Reengineering for Testability

Workshop on Software Reengineering (WSR)
Bad Honnef, May, 2006
by

Harry M. Sneed
ANECON GmbH, Wien
Universities of Regensburg and Passau

1 Rationale for Reengineering

There are many reasons to want to reengineer an existing software system.

- to increase the maintainability [1],
- to improve the performance [2],
- to increase the interoperability [3],
- to decrease the personal dependency.

These goals have been covered in the pertinent literature. [4] This paper is devoted to promoting yet another goal for reengineering, namely

- to improve the testability.

These diverse goals may in some ways be related, but in some instances they are contrary to one another. For instance improving performance is often done at the cost of decreasing maintainability. The planer of a reengineering project must be aware of what the exact project goals are, since they will differ from project to project. [5]

2 Defining Testability

Test costs are driven by the size and complexity of the software. Size can be measured in terms of the number of elements making up the system. These could be statements, methods, classes, components, interfaces, files, database tables and GUIs. Complexity is measured in terms of the number of interactions between the elements. These could be associations, calls, messages, file transfers, database accesses, import, exports and events from outside. The less there are, the less there is to test.

Another factor which influences testability is the visibility of the data interfaces. Data passed between components can be encoded in internal data formats or it can be passed as readable character strings. An example of the former is a CORBA API. An example of the latter is an XML document. The easier it is to read and interpret the data, the easier it is to generate and validate that data.

A very critical factor in testability is the separation of the user interface from the processing logic. It should be possible to create input data streams without using the user interface. The same applies to the data output. It should be possible to intercept and store the outputs without having them displayed in the user interface. This separation of presentation from processing is a prerequisite to testing the processing, i.e. the business logic, without having to enter the data in the user interface, which requires a lot of time and is difficult to automate.

A final factor in reducing test effort is the separation of the data access operations from the data processing. By having a separate access shell, it is possible to test the data storage and retrieval without having to go through the business logic. On the other hand it is possible to test the business logic with simulated data accesses without having to have all of the databases filled with suitable test data. [6]

The goal of reengineering for testability is to restructure the software in such a way that testability criteria can be met while at the same time reducing the size and the complexity of the system.

3 Achieving Testability

Testability of software implies that the software can be tested with a minimum of effort. This in turn implies that a minimum of test cases are required to test all features. To reduce the number of test cases at the component level, one has to

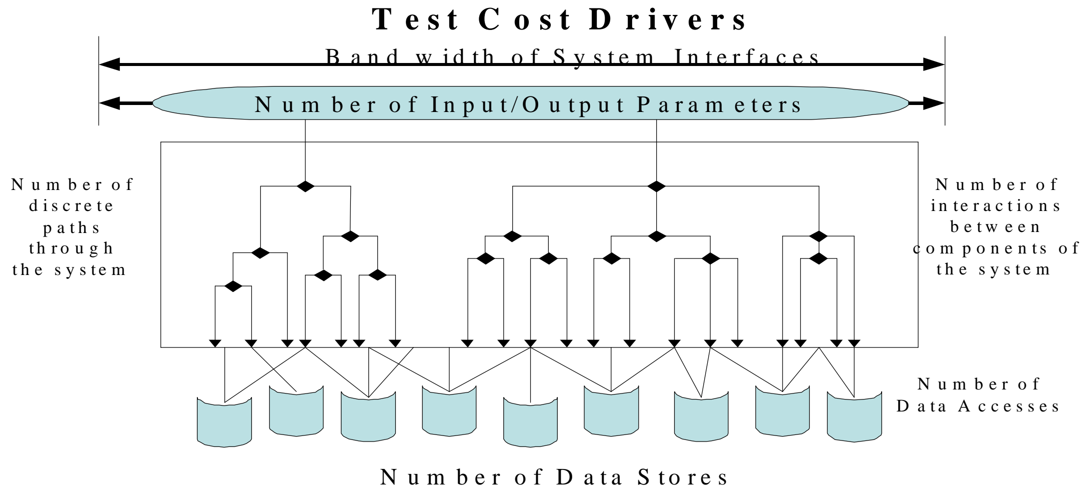
a) reduce the number of paths through the software unit, since there should be a test case for every path
b) reduce the number of entries, since there should be a test case for every entry
c) reduce the number of parameters, since there should be a test case for every combination of parameters

To reduce the number of test cases at the system level, one has to

a) reduce the number of interactions between the components, since every interaction has to be tested
b) reduce the number of database accesses, since every database access should be tested
c) reduce the number of database tables, because tables have to be filled with test data

d) reduce the number of system imports and exports, as these have to be generated or validated
e) simplify the user interfaces, since every control button, every menu point and every combination of parameters should be tested.

(see Figure 1: Test Cost Drivers)



4 Measures for Improving Testability

There are a number of reengineering steps which can be taken to improve the testability of an existing software system.

4.1 Restructuring to eliminate unnecessary paths

The number of paths through a software unit depends on the number of conditions in the code and how deep they are nested. By reformulating conditions it is possible to eliminate paths. For instance, nested ifs can be converted to a single compound if. Nested loops can be factored out into separate methods or procedures. The idea here is to flatten the control structure. [7]

4.2 Refactoring to reduce complexity

Deeply nested code can be factored out into separate methods or procedures. This will not decrease the number of paths but it is easier to test smaller units than it is to test large, complex ones. So refactoring has a positive effect on testability. Besides it can be easily automated. [8]

4.3 Removing clones

Clones are variants of a particular coding pattern scattered through out a system. The coding pattern could have been made into a super class or implemented as a common subroutine. For whatever reasons it was not. It was copied from component to component and modified there to fit some local requirement. Detecting such clones has been well researched in the software reengineering community. There are many tools available to support this. By removing clones, the number of test paths and with that, the number of test cases can be reduced significantly. The less code there is, the less there is to test. [9]

4.4 Removing redundant parameters

Since the number of data driven test cases is dependent on the number of parameters, it makes sense to remove those which are not absolutely necessary. Often, too much control data is passed to a component, data which could be derived from the context of the operation being invoked. Such parameters could be left out, thus reducing the number of possible data combinations.

4.5 Grouping database accesses into an access shell

Very often database accesses are a chain of development operations. First one access is made to determine what access to make next. By reformulating the query it may be possible to eliminate subsequent accesses, e.g. by extending the WHERE clause to include OR and AND conditions. Rather than having the database accesses scattered throughout the code, it is also better to pull them together into a single access component for each database table. In this way different procedures can use the same accesses. [10]

4.6 Merging database tables

It may come as a shock to database modelers, who are keen in factoring data down to the 5th normal form, but increasing the number of tables has a negative effect on testability. For every additional table, there has to be a test procedure to generate and validate it. So having too many tables means having too many test procedures. From the viewpoint of testing, it would be better to have tables with only a few attributes merged into the next higher level tables, e.g. to merge the address data back into the personnel data.

4.7 Eliminating unnecessary import/export interfaces

It may help to increase the function point count by having lots of different input/output data streams but it also increases the test effort. To keep the test effort down data changes with the environment should be kept to a minimum and, where possible, merged together into one interface. The less interfaces a system has, the less have to be tested.

4.8 Simplifying user interfaces

The greatest source of test effort in systems testing is the complexity of the user interfaces. The more "comfort" the user is offered in form of widgets, bells and whistles the more there is to test. Every additional window or interface object increases the number of test cases. One of the goals of interface reengineering should be to simplify the interfaces by removing all features which are not absolutely necessary. Why should it be possible to enter data via the keyboard or by selecting from a menu? That only doubles the number of test cases. One should decide for one or the other. Alternate means of submitting or displaying data may be good in the sense of usability, but it is detrimental to testability. Here the system architect must decide what is more important. [11]

4.9 Revising the Algorithms

For every problem there is a large number of possible solutions, some good, some bad, some simple and some overly complicated. Often developers under time pressure select the first best solution that occurs to them. It is seldom the simplest one. The result is a complex algorithm which requires a lot of test cases to test, more test cases than would be necessary for a sample solution. When this happens, testability becomes low. To really raise testability, the human reengineer should reformulate the problem and select a simpler solution. This is often cheaper than trying to test an over complicated algorithm.

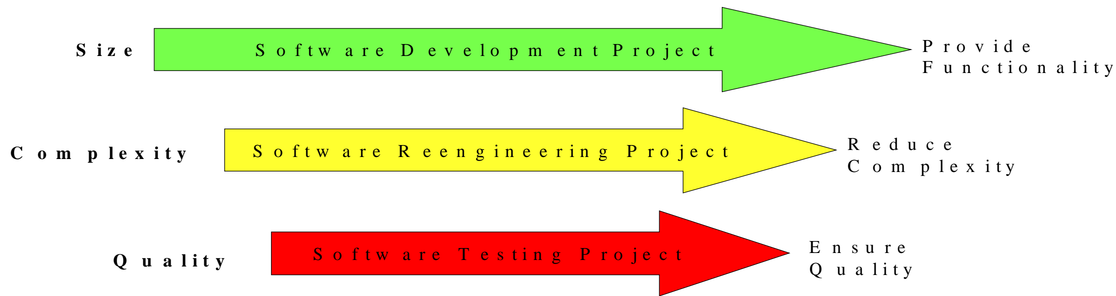
4.10 Restructuring the Architecture

The architecture of a software system has a profound effect upon the amount of effort required to test that system. For instance, in peer to peer communication systems there is a potential interface between each and every network node which would have to be tested. By restructuring the architecture to have a hub for connecting all components, each network node would only communicate with the hub, thus drastically reducing the number of possible interactions and with it the number of test cases.

The same is true for database accesses. If a particular database table is accesses by every component which uses it, then this table has to be generated to test every component. If, however, the accesses are restricted to a single access component, then the database table need only be generated to test that component.

Finally, the greatest contribution to testability is the separation of the use interfaces from the business logic. In an article in the IEEE Software, Robert Martin makes a strong case for an architecture that supports automated system testing. This can best be achieved by creating an open interface – XML or WSDL – between the user interface components and the back end components. It should be possible to bypass the user interface and to test the business logic directly via a test bus. Running the tests through separate APIs drastically increases the speed of the test and makes it possible to repeat the tests several times a day. This feature alone promises to reduce the test costs by more than 50%. [12]

From these measures, it can be seen that much could be done to increase testability and thereby reduce the testing effort. Experts claim that at least 33% of the complexity of software systems is artificial complexity, i.e. unnecessary complexity, which comes not from the problem itself but from the solution. [13] By eliminating this artificial complexity one could reduce the test effort by half. It would, of course, be better if the software were to be constructed from the beginning with testability in mind. This is one of the main goals of test driven development. However, if this is not done and it seldom is, then reengineering for testability can still be worth while before going into system testing. It might also mean that the development process is being complemented by a parallel reengineering process, intended to raise the quality of the software, including testability. (see Figure 2: Parallel Projects)



5 Conclusions

Testing requires a significant amount of a software project budget, in internet, distributed systems, data warehouse and integration projects well over 50%. According to a leading SAP manager over half of their development resources go into testing and integration. [14] Modern technology such as web services only increases the need for more testing, since it multiplies the number of potential paths throughout the software network. On the other hand at least 33% of a system's complexity is artificial. It is caused at the unit level by sloppy, unconsidered coding, at the component level by unnecessary and redundant functions and data, and at the system level by an over complicated architecture and overloaded user interfaces. Much of this artificial complexity could be removed, thus significantly reducing test costs. [15]

Reengineering software for testability is definitely a worth while effort. Identifying and removing clones, refactoring deeply nested code and restructuring the architecture are tasks that can be automated. Several tools exist which support that. By using them, reengineering costs can be minimized. Other tasks such as algorithm optimization, merging data accesses and simplifying user interfaces can be done manually at a rather low cost. In view of the potential savings in testing costs, it is well worth it to invest in a reengineering project running parallel to the development project.

References:

01] Fowler, M.: Refactoring–improving the design of existing code, Addison-Wesley, Reading, MA., 1999, p. 53
02] Sneed, H.: "Measuring the Reusability of Legacy Software Systems", Software Process – Improvement and Practice, Wiley Pub., No. 4, March, 1998, p. 43
03] Sneed, H., Nyary, E.: „"Downsizing large Application Programs", Journal of Software Maintenance, Vol. 6, No. 5, Oct. 1994, p. 235-248
04] Sneed, H.: "The Economics of Software Reengineering", Journal of Software Maint., Vol. 3, No. 3, Sept., 1991, p. 163-182
05] Sneed, H.: „Planning the Reengineering of Legacy Systems", IEEE Software, Jan. 1995, p. 24
06] Henard, J./ Hick, J.-M./ Thiran, P./ Hainut, J.-L.: „Strategies for Data Reengineering, Proc. of WCRE-2002, IEEE Computer Soiciety Press, Richmond, Nov. 2002
07] Quante, J., Koschke, R.: „Dynamic Object Process Graphs" in Proc. of 10th European CSMR-2006, IEEE Computer Society Press, Bari, Italy, March, 2006, p. 92
08] Mens, T./ Tourwe, T.: „A survey of software refactoring" in Trans. of S.E., Vol. 30, No. 2, 1004, p. 126
09] Baxter, I./Yahin, A./Moura, L./Anna, M.S.: Clone detection using abstract syntax trees, in Proc. of ICSM-1998, IEEE Computer Society Press, Washington, Sept. 1998, p. 368
10] Cleve, A.: „Automating Program Conversion in Database reengineering", in Proc. of 10th European CSMR-2006, IEEE Computer Society Press, Bari, Italy, March 2006, p. 321
11] Merlo, E. et. al.: "Reengineering User Interfaces, in IEEE Software, Jan. 1995, p. 64
12] Martin, R.: „The Test Bus Imperative" IEEE Software, July, 2005, p. 65
13] Weyuker, E.J.: „Evaluating Software Complexity Measures" in Trans. on S.E. Vol. 14, No. 4, Sept. 1988, p. 1357
14] Sommer, W.: "Product Innovation Lifecycle Maintenance and Solution Management at SAP", Keynote Speech of CSMR-2006, Bari, Italy, March, 2006
15] Sneed, H./Jungmayr, S.: Software Testmetrik, GI-Informatikspektrum, Feb. 2006