Self Verifying Data – Testing without an Oracle

Noel Nyman (noeln@microsoft.com)
Software Test Engineer
Semantic Platforms Test, Microsoft Corporation

Abstract[1]

Most of today's software applications can work with large amounts of data. Whether you're testing a database, a word processor, an Internet browser or a 3D-action game, you need to do some of your testing with very large, rich data sets. Maintaining oracles for those data sets is time consuming and expensive. But without a good oracle, a traditional data set is worthless for testing, because your tests' accuracy can't be verified. In this paper, we describe "Self Verifying Data," a method that stores both the test data and the oracle in a single data set. Combining the data and oracle solves most of the expensive maintenance problems and makes verification simpler. This method easily scales to data sets of any size and it can be used with manual and automated testing. You can adapt Self Verifying Data principles to the data you use to test almost any kind of software application.

The Large Data Set Oracle Problem[2]

Suppose your team tests a database application designed to work with state income tax data. Your test data set, like the real world data, contains millions of taxpayer records. You've created automated tests to retrieve individual fields from those records. For example, one test picks a name at random from the data set…perhaps "Adam Aardvark." Then the test queries the database for Adam's adjusted gross income, and finds the value $32,327.

Is that really Adam's adjusted gross income? How do you know whether this test passed or failed? To answer that question, you need an "oracle"…an all-knowing system that can tell you if the result of any of your tests is correct. There are many kinds of oracles.[3] For simple tests on small data sets, you, the tester, can be the oracle. Perhaps *you* are Adam Aardvark and you've memorized your adjusted gross income. Or Adam may be a fictional person you've inserted in the data set for testing purposes and you remember the data you entered in his record. Either way, your memory is the oracle.

Human memory oracles won't work for large data sets because no one can remember the data for millions of records. When you start choosing records at random from a large data set you need an external oracle. The oracle could be a set of filing cabinets with all the paper tax returns for last year. After choosing another name at random for a test, "Xen Xenomorph" perhaps, you'd go to the cabinet with the "X's" and hunt for the Xenomorph return. Then you'd find Xen's adjusted gross income on the appropriate tax form, write it down, and re-file the form (don't lose it, you may pick Xen Xenomorph's name again for another test). Finally, you'd compare the figure from the oracle with the data your program returned. Repeat this a few thousand times for each field on each of the tax forms and you've done some testing.

If you're luckier, the tax return information is stored electronically and you don't have to hunt through all those paper files. You'll probably have two computers on your desk, one running your new program and the other accessing the oracle data. Automated testing is much more efficient than manual verification for a project like this. But you'll have to teach your test automation to read the electronic oracle.

What happens if the oracle and your program disagree? Is the oracle always correct? Perhaps the tester who worked on the old system wasn't as diligent as you and missed a bug or two. When the program you're

[1] This paper was first presented at STAREast '99.
[2] We're using "oracle" in the generic testing sense, not as a reference to Oracle™ Corporation or its products.
[3] For information on several types of oracles used in software testing, see "A Taxonomy for Test Oracles" by Doug Hoffman at http://www.stqe.com.

testing makes changes in the data, the oracle will still contain the old information. Can your automated tests make changes in the older format oracle files? If not, how will you update the oracle when the tax authority wants you to test changes in 100,000 returns?

The Rich Data Set Oracle Problem

There may be bugs in your program that the current test data doesn't expose. Inflation will increase tax payers' incomes over time. You'll want some test records that have much larger numbers than any that are in the data now. People moving to your state from other parts of the world have names with character combinations and lengths that are different from those in the current data. To cover all the possibilities, you want some records to contain random collections of all the accepted characters in many different string lengths.

How will you create an oracle for this new, rich test data and merge it with the current oracle? What happens if some accidental keystroke, or a bug in that old system, wipes out the oracle file? Is "Çr¥©øg9»½¿" a collection of random characters that were entered in the rich data set to represent a first name? Or is it a meaningless mess, garbled by a bug in the application you're testing? You can't tell good data from garbage by simple inspection when all the data are collections of random characters. That means you can't recreate an oracle for that data, either. If the oracle is lost or damaged, the expensive data set is worthless and you have to create a new one with a new oracle.

The Self Verifying Data Solution

Self Verifying Data (SVD) solves the large, rich data set oracle problems. SVD is data created specifically for testing. You can think of it as a "debug" version of test data, because SVD works like a debug build of the executable code for your application. Debug versions of executable code contain extra, embedded information to help developers step through the program and identify bugs in the source code. Debug code is larger and usually runs more slowly than the "release" version you'll ship to customers. You will probably do most of your testing with debug versions, then verify your tests on the release version after all the known bugs are fixed.

SVDs also have "extra" information. You use that information to verify that the data retrieved by your tests is correct. Like the debug version of your program, an SVD is larger than a similar "real" data set. But the testing time it saves over using traditional oracles more than makes up for that. You don't need a separate oracle file, so there's nothing to get lost or out of synchronization as you make changes to the data during tests. An SVD can also contain all the information your automated tests need to determine how to use the test data, such as the total number of records in the data set, the names of the fields, and how the records inter-relate. If you plan your automated tests well, you can use the same tests on many different SVDs of any size. You can verify that your tests work with a few dozen records, then switch to SVDs with millions of records without making any changes in your automation.

In this paper we use a hypothetical database program to demonstrate testing using SVD. But you're not limited to testing just databases. The SVD method is easy to adapt for many other kinds of applications. We'll show you how to use SVDs with spreadsheets, word processing documents, Web applications and even graphics programs at the end of this paper.

Five Data Properties You Usually Want to Verify

The tax authority has decided to make sure their new program will be able to handle "any" taxpayers' names in the future. Many modern programs use Unicode to address this situation. Unicode capable software can handle every human language with ease. Our tax authority isn't that progressive. They've limited their program to the printable ANSI (American National Standards Institute) character set.[4] That set

[4] See http://www.unicode.org for more information on Unicode and http://web.ansi.org/default_js.htm for details on ANSI standards.

includes everything from the space character (ANSI code 0032), through common punctuation, to digits and the English alphabet characters. There are some strange looking characters in the higher ANSI codes, things like the copyright symbol "©" and the fraction "½". The upper part of the set contains several foreign language characters and ends with "ÿ" (ANSI code 0255). To test your program thoroughly for all combinations of ANSI characters, you'll need many, many records with "names" like this:

```
First name: Ñr£¼üxµ§¿
```

When you retrieve data during a test, your oracle needs to verify these five things about each item for you:

1. **Is it Legal** – Did this data come from the tax record data set, or did your program open a bitmap graphic file by mistake? When a name can contain all the ANSI characters, it's hard to tell by inspection that it's not really "binary" data instead of a character string. If this *is* data from a tax database, is it the *correct* database, or an older version you're no longer using for tests?
2. **Is it Valid** - Is this data that should be here? When you start testing updates to the data records, you'll want to verify changes, deletes and rollbacks. Is this the new data or is it the old original data that wasn't changed because of a bug?
3. **Is it the Correct Type** – Is this string really from a first name field? Perhaps it came from a last name or an address field instead.
4. **Is it from the Correct Record** – Is this the first name from the record we asked for? If you use, for example, a taxpayer ID number to access records, you need an oracle system to match the ID with this strange first name.
5. **Is it Accurate** – Are there any missing, extra, incorrect or transposed characters? With random character data, it's impossible to tell by simple inspection.

Using SVD

SVD adds codes to your test data for each of the five verification levels:

```
First name: Ñr£¼ü<Legal><Valid><Type>
                <Record><Accurate><Delim>xµ§¿
```

To make the example easy to read, we've shown the "first name" on two lines. But in the SVD it's really a single string with the codes embedded in the first name data. It's best to put the SVD codes in the middle of the data, because some bugs are exposed only when unusual characters or combinations appear at the start or end of a field. If our SVD codes lead or trail the data, those bugs might be hidden.

An SVD for manual testing could use codes like these:

```
First name: Ñr£¼ü<Set 123><Original><FirstName>
                <55-12-1234><00619><Delim>xµ§¿
```

A human tester can verify these codes easily by inspection during manual tests. For this example, the test looked in data set #123 for the first name of the taxpayer with ID #55-12-1234. We haven't run any change/insert tests yet, so we should find the original data. Using the embedded SVD codes <Set 123>, <FirstName>, <55-12-1234> and <Original>, we can verify the first four levels for the test result.

Several common algorithms can check the accuracy of data. The Checksum method is easy to calculate, but it won't catch errors that transpose characters, and may miss other bugs as well. The slightly more complex Cyclical Redundancy Check (CRC) catches more bugs. Using the algorithm you choose, the system creating the SVD calculates the accuracy value for the first name string and embeds it in the data. The human tester can use the same algorithm on the retrieved data string and compare the result to the SVD accuracy code, <00619>. The delimiter code <Delim> helps the tester separate the data from the SVD codes. It may not be needed for some SVD formats, as we'll see later.

Unique SVD Fields

An important side effect of adding SVD codes to the data is that the first name field is now guaranteed to be unique in the data set. It's likely that many people have the first name "Betty" in a large taxpayer data set. But when we add the SVD codes to a particular Betty's first name, we create this:

```
First name: Bet<Set 123><Original><FirstName>
                 <57-89-7654><08492><Delim>ty
```

By definition that new first name value is unique because it contains the code for a unique record ID in our data set. That's no problem when we want to verify that we found the first name for taxpayer #57-89-7654. But we can no longer search for all taxpayers with the first name "Betty." The tax authority probably will never want to do that, either. So, we don't need run that test with our SVD data set. But searching for everyone with the *last* name "Rubble," or everyone living at a particular address, is just the sort of thing a tax agency might do. If the agency will do it, *we* need to test it. That means we can't make last names or addresses unique in our test data. We need a different method to use SVD with those data fields.

Non-Unique SVD Fields

There are at least three kinds of data fields in typical records that we shouldn't add SVD codes to:

- **Shared value fields** – Fields that share their values across several records. Typical examples are last names, job titles and company affiliations.
- **Non-string data** – Usually SVD codes are string or character data type. We can't add them to numeric or date fields.
- **Limited length** – Fields such as first name, that would otherwise be good candidates for SVD codes, may be too small to accept them.

To use SVD codes with any of those field types, we need to add a new field to the data set for each record:[5]

```
  First name: Bet<Set 123><Original><FirstName>
                   <57-89-7654><08492><Delim>ty
   Last name: Rubble
Gross income: 67421.89
    OurCodes: <OurCodeField><Set 123>
                   <57-89-7654><Delim>
                   <LastName>Rubble<Delim>
                   <GrossIncome>67421.89<Delim>
```

When a test uses one of the fields that doesn't contain SVD codes, it also retrieves the **OurCodes** field from that record. Like other unique fields, **OurCodes** contains an SVD code that identifies the "**OurCodes** type," so we know that's the field we're looking at (human testers may not need this, but automated tests will). It also contains the legal data set code and the record ID for verification. For each field in the record that has no SVD codes, there's a code in **OurCodes** that identifies the field type. It's followed by the value for that field. Delimiters separate the sets of codes and values to help humans (and later test automation) read them. This example doesn't use codes for valid data and accuracy. We'll detect most validity and accuracy errors when we compare the value in the source field with its counterpart in **OurCodes**. But you can add validity and accuracy codes if you want to be sure you don't miss *any* bugs.

Choosing the Code Characters

Our simple examples use long, descriptive SVD codes. In real SVDs they would take up too much space. Because the codes are part of the data your program will work with, you need codes that won't get in the

[5] If you can't easily add a new field for SVD codes, you may be able to put them in a field normally used for other data, such as a description field.

way of normal operations and cause their own bugs. You also need codes the automation (and perhaps humans) can easily distinguish from the real data.

The best SVD codes are part of an "equivalence class," a set of characters, any one of which should work just like all the others. The letters in the English language alphabet are an equivalence class for most testing. Any well written string handling routine should treat the letters "b," "m," and "w" in exactly the same way. Normally we think of equivalence classes only when we choose test data to help us find bugs. If we run a test using the letter "b" and it passes, we probably won't find a bug using "m" or "w" either and we can move on to another test.

When we choose SVD codes, we look at the likelihood of our codes *causing* bugs instead. If we already have strings with "b" and "w" in our data, adding an SVD code with an "m" isn't likely to cause bugs, because "m" is in the equivalence class with our data. It's best to avoid the boundaries of an equivalence class, so we wouldn't use "a" or "z" for SVD codes.

If your program works with Unicode strings, you don't have a problem choosing SVD codes. Unicode reserves a range of codes as "user space" for testing purposes. User space has 6400 characters we can use as SVD codes. By definition, a program must treat all Unicode characters the same way, regardless of what language they represent. So, *all* Unicode characters, even if they're SVD codes in user space, are part of the same equivalence class. If Unicode SVD codes cause bugs, they're bugs that we need to find and get fixed.

Unfortunately, Unicode applications still aren't commonplace. Our example tax application is limited to the much smaller ANSI character set:

```
0000 - 0032: control codes - tab, cr, lf
0032 - 0047:  !"#$%&'()*+,-./
0048 - 0057: 0123456789
0058 - 0064: :;<=>?@
0065 - 0090: ABCDEFGHIJKLMNOPQRSTUVWXYZ
0091 - 0096: [\]^_`
0097 - 0122: abcdefghijklmnopqrstuvwxyz
0123 - 0127: {|}~•
0128 - 0159:  •‚ƒ„…†‡ˆ‰Š‹Œ   •‘’“”•–—˜™š›œ  Ÿ
0160 - 0191:  ¡¢£¤¥¦§¨©ª«-®¯°±²³´µ·¸¹º»¼½¾¿
0192 - 0214: ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ
       0215: ×
0216 - 0223: ØÙÚÛÜÝÞß
0224 - 0246: àáâãäåæçèéêëìíîïðñòóôõö
       0247: ÷
0248 - 0255: øùúûüýþÿ
```

The full ANSI character set has 255 characters.[6] There are several groups that might be equivalence classes. The digits 0-9 look promising, but we'd have to use some complex combinations to tell the difference between SVD codes and real data values. Punctuation characters are almost always a bad choice. Many legacy systems reserve characters like "*" (asterisk) and ":" (colon) for special purposes. The "foreign language" alphabet characters starting at code 0192 might work. But they've only recently come into common use as language characters. We should keep them *all* to use as test data, in case they expose bugs in legacy routines. Someone may have used them in old programs as "flags," like the magic date "9-9-99" that caused some people Y2K grief.

For ANSI data systems, your best choices for SVD codes are still the characters from the English alphabet. The sets "B-Y" and "b-y" form the most useful true equivalence classes. As we'll see in the next section, you can create quite a few SVD codes by borrowing just a few alphabet characters.

[6] If your program accepts only characters from the older seven bit ASCII (American Standard Code for Information Interchange) set, just ignore the codes above 0127.

A Real SVD Using English Alpha Codes

Now that we've looked at the principles of SVD, we'll examine a real example. This application is also a program that accesses a database and uses the ANSI character set. To keep SVD code parsing simple, we decided to create our data without using any of the SVD code characters. So, if we decided to use "m" as an SVD code character, none of our data could contain the letter "m." Mickey Mouse could not have a record in our data. Leaving Mickey out might not be a serious limitation. But our application doesn't support a "date" data type, yet. All its day-of-the-week and day-month-year data are stored as strings. So, we had to keep all the letters in the days-of-the-week and names of the months for use as data characters. That left us these letters for SVD codes (we didn't include "z" because it's on the class boundary):

```
KQX  kqx
```

Having only six codes looks limiting at first. But, if we use codes that are five characters long and make the codes case sensitive, we can get 7,776 combinations from "KQX" and "kqx." To make parsing easier, we reserve the code "xxxxx" as the SVD delimiter and we don't allow any of the other SVD codes to have an "x" as the first or last character.

Here's how Betty Rubble's record would look with our real SVD codes:

```
  firstname: BetKKKKKkkkkq057089765408492ty
   lastname: Rubble
grossincome: 67421.89
   OurCodes: qqqqqKKKKK0570897654xxxxx
                  kqkkkRubblexxxxxqqkkq67421.89xxxxx
```

We've used only upper case characters for both the legal and valid codes, and combined them into one five character code…"KKK" for the first data set and "KK" for original data. The second data set will be "KKQ" and updated data would have a "KQ" validity code. So, if we updated Betty's first name in the second data set, the new code would be "KKQKQ." We use lower case letters for the type codes, and "kkkkq" is the code for "**firstname**."

The record ID number and the CRC accuracy code look squashed together in Betty's **firstname** data. It may be tricky for humans to visually separate the two, but we made it easy for the automated tests by padding both values where necessary so they're always the same size. This is string data, and common string or array routines can extract fixed lengths quickly. If we'll use the same data for manual tests, we could place the ID number between the validity and type codes to make parsing by human eyes easier. In the **OurCodes** field, "qqqqq" identifies the field itself, "kqkkk" represents the "last name" type and "qqkkq" is the code for "gross income."

We've used the "xxxxx" delimiter code to separate the information in the **OurCodes** field to make automated parsing easier. We don't need it in "unique" fields like **firstname**, because the final SVD code in unique fields is the accuracy code and it's always padded to a fixed length. Test automation can parse unique fields easily without the "xxxxx" delimiter. We add the delimiter to unique fields in SVDs that we use for manual testing, because that makes parsing easier for the humans.

Getting SVD Code Lists and Data Set Statistics from the SVD

The real power of SVD is its scalability. We may add more records, or more fields, or both to future SVDs and we'd like the test automation to adapt to those changes automatically. A separate list of which SVD code represents each field type defeats the SVD method because the list is an *oracle*. Even though it's a "test" oracle, it's still subject to all the oracle maintenance problems, and we need to get rid of it!

We do that by reserving a special record in the data set to hold all the statistics. Our SVD uses record #0 for that purpose, with the provision to point to other records from record #0 if we need them later. We fill each string field in record #0 with as much SVD information as it will hold. Here are the first few characters in some of the fields:

```
   employee: 0000000
 emailalias: Xcreation_datexxxxx11/24/98xxxxxhasa
  firstname: Xreportstoxxxxxkkqkkxxxxxworksinxxxx
   lastname: XLENGTH_OF_EMPLOYEE_NUMBERxxxxx7xxxx
      phone: Xfirstnamexxxxxkkkqxxxxxlastnamexxx
```

We use the character “X” at the start of each string to make sure we’re looking at SVD stats. Letter “X” is one of our reserved SVD characters and it won’t be used in any of the data set values. The “valid stats” character “X” and the delimiter code “xxxxx” are the only two things we need to define before the test automation reads this record. Those two codes are the same for all SVDs we create.

The automation fetches the data set “creation date” from the **emailalias** field and prints it in our test logs for reference. The data in **firstname** tells the test automation that “kkqkk” is used in this SVD for the “reportsto” field type. Other types are defined in the **phone** field. The **lastname** field tells the automation that every employee (record) number will be padded to seven characters. This SVD represents fictional companies with several levels of management and the tests are written to take advantage of that. The number of companies, workgroups, and managers in the companies change as the number of employees in the SVD increases. The test automation reads all the information it needs to adapt to those changes from the employee #0 record.

A Typical SVD Query

Here’s a typical automated test we run on our application:

```
Pick a random employee who has a manager. Then return the
phone number of that employee’s manager’s assistant.
```

Our demonstration SVD uses a set of “real” names instead of random ANSI characters and the phone numbers are formatted to look like real phone numbers. We ran the test and it returned this data:

```
(878) 312-3269
```

That result looks like a phone number (we don’t try to use real area codes). At least it doesn’t look like a name or an address. But there are over one million phone numbers in this SVD. Is this the right one? Are the digits in the right order? Did the test really pass? We use SVD codes to find out.

First, the automation needs to know which employees have managers. We access records in our SVD by employee number and the structure data in record #0 told our automation which range of employees report to managers. So, the automation picked a random number between “first employee who has a manager” and “last employee who has a manager.” For this test, the automation chose employee #478. Then the automation asked the program we’re testing to retrieve employee #478’s manager’s assistant’s phone number. That’s a complex query, but our application under test can handle it. To make sure the program returns the correct phone number, the automation also asks our application to return additional information for the random employee, the manager and the assistant.

The automation needs the **reportsto** value for random employee #478 because it contains the employee number for #478’s manager. That’s a shared value field, since other people may also report to #478’s manager. There aren’t any SVD codes in shared fields, so the automation also needs the **ourcodes** field.

Just for good measure, the test asks for the **firstname** and **lastname** values, and prints everything to the test log:

```
firstname: ZanKKKKKkkkkq00004780009184nan
 lastname: ElaXKKKKKkqkkk00004780056195ce
reportsto: 0000023
 ourcodes: qqqqqKKKKK0000478xxxxxkkqkk0000023xxxxx
```

The real **ourcodes** field is much longer than this. It's abbreviated here for clarity. The employee number is seven characters long, padded with zeroes to "0000478." The CRC accuracy codes are also padded to seven characters. The automation "knows" that we should be looking at data set "KKK" because that was an input parameter we used for this test run. It will also accept validity codes "KK," "KQ," or "QQ" for this test. Anything else will cause the test to fail.

Our automation verifies the legal, valid, type, record and accuracy codes for the **firstname** and **lastname** field values. It also verifies the legal, valid, **ourcodes** type and record number for the **ourcodes** field. Any discrepancies there cause the test to fail.

Next, the automation looks for the **reportsto** code in the **ourcodes** field. If it doesn't find one, employee #478 has no manager, according to **ourcodes**. Since the automation specifically queried for an employee *with* a manager, the automation would report a failure in that case. But this **ourcodes** field *does* have the "kkqkk" code and the automation "knows" that's the code for **reportsto**, because it's defined that way in our special employee #0 record. The automation grabs the value between the "kkqkk" code and the "xxxxx" delimiter. That's "0000023" for employee #478.

Then, the automation looks for the **reportsto** field value for employee #478. If there's no **reportsto** field or its value is empty, the test fails. In this test, the automation found a value for **reportsto** and compared it with the employee number it found in #478's **ourcodes** field. The values are the same. So, we can be fairly confident that employee #478's manager is employee #23. This part of the test passes and we got all of our pass/fail information directly from the SVD…no external oracle was required.

Next, the automation looks at the fields our application returned for the manager, employee #23:

```
   firstname: AnKKKKKkkkkq00000230000623ji
    lastname: CubeKKKKKkqkkk00000230034226rto
hasassistant: 0000272
   reportsto: 0012345
    ourcodes: qqqqqKKKKK0000023xxxxxkkkxq0000272xxxxx
                   kkqkk0012345xxxxx
```

The automation checks the manager's **firstname**, **lastname** and **ourcodes** fields. If there are any legal, valid, type, record or accuracy errors, the test fails. Everything is OK so far. Manager #23 also reports to someone, manager #12345. But our test isn't interested in that and ignores it. Instead, it looks for the **hasassistant** code in the **ourcodes** field. Notice that the "kkkxq" code for **hasassistant** is in the position occupied by the **reportsto** code in #478's **ourcodes** field. Position isn't important and our tests look for field codes anywhere in the **ourcodes** field. There's no way to know in advance what codes may be in any employee's **ourcodes** field. Manager #23's assistant is employee #272, and the automation compares that value with what it finds in the **hasassistant** field. Everything matches and the test is still passing.

Next, the automation looks at the results for the assistant, employee #272:

```
firstname: CheyKKKQQkkkkq00002720030024ane
 lastname: LoKKKQQkqkkk00002720006613el
    phone: (878) 31KKKQQkqkkq000027200561672-3269
reportsto: 0000023
 ourcodes: qqqqqKKKQQ0000272xxxxxkkqkk0000023xxxxx
```

The automation tests all the usual stuff for **firstname**, **lastname** and **ourcodes**. Everything passes, but note the "QQ" part of the legal/valid code indicating that this record has been updated since the SVD was created. Then, the automation looks at the **phone** field value. Phone numbers are unique in our SVDs and they contain SVD codes. Humans may have trouble parsing this data, but it's a snap for our automation. The SVD parts are:

```
KKKQQ   kqkkq   0000272   0056167
```

"KKKQQ" verifies this is updated info from our target data set. The "kqkkq" code means it's a **phone** field and the "0000272" matches the record number we expected. Concatenating the remaining pieces, we get "(878) 312-3269" for the phone number value. When the automation runs its CRC algorithm on that string, the result is "56167." That matches the CRC stored in the **phone** field. Everything passes and all our verification was done using the SVD codes in the data set…without an external oracle.

Another Typical SVD Query

Our next test is more complex:

```
Pick a random employee and a random meeting subject. Give
us the e-mail aliases of everyone attending meetings about
the random subject that were organized by the random
employee.
```

We get these results:

```
JameKKKKKkkkqk00000320019624Rud
PilKKKKKkkkqk00001080051605App
CarltKKKKKkkkqk00003210006486Man
EdwKKKKKkkkqk00004610061632Vargh
```

The SVD codes tell us that these are original e-mail aliases from our data set and they are the accurate values. But they don't tell us if they are the *right* e-mail aliases…yet.

To get that information, our automation records the random meeting subject, "Sinking Fund," and the random employee number, #32. Next the automation gets information on all the "Sinking Fund" meeting(s) organized by employee #32. In this data set there's only one matching meeting:

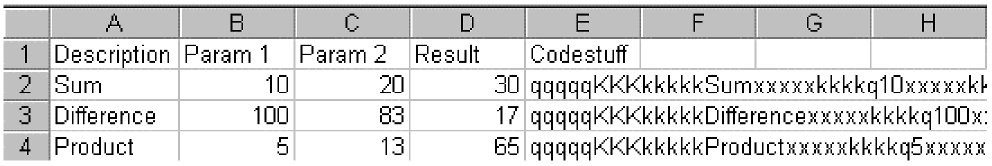
```
           event: E0054321
eventdescription: Sinking Fund appropriation meeting
     organizedby: 0000032
invitationsentto: 0000108
invitationsentto: 0000321
invitationsentto: 0000461
        ourcodes: qqqqqKKKKKE0054321xxxxxxx
                        kkqqqSinking Fund appropriation
                          meetingxxxxx
                    kqkxq0000108xxxxx
                          0000321xxxxx
                          0000461xxxxx
                        kkxqq0000032xxxxx
```

The test automation uses the same techniques we described in the previous section to verify the description, the event organizer and the people invited to the event. Shared data fields **organizedby** and **invitationsentto** contain no SVD codes. The automation verifies their values by comparing them with the values in **ourcodes** associated with the “kqkxq” and “kkxqq” codes. The data used by the application we’re testing isn’t stored in typical column/row fashion. So if more than one person is invited to the event, each one has their own **invitationsentto** field in the event data. Rather than take up space in **ourcodes** with duplicate SVD codes for each invitee, we let them share the same code as shown here. The automation doesn’t know how many invitees there are. Instead it keeps reading employee numbers until it sees either the end of the field or a new SVD code following a delimiter.

Now the automation compares the list of invitees it found in the event record with the list it created from the returned e-mail aliases. The automation knows that the e-mail alias of the organizer, employee #32, should have been returned by the application…and it was. Three other employee’s e-mail aliases were also returned: #108, #321, and #461. The automation checks to see that each of these employees was invited to the meeting. They all were. Then automation looks for two other possible errors. Were e-mail aliases returned for any employees *not* invited to the meeting? Were e-mail aliases missing for any employees who *were* invited to the meeting? An affirmative to either means the test failed. Otherwise the test passes.

Using SVD to Test Applications that aren’t Database Programs

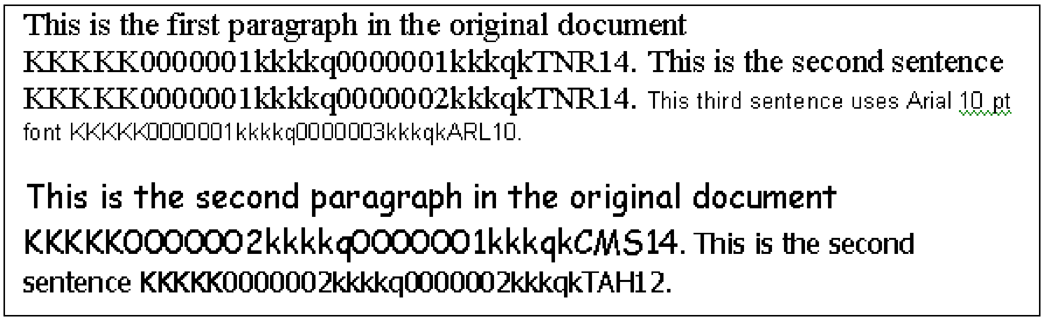
Once you understand how to use SVD codes with a database, you can add SVD codes to the data for most any kind of application. Spreadsheets are very much like traditional databases. You can think of each row as a “record” and each column as a “field.” In this example we’ve added a new **Codestuff** column to the spreadsheet, because the cells hold either functions or numeric data, and they won’t accept SVD codes.



	A	B	C	D	E	F	G	H
1	Description	Param 1	Param 2	Result	Codestuff			
2	Sum	10	20	30	qqqqqKKkkkkkSumxxxxxkkkkq10xxxxxk			
3	Difference	100	83	17	qqqqqKKkkkkkDifferencexxxxxkkkkq100x			
4	Product	5	13	65	qqqqqKKkkkkkProductxxxxxkkkkq5xxxxx			

The **Codestuff** cell in each row stores the name of the function being tested and the original parameters stored on the sheet. We could add codes for expected results or any other information we’d like to verify during tests.

There are several ways to adapt the record/field structure to a word processing document. For this example, we’ll make each paragraph a “record” and each sentence a “field.”

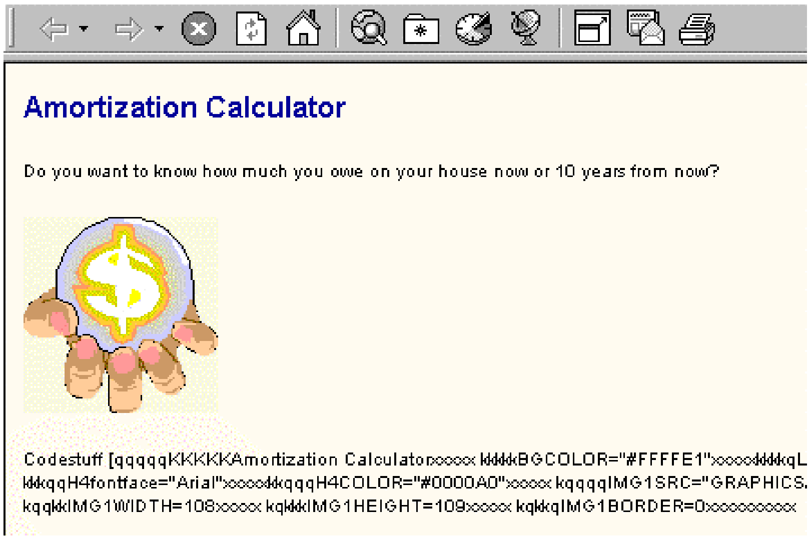


This is the first paragraph in the original document KKKKK0000001kkkkq0000001kkkqkTNR14. This is the second sentence KKKKK0000001kkkkq0000002kkkqkTNR14. This third sentence uses Arial 10 pt font KKKKK0000001kkkkq0000003kkkqkARL10.

This is the second paragraph in the original document KKKKK0000002kkkkq0000001kkkqkCMS14. This is the second sentence KKKKK0000002kkkkq0000002kkkqkTAH12.

We’ve added SVD codes to the “data values” of each sentence. In this example, the SVD codes identify the original paragraph for the sentence, its order in that paragraph and the size and name of the font used. We could use those codes to verify “find, using font type and size” tests, or the results from copy/cut/paste tests. We’d store different kinds of SVD codes in documents used for other kinds of tests.

We can also use SVD codes to help test Web pages.



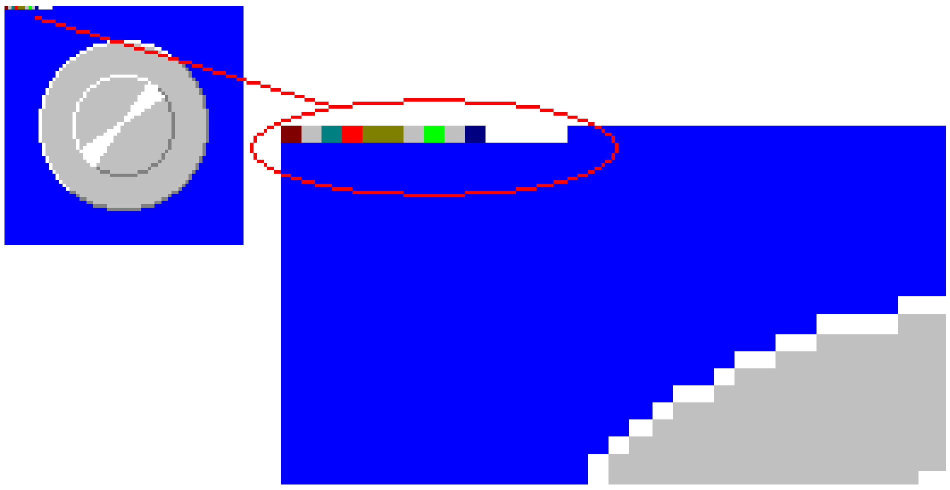
Amortization Calculator

Do you want to know how much you owe on your house now or 10 years from now?

```
Codestuff [qqqqqKKKKKAmortization Calculatorxxxxx kkkkkBGCOLOR="#FFFFE1"xxxxxkkkkqL
kkkqqH4fontface="Arial"xxxxxkkqqqH4COLOR="#0000A0"xxxxx kqqqqIMG1SRC="GRAPHICS.
kqqkkIMG1WIDTH=108xxxxx kqkkkIMG1HEIGHT=109xxxxx kqkkqIMG1BORDER=0xxxxxxxxxx
```

In this example, we've used an HTML paragraph with SVD codes to identify the page title, background color, and the name, size and placement location of the graphic on the page. Using test automation, we can locate the SVD paragraph on the page when it's displayed in a browser, then verify the page information by parsing the SVD codes and data. You could also store the SVD codes in HTML comments so they don't clutter the display, but would still be visible to test automation that can "view source."

In this last example, we've added SVD codes to a graphic element. This graphic might be "data" used to test a picture editor, a CAD program or a computer game.



We can't usually use string characters as SVD codes in graphics. Instead, we create binary SVD codes. Depending on the application we're testing, they might be single bytes, DWORDs or individual pixels with the SVD codes expressed as RGB values. We might even be able to hide our SVD codes in the description/comment area some graphics formats support.

In this graphic we've added codes that give us the name of the image and tell us something about how we're going to use it in the application. The codes are followed by a series of "white" pixels that act as a delimiter between the SVD codes and the rest of the graphic. As we test the application, we can read the SVD codes in each graphic to make sure the right things are happening.

SVD – It's Not the *Only* Way to Test

SVD works best when you need to automate tests with large amounts of rich data. It can save you a lot of time, find interesting bugs and help you release a high quality application. You can create *really* cool tests using SVD!

But don't make the mistake of using SVD to do *all* your testing or waste your time testing applications that can't use SVD effectively:

- SVD adds "extra" information to your data sets. Some data set structures don't adapt well to adding large amounts of extra information. Even if you can squeeze some SVD codes into the data, you may not find enough bugs to make the effort worthwhile.
- SVD helps you find bugs in how your application handles data. While that's very important for some applications, it's only a small part of what other programs do. As you've seen in our examples, it takes time and planning to create good automated tests to verify all the SVD data. If data handling is only a small part of your application's functionality, using SVD may be too expensive to justify the few bugs it's likely to expose.
- SVD verifies the values of the data, but it tells you nothing about what that data looks like on the screen. For some applications, *how* things look is just as important as the application's data processing accuracy. SVD may not help you find "look and feel" bugs.
- For the final release version of any application, do some testing with realistic data that doesn't contain SVD codes.

Noel Nyman

Noel Nyman has worked in software product development and testing for over 20 years on an eclectic project mix including embedded controllers, shrink wrap applications and operating systems. As the test lead for the Microsoft Windows NT 4.0 32-bit Applications Test team, he pioneered the use of “dumb” monkey automated test tools to increase operating system reliability. Prior to the release of Windows 2000, Noel wrote the desktop applications test plan for the Certified for Windows 2000 program. The 400-page test plan has been called “the most rigorous testing outside the military.” Noel and James Bach co-created the Windows Exploratory Testing method used by the Microsoft Windows group manual testers to create test plans for their applications.

Noel developed the Self Verifying Data method for use in automated search engine verification by the Semantic Platforms group at Microsoft. He’s currently responsible for automation, bug triage, and component/tools testing in the Windows Application Compatibility group.

Noel wrote the “Black Box Monkey Testing” chapter in the *Visual Test 6 Bible* that demonstrates how to use sophisticated Windows API testing with Rational Visual Test. He’s a regular participant in the Los Altos Workshop on Software Testing and the Austin Workshop on Test Automation, and an occasional contributor to *STQE* magazine.