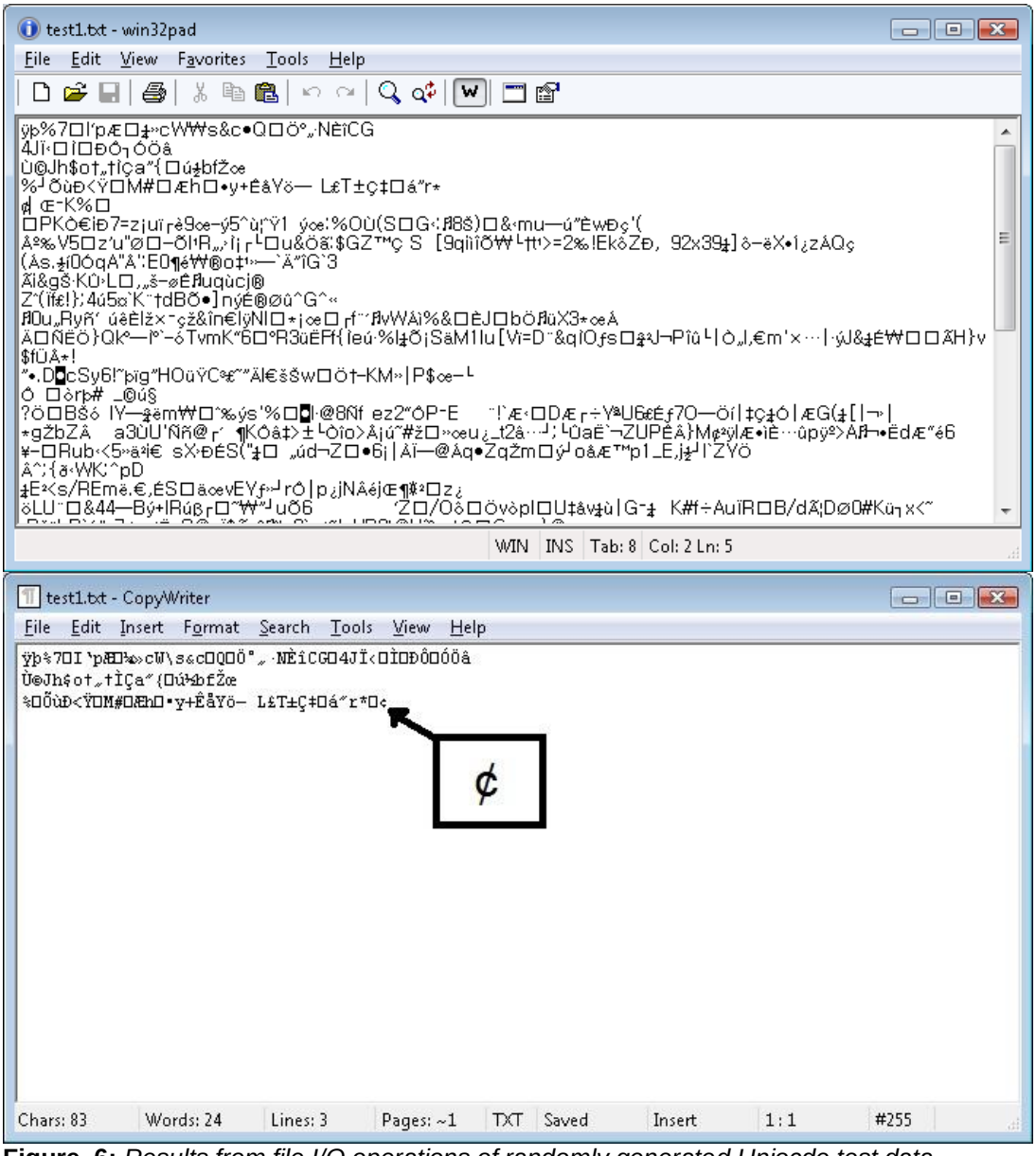

Figure. 6: *Results from file I/O operations of randomly generated Uniocde test data*

Reopening the file in Notepad produced no errors, but the oracle indicated 2 major errors in both the Win32Pad and CopyWriter programs. The most noticeable problem illustrated in Figure 6 is the obvious character corruption. This is caused by the stream reader algorithm for both applications reading the data stream byte by byte instead of by wide_char or as a Unicode encoded stream. However, also notice that the file contents in Copywriter only contains 83 characters instead of 1000 characters that are contained in the file. This occurs because the character immediately following the last character displayed is a Unicode value of U+8CAF. The 8C character code point by itself is the partial line backward control character. Once the stream reader in Copywriter hits a control character it stops reading the rest of the stream which causes the data to be truncated. Interestingly enough, after selecting all the text in the Win32pad application and compare it against the expected result the oracle also indicates truncation after the ¢ character in the stream although additional characters are displayed in the rich edit control.

4.3 Testing With Surrogate Pair Characters

Surrogate pair characters are a uniquely encoded Unicode character composed of a pair of 16-bit code points which make them especially problematic for a developer to parse. Currently, most defined surrogate pair characters are in the CJK language family and the Chinese National Standard GB18030-2000 requires implementation for an official certification of compliance from the People's Republic of China (PRC). Additional surrogate pair characters are defined in American Indian, African, Central Asian, and Ancient language groups as well.

Ad hoc testing of various applications has revealed a variety of problems with randomly generated Unicode strings that contain surrogate pair characters. The majority of errors are unhandled exceptions due to confusion between character counts and byte counts. Other errors include unexpected error dialogs, string or character corruption, and data loss. Random samplings of applications have revealed numerous errors suggesting this is a highly problematic functional problem that requires greater attention.

5. Conclusion

Random test data generation using parameterized data from equivalence partitions is well researched and has been successfully employed in several areas throughout the industry. Thenevod-Fosse, Waeselynck and Y Crouzet (Thenevod-Fosse 1991) presented an approach for random inputs based on specific criteria for improved structural testing. Murphy, Kaiser and Arias (Murphy 2007) discuss a similar approach except their method parameterizes random data according to equivalence partitions for system level black box testing approaches. Also, Demillo and Offutt (Demillo 1991) present a technique for generating test data based on constraints and mutation analysis for fault based testing criteria.

Parameterized random test data is an approach for generating unbiased samples of test data than can be applied to both positive and negative tests. This approach allows the tester to stipulate the specific properties or parameters to generate random test data elements from large populations of possible data, or mutate randomly generated test data in order to satisfy the specified requirements. The ability to generate large sets of both valid and invalid pseudo-random test data for use in testing helps reduce uncertainty by increasing the amount and variability of test data used during the testing cycle.

References

Beizer, Boris. *Software Testing Techniques*. New York, NY: Van Nostrand Reinhold, 2009

Wikipedia, *Luhn Algorithm*, http://en.wikipedia.org/wiki/Luhn_algorithm

Myers, Glenford, *The Art of Software Testing*. New York, NY: John Wiley & Sons, 1979

Knuth, Donald E. Art of Computer Programming, Volume 2: Seminumerical Algorithms, Reading, MA: Addison-Wesley, 1981

Attwood, Jeff. *Computers are Lousy Random Number Generators*, http://www.codinghorror.com/blog/2006/11/computers-are-lousy-random-number-generators.html

Kohl, Jonathan. *Reckless Test Automation*. http://www.kohl.ca/blog/archives/000160.html

Hamlet, Richard. *Encyclopedia of Software Engineering.* New York: Wiley, 1994, s.v. "Random Testing."

P. Thevonod-Fosse, H. Waeselynck and Y Crouzet. "An Experimental Study on Software Structural Testing: Deterministic Versus Random Input Generation," *Proceedings of the Twenty-First International Symposium on Fault-Tolerant Computing*, June 1991, pages 410-417

C. Murphy, G. Kaiser and Marta Arias, "Parameterizing Random Test Data According to Equivalence Classes", *Proceedings of the 2nd International Workshop on Random Testing*, 2007, pages 38 – 41

R. Demillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Transactions on Software Engineering vol 17, no. 9*, 1991, pages 900-910

B.A. Wickmann. *Some Remarks about Random Testing*. National Physical Laboratory, http://www.npl.co.uk/upload/pdf/random_testing.pdf, May 1998