December 1997

Software Testing for Specialised Applications-Screenflow Engineering: A case study

John Paynter
University of Auckland

Software Testing for Specialised Applications - Screenflow Engineering: A case study

John Paynter
University of Auckland, New Zealand,
j.paynter@auckland.ac.nz

Executive Summary
Software testing is a topic that has been widely researched over the past thirty years, yielding a multitude of different testing strategies, tools and techniques aimed at increasing the reliability of software. Many of the methods developed for earlier paradigms appear to be insufficient for modern applications. This paper introduces a method of software testing for data-driven applications based on a New Zealand case.

The method used is part of the Screenflow Engineering process. This process is based on the premise that computer system applications should share a common pool of data that is updated on-line and made available simultaneously to any user of any application. It is generally accepted that relational databases provide the best means of achieving this by presenting the data in a standard, easily understood and easily accessible manner. The Screenflow systems development methodology provides an effective and efficient means of creating applications based in such a shared relational database environment. It offers the following advantages:

- Applications structures are developed in cooperation with users in a diagrammatic manner that is analogous to the development of data structures.
- The relationships between applications structure and data structure are explicitly stated and readily understood by users and developers alike.
- Screenflow applications allow users to browse and to update the database in a standard manner that is easily learned, natural and user-efficient by providing access paths that follow inherent relationships within the data.
- The standardisation of application components maximises the opportunity for reusable code. This results in substantial improvements in application testability, quality, reliability and maintainability together with a large increase in development productivity. These factors are measurable.

The testing strategy uses a combination of white-box methods for testing the templates and black-box methods for validating the applications. These are seen as complementary approaches that are likely to uncover different classes of error. The white box tests are done at the time that the templates are created or modified, the black box tests are conducted during development of the individual applications. The approach is Top-Down using the subsystems's navigator as a driver (following the navigation paths of the screenflow diagram) although this can be changed as circumstances require.

The testing method consists of creating a test template. For each application module this can be changed to suit the special features and requirements of the application. The resulting Function versus Program Mode matrix can be checked off or otherwise annotated as the testing progresses.

The advantages (eg. separation of white and black box testing to different stages of the SDLC) and disadvantages (eg. the dangers of not adequately testing structures when the reuse is in a new context) of such a method are discussed.

This case demonstrates an example where testing using a combination of test plan, strategies and techniques could provide improvements to the outcome of the software development process and the quality of the resultant applications. Some of the principles discussed in this case can be extended to testing in other specialised environments such as object-oriented systems.

Software Testing
Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding (Pressman, 1997). The increasing visibility of software as a system element and the attendant "costs" of software failure are motivating forces for well-planned, thorough

testing. It is not unusual for testing to take between 30 and 40 percent of the total project effort. The objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort. If testing is conducted successfully it will uncover errors in the software and demonstrate that software functions appear to be working according to specification and that performance requirements appear to be met. In addition, data collected as testing is conducted provides a good indication of software reliability and some indication of software quality. Testing however cannot guarantee the absence of defects, nor can it 'test-in' quality.

Software testing templates
Testing is a set of activities that can be planned in advance and conducted systematically. Pressman (1997) reasons that a template for software testing - a set of steps into which we can place specific test case design techniques and testing methods - should be defined for the software engineering process.

The importance of software testing
During the early years of computing when hardware costs comprised the highest percentage of the overall cost of computer-based systems, little emphasis was placed on testing strategies and techniques. The software testing components had little impact on the overall system implementation time.

In recent years, this situation has been reversed due to the decrease in hardware costs and the use of advanced programming languages (eg 4GLs). Software is now the most expensive component in the majority of information systems projects (Conte, Dunsmore and Shen, 1986) requiring considerable testing effort to ensure that the software meets the requirements.

Many authors have frequently stressed the need for the creation of test plans early in the SDLC. Mullin and Hope (1996) reported that for 35 student projects assessed over three years, testing was the variable that could best predict software Quality.

A brief overview of testing methods
Software (and other) products can be tested in one of two ways: (1) knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational, at the same time searching for errors in each function; (2) knowing the internal workings of a product, tests can be conducted to ensure that the operation performs according to specification and all components have been adequately exercised. These test approaches are called *black-box testing* and *white-box testing* respectively.

A discussion on the relative merits and weaknesses of all of these techniques is beyond the scope of this paper. However, an important fact to note is that the majority of research has focused solely on black box versus white box testing methods. This paper looks at use of such methods, before considering a strategy for applying white-box and black box tests at different stages of a project.

Test Case Design
The design of tests can be as important as the initial product design. Yet testing is often treated as an afterthought. Test cases are scripted that may "feel right" but have little assurance of being complete. Considering the objectives of testing, test cases must be created which have the highest likelihood of finding the most errors with a minimum amount of time and effort. Black and White Box methods are used to optimally select such test cases.

Black box testing
These models apply tests to the software interface. They are used to demonstrate that software functions are operational; that input is properly accepted and output is correctly produced; and that the integrity of external information (eg database tables) is maintained. A black box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) performance errors, and (5) initialisation and termination errors.

Black-box testing techniques include graph-based testing methods, equivalence partitioning, boundary value analysis and comparison testing.

White box testing
White box testing examines the internal logical structure of the software. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The status of the program can be examined at various points to determine if the expected or asserted status corresponds to the actual status. White-box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box methods, the software engineer can derive test cases that (1) guarantee that all independent paths within a module have been exercised; (2) exercise all logical decisions on their true and false values; (3) execute all loops at their boundaries and within their operational bounds; and exercise internal data structures to assure their validity. White-box testing techniques include Basis Path (utilising Flow Graphs, Cyclometric Complexity and Graph Matrices to derive test cases) and Control Structure Testing (Condition Testing, Data Flow testing and Loop Testing).

Problems with testing models.
A number of problems with testing models have been identified. The most important ones include difficulties in exhaustively exercising all possible paths and combinations within a program. Exhaustive testing is impossible for large software systems. Accordingly a limited number of important logical paths should be selected and exercised. Important data structures can be probed for validity. Black-box and White-box testing are not alternatives, rather they are complementary approaches that are likely to uncover different classes of errors. The attributes for both black- and white-box testing can be combined to provide an approach that validates the software interface and selectively assures that the internal workings of the software are correct.

Testing for specialised environments and applications
As computer software has become more complex, the need for specialised testing approaches has also grown. The white box and black box testing methods discussed are applicable across all environments, architectures and applications, but unique guidelines and approaches to testing are sometimes warranted. Specialist areas include: Graphical User Interfaces (GUI), Client-Server Architectures, Real-Time Systems and Object-Oriented Systems.

We will now examine one NZ case based on a template-approach to application building to see if it can throw light on the appropriateness of testing techniques.

Screenflow engineering
Screenflow Engineering (Snow and Paynter, 1992) derives its leverage in constructing applications from two sources. The Screenflow specifications (Figure 1) could be derived from the initial data model (Figure 2) and knowledge of the business procedures. While Data Diagrams describe the structure of the database, Screenflow Diagrams describe the structure of the application.

The Screenflow Diagram is central to the development of Screenflow applications, being the primary document produced in the systems analysis phase. It is essential for detailed design and programming, and is useful in establishing and enhancing an understanding of the application between developers and users. This diagram is also a key component in documenting the application from both the technical and the user viewpoints.

The Screenflow and Data Diagrams
The Screenflow diagram shows the set of programs within a subsystem plus the relationship between them, that is, how you move from one to another (ie how you navigate through a relational database). This is controlled by the subsystem's navigator. The lines depict the paths that the user may follow in moving from one program to another. These lines, and the paths that they represent, come in two varieties. Each one-to-many relationship shown in the data diagram will normally be represented by a *downward* move from one screenflow program to another. Where a given entity has several one-to-many relationships, then the screenflow diagram will be drawn to show the navigation between the different programs sharing the same screentop. It is also possible to move between topless screens. These are known as sideways moves and are selected from the side menu. Downward moves are similarly selected from the down menu. For instance, a customer order screen will have the order (header) details on the

screentop and the order-line details on the screenbottom. These represent the one-to-many relationship "one order may have many order-lines."

Second, each screenflow program was based on a corresponding template program. There are ten basic screenflow templates each one dependent on the relationship between the data (entity-type) displayed on the screen-top and that on the screen-bottom. In addition to the template programs and navigator program, a security program controlled the users' access to the various subsystems.

Each box on the Screenflow Diagram represents one application program. As, in general, each program in a Screenflow application controls the input and output from one and only one screen panel, each box on the Screenflow Diagram also represents one panel.

The box is divided into five horizontal segments. The top segment contains the program/panel title as it appears centrally at the top of the panel. The second segment contains the program/panel name and the identification number of the template program on which the application is based. (This is not shown on the final documentation as it is irrelevant for the users.) The third segment contains a brief description of the program's purpose. The fourth and fifth segments contain the file name of the screentop and screenbottom tables respectively.

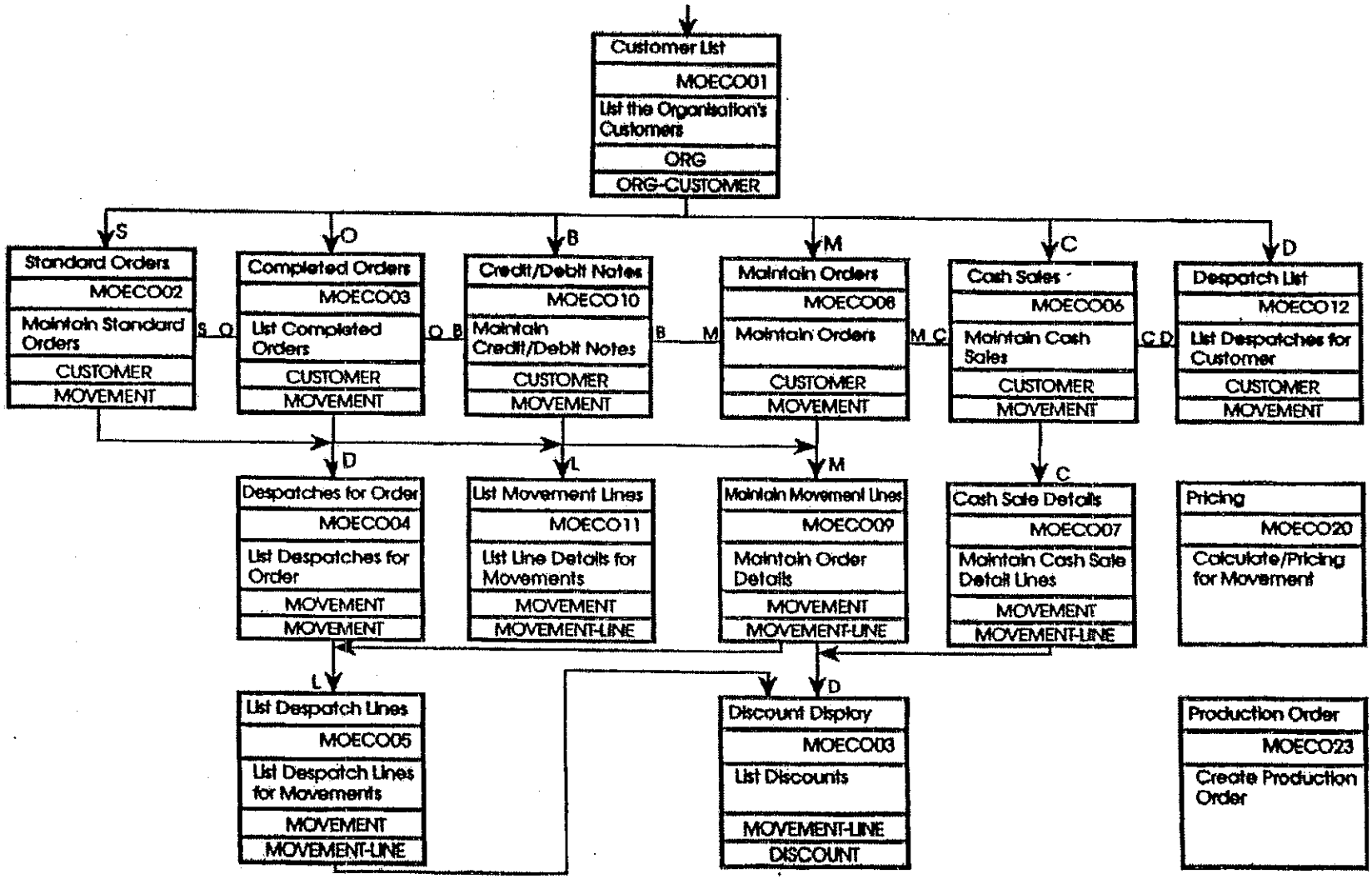

Figure 1. Screenflow diagram.

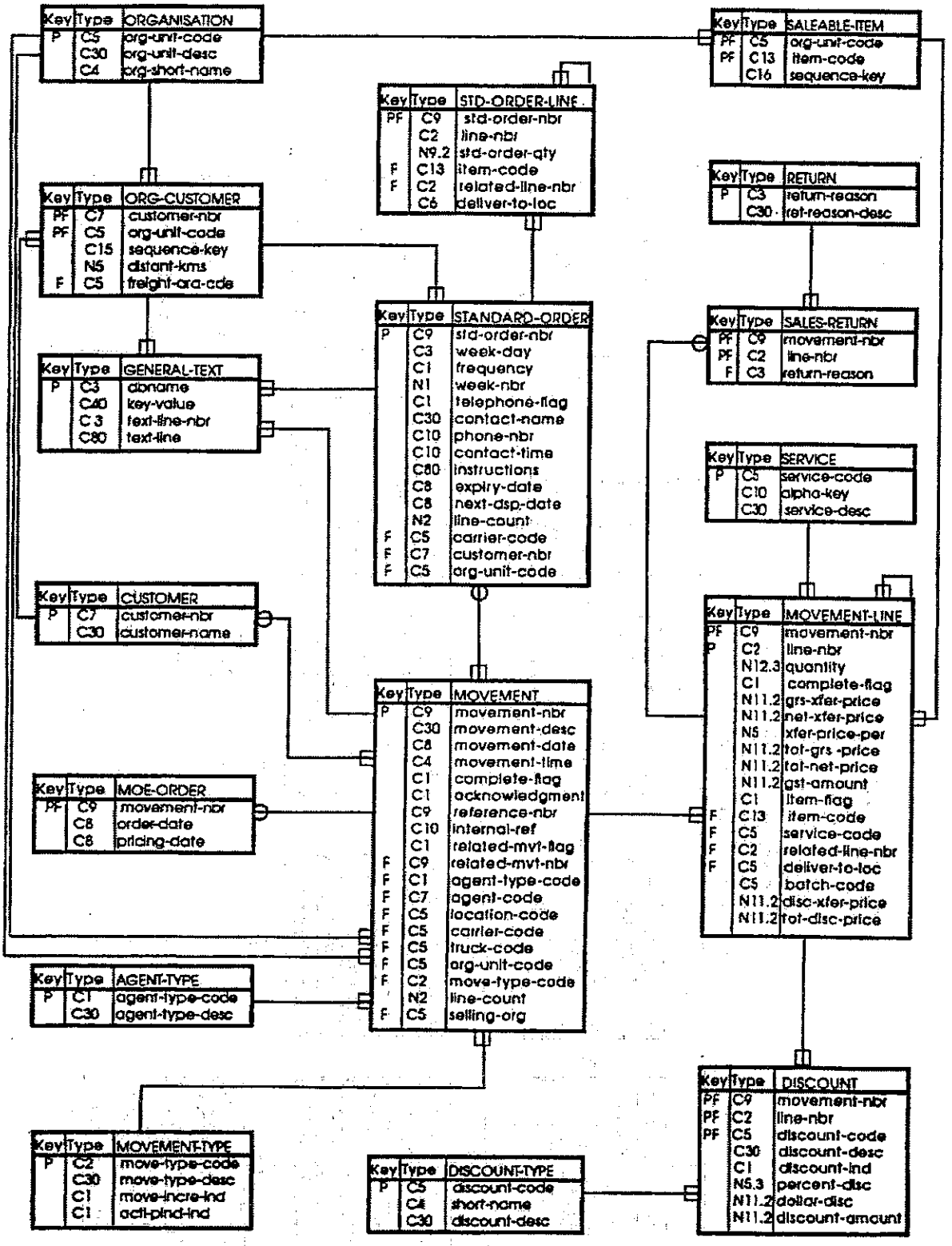

Figure 2. Data diagram.

Template programs
There were ten basic screenflow templates, each dependent on the relationship between the data (entity-type) displayed on the screen-top and that on the screen-bottom. Thus, the ten different types of data relationships and access rights (that is, update or display only) are represented in one of each of the programs. The template programs themselves differ only in the Modes present

The original template programs were created by two people over several weeks. The first system to be developed using them was the Refrigerated Freight Lines Weekly Trip Summary application. This system did not use the security application later adopted for all screenflow systems. These initial template programs were not robust enough during maintenance and enhancements. After six months, they were replaced by a new generation of templates and used in an Order Entry System (Paynter, 1995). As programs built with the old templates were maintained, the template program (ie. the basic program structure) they were based on was updated. Any specialist code written into the application programs (for example edit rules) was copied into the new version of the program, based on the new template.

These components of data model, screens, screenflow and template programs can be put together to create the screenflow engineering approach.

Screenflow and Object-Oriented Systems
The screenflow method lends itself to an Object-Oriented philosophy in that it is data driven. In common, the single factor that gives the greatest productivity leverage is Reusability. In addition the screenflow method whereby an entity is only maintained by one program - with its ADD, RELate, UPDate and DELete (ie CRUD) modes - associates the procedures directly with the data, ie functions are a property of data. Thus the data and functions of the entity (or Abstract Data Type) are encapsulated together.

This is taken one step further by the use of the template programs that enables you to focus on *what* you want to do and not *how*. Thus the screenflow program, representing maintenance of the CRUD activities of an entity, behaves like an Object Class. The focus on specifying the screenflow program concentrates on the "methods" available to other interacting Object Classes. For the sake of simplicity, private responsibilities (including the standard CRUD operations) common to all classes of objects are inferred and are not therefore modelled in the screenflow diagram. The public responsibilities that usually involve collaboration with one or more separate classes (invoked in a fashion reminiscent of subroutine calls) are shown on the screenflow diagram as the sideways and downwards menu options (Figure 1).

Navigation from one screenflow program to another is akin to the transformation diagram as depicted in Meyer (1988) Indeed the transformation diagram and screenflow diagram for an application are identical in their structural representation. Both can be derived from the aggregation and association relationships between entities (Meyer et al. 1991), with the inheritance relationship additionally handled in the case of OOPLs by delegation.

Meyer compares the object-oriented approach with classical functional decomposition (eg Gane and Sarson, 1979), discussing the use of "states" to depict the transformation of the system from one step to another. This was depicted for a hypothetical airline reservation system. The problem, as stated by Meyer, "...is to come up with a design and implementation for such applications, achieving as much generality and flexibility as possible." An equivalent screenflow can be constructed thus demonstrating that screenflow diagrams represent such a design and screenflow programs their direct implementation (Paynter, 1993).

Testing Screenflow Applications
Although, as described, templates were set up for the data model, screenflow diagram and the screens, Help Screens and Programs; and boilerplate documentation for the system specifications, it was not until the last project undertaken by the author that an attempt was made to create a testing template.

Testing Strategy
Testing strategies can be based on either a Top-Down or Bottom-Up approach. It begins at the module level and works "outward" toward the integration of the entire computer-based system. Screenflow systems favour the Top-Down approach. The navigator is equivalent to the 'driver' used to

test many bottom-level programs. As it is relatively straightforward to build the driver it is only necessary to define in it (as resources) the screenflow programs. Examination of the screenflow diagram (Figure 1) will reveal which programs can be written (and tested) first. Alternatively the navigator can be set to call any particular program and to pass the parameters required for that program. During the life of the project though, the modules were written and tested in a top-down order based on the screenflow, although sometimes the program that was to be initially invoked by the navigator was changed for another also at the top-level.

Unit Testing
As outlined in the introductory section, the major testing techniques involve either Black box or White box testing. The former is concerned with external function and validation, the latter with internal structures and verification. To a large extent the white box testing was performed in verifying that the templates worked. Consequently unit testing concentrated on black box methods.

Verification Testing
As outlined, White Box testing concentrates on exercising control structures, decision logic and loops. In testing the templates this involved checking the handling of 0, 1, n-1, n, n+1, and >2n records (where n is the number of screenbottom records). Parameter passing between the navigator program and the screenflow template programs was also checked. The parameters included the entity identifiers of the screentop entity(s) and the stack of downwards moves (program calls). These parameters thus represent the internal data structures of the application. The action of Insert, Add, Update and Deletion operations was checked against the database.

An example of condition testing was the Y/N Deletion logic in the program's Delete Mode. Another example was the transition from one mode to another (eg Related to List Mode). One aspect that was not tested thoroughly in the early templates was the handling of screenbottom scrolling when the sequence keys (which determined the order in which records were displayed on the screenbottom) were not unique. This problem was not picked up until unit testing of some of the later applications. The required modification was made to the second generation of screenflow programs.

Template Testing Strategy
It is not possible to directly test the template programs. The parameters used by a program must be entered (Snow and Paynter 1992) before the code can be successfully compiled.

As mentioned earlier the approach was started with RFL. "The first programs were created mostly to determine the functions they would perform. These programs were not pretty. So we then broke these down into their parts and started on the generic model. We would add the function to the program and then test only that portion that was added (eg. scroll forward, scroll backward). We did this iteratively until all the function was added to the one program (Top/Bottom update).

The rest basically were just copies of this one and were tested but not as much as TP01. The only exception to this was the navigator template program, however, it went through the same iterative process during its development as well." (Tod Elbourne, personal communication, 22 October 1996.)

Table 1. MOECO08 Test Template.

ProgramID: *MOECO08*	Program name: *Maintain Orders*					
Function	**Mode**					
	Related	**List**	**Insert**	**Add**	**Delete**	**Update**
PF1 Help	✓	✓	✓	✓	✓	✓
PF2 Upd	N/A	N/A	N/A	N/A	N/A	MVT1098✓
PF3 Pop	✓	✓	✓	✓	✓	✓
PF4 Quit	✓	✓	✓	✓	✓	✓
PF5 Add	MVT1099✓	-	N/A	-	N/A	N/A
Insert	N/A	-	-	N/A	N/A	N/A
PF6 Del	Disabled✓	N/A	N/A	N/A	-	N/A
PF7 Bck	✓	✓	N/A	N/A	N/A	N/A
PF8 Fwd	✓	✓	N/A	N/A	N/A	N/A
PF9 List	-	N/A	-	-	-	-
PF10	New ✓	-	-	-	-	-
PF11	Price ✓	-	-	-	-	-
PF12	M.order ✓	-	-	-	-	-
Sideways	S,O,B, C,D ✓	S,O,B, C,D ✓	N/A	N/A	N/A	N/A
Down	D,L,M, C ✓	D,L,M, C ✓	N/A	N/A	N/A	N/A
Screentop	✓	N/A	N/A	N/A	N/A	N/A
Sequence	✓	✓	N/A	N/A	N/A	N/A
Alt. keys	-	-	N/A	N/A	N/A	N/A
Special	P, M ✓	P, M ✓	-	-	-	-

Legend

N/A	Not applicable to template	xxxnnnn	Entity instance updated/created	S etc	Menu option
-	Not applicable to this program	✓	Function checked	Text	Additional notes

Validation testing

In order to do this a matrix could be created which mapped the function against the program mode (Table 1). Some functions did not pertain to certain modes or only could be performed in a specific mode (eg Deletions could only be initiated in Related Mode and executed in Delete Mode). By constructing this matrix is was possible to specify the tests to be performed for each mode. The testers could then fill in the matrix as they exercised the program. This exercise did not specifically cater for the various test cases required (eg using basis path testing). Each program had a series of edit rules. Any significant checks that need to be made could be inserted into the test template for the program (eg MOECO08 from Figure 1). It was necessary to test the program for Insertions, Adds and Updates for data both within and outside the range of valid values. For Delete mode it was necessary to attempt to delete data where deletions were possible and also when they would be prevented (eg in order to maintain referential integrity, in MOECO08 the DELete function is disabled). The test template could be printed and annotated or maintained on-line. Queries were developed to test each database operation (ie Add, Update and Delete) on an entity. These represented the external data structures used in the applications. When an entity was set up a query was created to display its attributes. The query was given a name corresponding to its three-character database name (eg MVT for the Movement entity). Where the entities had relationships to other entities the query was given the concatenated name (eg MVT-MVL for the Movement-lines corresponding to a particular entity). The conditions (ie where clause) would be edited when the query was retrieved to reflect the desired test. There could be separate queries set up for the different database operations where necessary (eg MVT-MVL-DEL, MVT-UPD).

During Integration testing, each of the navigation functions (Pop, Quit, Sideways and Downwards moves) would be tested and checked off. System testing was straightforward as all screenflow subsystems had a common architecture, 1992).

Discussion

Screenflow Software Testing

As we have seen from this project, different testing techniques are appropriate at different points of time. Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm (1981) states this a different way: "Verification - Are we building the product right? Validation - Are we building the right product?" Thus testing the template and enhancements to the programs created from standard templates (White Box) testing is a verification activity, whilst testing the resultant applications is a validation activity.

Screenflow engineering methods provide the foundation from which quality is built. Analysis, design and construction methods (coding from the templates) act to enhance quality by providing uniform techniques and predictable results. Throughout the process, measurement and control are applied to every element of the software configuration. Standards and procedures help ensure uniformity, and a formal SQA process enforces a "total quality philosophy".

Enhanced Screenflow Software Testing

Non-standard screenflow features could also be handled. These included multiple sequencing and search keys, changes to the screentop (eg scrolling forward and back through financial periods) and alternative screen bottoms in LIST mode.

Table 2. Estimation data for additional features.

Feature	Time (hours)
Sideways	0.5
Downwards	1
Alternate sequence keys	1
Additional screenbottom	4
New Screentop Record	5
Screentop Changes	3

As these figures show, enhancements could be made relatively rapidly. Hence no separate testing times were recorded.

After two years the organisation embarked on some more ambitious projects (Paynter, 1995). These were extensions of the sales and manufacturing applications. In the former, sales figures were summarised and presented on-line in a variety of formats. The user could nominate the customer (or customer-group), product (or product-group) combination that they wished to look at and select the time period involved. They could then scroll forward and back through the displayed records, change the financial period or look at the data in another way (using a screenflow option). The programming of this introduced a level of complexity over the standard database operations of retrieval and record storage (ie a one to one correspondence between displayed and stored data did not apply). A mechanism needed to be developed to index the retrieved data in working storage and to accumulate the data so it could be presented. After much trial and error the first subsystem (Sales by Customer group) containing the modified screenflow programs using this technique was put into production.

The second project involved creating front-end data gathering screens for the manufacturing operation. In this case several timesheet records would be entered and accumulated for a variety of personnel allocating their time across different work centres and jobs. At any stage the operator could choose to interface these records to the manufacturing system. This operation either created new transactions, transaction lines and allotments or altered existing ones. Unlike the previous example, updates were involved, thus the programming was inherently more complex. Mediating this however, was the fact that some of the lessons learnt on the previous project (eg accumulating records, switching time periods) could be put to good use. Relatively more time was spent on testing during these projects as they involved changes in the structure and function of the template programs.

Several classes of errors were encountered during these enhancements. Examples included date manipulation in the sales summary system (the function to convert a system date to the calendar date was not well documented); the pointer size used to hold the count of the number of records on the screen bottom was insufficient to hold the array of summarised records in the Sales Summaries; and record counters used in the Theoretical Consumption program were difficult to maintain when two sets of transactions, transaction lines and the allotment records were involved. All of these were either non-standard screenflow functions or using them for a purpose for which they were not initially intended.

Besides being time-consuming the testing process had other shortcomings. For instance, it was tempting to take short cuts, but this was dangerous where structures were used for purposes for which they were not originally intended. When programs were cloned and not adequately tested it was possible to introduce generations of errors, which once detected had to be traced back to the originating program.

Other problems posed for testing are inherint in the data-driven screenflow method. For instance surrogate (ie machine-generated) keys were used as unique identifiers. An implication of this approach to testing is that the only way to ensure a stable benchmark (for say, regression testing) is to back-up a standard set of master-file and transaction data. Care also had to be taken in debugging/antibugging - it was all too easy to forget to remove trace statements and the output from these could fill up the production database. As programs were created from the templates or cloned from application programs not all the functionality of the source program was required. It was therefore important to test for removed functions (eg PF keys which should be disabled).

Screenflow and the Object-Oriented Paradigm

It might be argued that as OOA and OOD mature, greater reuse of design patterns will mitigate the need for heavy testing of OO systems. However each reuse is a new context of usage and retesting is prudent (Binder, 1994), as was found in the screenflow examples where the basic functions were enhanced to perform a different role. It seems likely that more, not less testing will be required to obtain high reliability in object-oriented systems. This can be confirmed by reference to the above discussion of screenflow applications. For instance, it is easy to propagate errors when cloning programs. An audit trail must be kept of the program's ancestry in order to track and rectify any errors introduced.

To adequately test OO systems, three things must be done: (1) the definition of testing must be broadened to include error discovery techniques applied to OOAD models; (2) the strategy for unit and integration testing must change significantly; and (3) the design of test cases must account for the unique characteristics of OO software (Pressman, 1997). Whilst the latter two points are outside the scope of this paper, it can be seen from the screenflow example that the models (like OOAD models) provide substantial information about the structure and behaviour of the system. For this reason the models should be subjected to rigorous review before generation of the code from the template programs. During analysis semantic correctness must be judged based on the model's conformance to the real-world problem domain. If the model accurately reflects the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed) then it is semantically correct. To do this it is presented to domain experts to examine for omissions and ambiguity.
Like OO the focus of verification (ie white-box) testing is on the screenflow program which encapsulates the screentop and screenbottom entities' attributes with the different program modes (analogous to services or methods). Thus the screenflow program is analogous to the OO paradigm's class. Unit (or class) testing then focuses on the operations (modes) encapsulated by the template program and the state behaviour of the class.

Screenflow (like OO) system validation is black-box oriented and can be accomplished by applying the same black-box methods discussed earlier for conventional software. However scenario-based testing dominates the validation of OO systems, making the *use case* (Jacobson, 1992) a primary driver for validation testing.

Summary

Software testing accounts for the highest percentage of technical effort in the software process. However we are only beginning to understand the subtleties of systematic test planning, execution and control. The objective of software testing is to uncover errors. To fulfil this objective a series of test steps - unit, integration, validation, and system tests - are planned and executed. Unit and integration tests concentrate on functional verification of a module and incorporation of modules into a program (in this case, navigator) structure. Validation testing demonstrates traceability to software requirements, and system testing validates software once it has been incorporated into a larger system. Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases. With each testing step the level of abstraction with which software is considered is broadened.

This case demonstrates the use of verification techniques (white box testing) to test the program templates at the Design stage of the SDLC. Validation techniques (black box testing) are then used to check the application programs created from the templates. The implications of reusability to the testing process are examined in light of the shift to the Object-Oriented paradigm.

References

Binder, R. "Object-Oriented Software Testing" Communications of the ACM, vol 37, no. 9, September 1994, p29.

Boehm, B.W., Software Engineering Economics. Prentice-Hall, New Jersey, 1981.

Conte, S.D.; Dunsmore H.E.; and Shen, V.Y. "Software Engineering Metrics and Models". Benjamin/Cummings, California, 1986.

Gane, C. and Sarson, T. "Structured Systems Analysis: tools and techniques", Prentice-Hall, Ingelwood Cliffs, New Jersey, 1979.

Jacobson, I., Object-Oriented Software Engineering, Addison Wesley, 1992.

Meyer, B. "Object-oriented Software Construction" Prentice-Hall, Ingelwood Cliffs, New Jersey, 1988.

Mullin, K and Hope, S. "An Application of Quantitative Techniques to the Question of What Contributes to a Successful Software Development Project", Proceedings of the 1996 Software Engineering Conference, Melbourne, Australia 1996, pp.

Paynter, J. "Project Estimation using Screenflow Engineering", Software Engineering: Education and Practice, Otago, New Zealand, 1996, pp. 150-159,.

Paynter, J., "Architecture of a System - a case study based in developing applications in a RDBMS using a 4GL". Proceedings of the Pan Pacific Business Association Conference, Dunedin and Queenstown, 1995, pp. 388-390.

Paynter, J., "Implementing Object-Oriented Systems using Screenflow Engineering". Proceedings of the Information resource Management Association Conference, Salt Lake City, 1993d pp. 114-123.

Pressman, R.S., *Software Engineering: A Practitioner's Approach*. 4th Edition, McGraw-Hill, Singapore, 1997.

Snow, C. and Paynter J. "Screenflow Systems - An Engineering Approach to Building Data-driven Applications". Tamaki Report Series No. 1, University of Auckland. 1992.

Sweet, F., "Building Database Applications", Boxes and Arrows Publishing, 1986.