System and acceptance testing.

This type of testing deals with the testing of an integrated application, as opposed to unit testing which will test at the class level.

1. Testing from requirements.

System tests are derived from the functional requirements and acceptance testing will also relate to the non-functional requirements - performance/ speed, ease of use, reliability etc.

Each requirement in the requirements document should be traced to a test or tests

In many cases each requirement is associated with a user interface screen, there may be a whole screen to a given requirement or there may be many requirements that can be driven from that screen. Although it is too early to plan out the detailed graphics of the screen it is still important to identify the key elements of the screen, the components that can be used by the user to instigate the process defined by a requirement, the extra information needed to be displayed for this and the result of the operation of the requirement displayed suitably.

It is sometimes a good idea to show the client some of your thoughts, on paper, of how the requirement relates to your interface ideas. This can then lead to a clearer understanding of what is required.

The system will respond to some *external* stimuli, these will be, for example, users interacting with a screen entering data, choosing options through mouse clicks, ticking boxes etc., messages from some other system, perhaps the results of a query to a database,

A SIMPLE EXAMPLE

Suppose that we are building a simple customer and orders database. We might identify a number of stories such as the following:

1. Customer details are entered customer by customer.

2. Customer details can be edited.

3. Orders are entered by customer

4. Orders can be edited when necessary.

The details of the structure of the customer and orders details are left until later, we try to build an abstract model of the user interface and then refine it. The test approach permits us to generate an abstract high level test strategy.

Now we try to identify from these requirements, what is prompting change (inputs), what internal knowledge is needed (memory), what is the observable result (output) and how the memory changes after the event. We also try to identify the risk that the requirement will be changed during the course of the project as a means of trying to manage its evolution.

The idea of a memory is simply a device to represent the internal data storage which might be involved, look up tables, databases, special variables etc.

requirement	function	input	current memory	output	updated memory	change risk
1.1.1	click(customer)	customer button click	-	new customer screen	-	low
1.1.2	enter(customer)	customer details entered	current customer database	confirmation details screen	-	medium (nature of details liable to change)
1.1.3	confirm(customer)	customer confirm button clicked	(current customer database)	OK message and start screen button	updated customer database	low
....						
3.1.1	click(order)	orders button clicked	-	new orders screen	-	low
3.1.2	enter(order)	new order details entered	current orders database	confirmation orders screen	-	high (nature of details of orders liable to change)
3.1.3	confirm(order)	orders confirm button clicked	(current orders database)	Ok message and start screen button	updated orders database	low
3.1.4	quit()	click on return to start button	-	start screen	-	low

Table 1. Requirements table (part)

The table above describes some of the functions from the stories in this form.

Now consider some simple screens which may help us to visualise how it might work in practice.

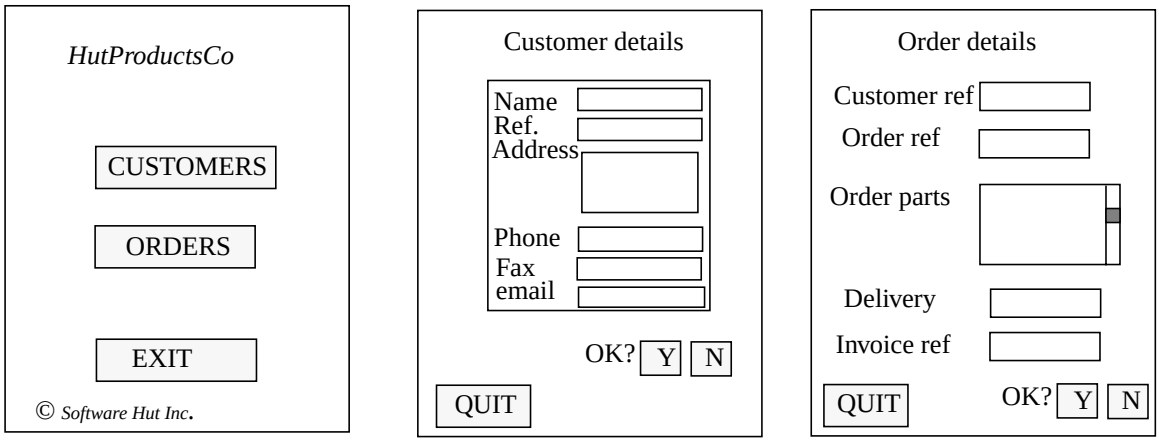


Fig. 1. Some simple sample screens.

The confirmation screens are not shown. Notice that we have filled in some of the data details which are not explicitly described in the requirements here.

The next task is to think about the order in which these screens will be deployed and the tasks that can be done on each screen. This information will be described using a state diagram.

From the diagram (Fig.2) one can see how the basic functions are organised. We build up a simple model like a state machine. Each state has associated with it an appropriate screen with

buttons, text fields etc. Of course, the model is simple and crude, there is no distinction between entering a new customer's details and editing an existing one but it is enough to explain the method. These refinements are, what they say they are, *refinements* that can be dealt with later and the work we do on test set generation here is built on them. The complete state machine is actually called an *X-machine*, a term first introduced in 1974! These machines differ from the standard finite state machine in the sense that there is a memory in the machine and the transitions involve functions that manipulate the memory as and when an input is received.

A good way to think about these machines is to draw the state diagram and remember that the transitions between each state represent system functions that are triggered (usually) by an external event (user actions) and carry out processing over some database or a global or local memory store. In many cases the functions can be described very simply. One could use Z for this but I prefer a simple functional notation.

This memory will be derived from the requirements in a natural way so we refer to the requirements table where the memory connection is described.

For example, the memory in this example is likely to be the database that contains the record of customers and orders.

The function click(customer) simply navigates between two screens and has no memory implications, but the function enter(customer) will involve some interaction with the database, it might search to see if the supposed new customer is in fact new before proceeding - generating a message if the customer is already on the database. we may also have a requirement that the user can ancel a data entry at suitable places in the interaction and this can be described by the abort(customer) function which has been developed as a result of thinking about the way the system fits together and the needs of users.

The function confirm(customer) actually changes the database by updating the records with the information relating to the new customer.

The memory structure now needs to be discussed. Essentially we need to think about this in terms of what basic types of memory structure is relevant at the different levels. At the top level, for example we could represent it as a small vector or array of compound types of the form:

customer_details × order_details

filling in the actual details later. It may be, for example, that these will represent part of a structured database with a set of special fields which relate to the design of the screens associated with these operations. So customer_details would involve name, address etc. which would be represented as some lower level compound data structure, perhaps and there would be basic functions which insert values into the database table after testing for validity etc.

Now we can describe some of the functions from the diagram. First note that the memory is just a set of records with 2 main fields, the first structured into:

customer_details : name, address, postcode, phone, fax, email

and the second into:

order_details : customer_ref, order_ref, order_parts, delivery, invoice_ref

Then the set of all *current* customer details is given by:

Customer_details

so Customer_details might be {customer1, customer2, customer3} after 3 customers have been put in,

where customer1 = [name1, address1, postcode1, phone1, fax1, email1] and so on.

The set of all possible sets of customer details that there ever could be can be defined as CUSTOMER_RECORDS.

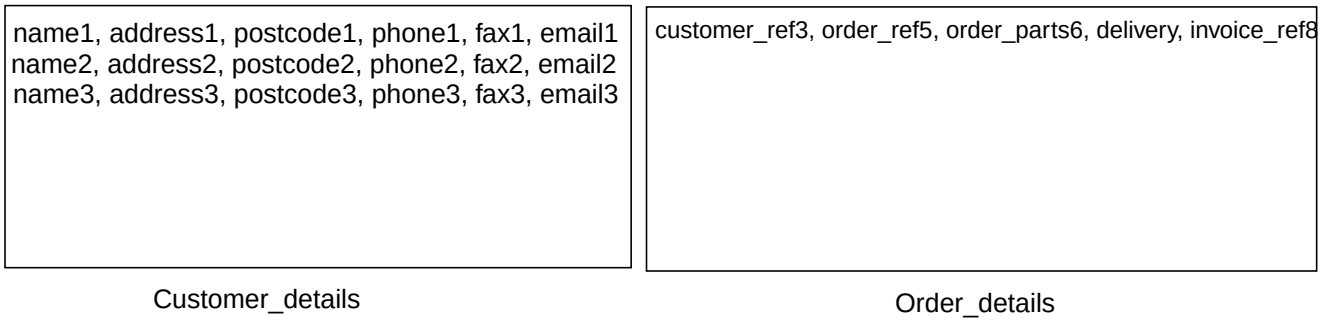
So Customer_details ∈ CUSTOMER_RECORDS

And the current order details is {order1} after one order has been put in, eg.

Order_details = {order1}

where order = [customer_ref3, order_ref5, order_parts6, delivery, invoice_ref8]

Thus Order_details ∈ ORDER_RECORDS



The state of the memory after 3 customers have been inserted and one order.

Let the set of current memory values is given by:

Customer_details × Order_details

A typical element of this memory set is

([name, address, postcode, phone, fax, email] ,[customer_ref, order_ref, order_parts, delivery, invoice_ref])

The function definition for enter(customer) would now look like:

enter(customer) ([name, address, postcode, phone, fax, email] , (Customer_details × Order_details)) =
((Customer_details × Order_details) ∪ {([name, address, postcode, phone, fax, email], -)}, display)

if [name, address, postcode, phone, fax, email] ∉ Customer_details × Order_details

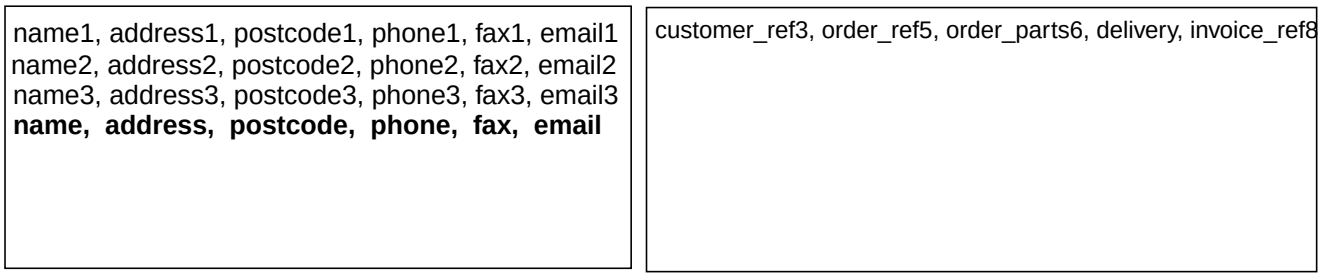
else error message : “customer already present”

The precondition is that these details are not already in the database. The order details are not altered we just update the customer details. The display element is the visible screen message asking for confirmation of the input data (the screen for state **confirmcustomer**) or an error message.

The function’s type would be:

$$\text{enter(customer)} : \text{INPUT} \times \text{CUSTOMER\_RECORDS} \times \text{ORDER\_RECORDS} \rightarrow \text{CUSTOMER\_RECORDS} \times \text{ORDER\_RECORDS} \times \text{OUTPUT}$$

Here INPUT is the set of all possible inputs/data entry and OUTPUT is the set of all possible displays. Note that this function is a *partial* function.It can only operate if certain pre-conditions hold such as the input is of the correct type.



This is now the state of the memory after the enter(customer) function has been applied with the new customer data: name, address, postcode, phone, fax, email

All the other functions in the diagram can be defined in a similar way.

All types of systems can be described in this way - we need to identify the functions involved, the data and memory they use and the states that sort out which function is valid and when. Such X-machines are extremely powerful - as powerful as Turing machines and much more useful!

We are going to use this state diagram to define how we can build a set of functional system tests that will link to the requirements and be extremely effective.

Most testing is *ad hoc* in the sense that the creation of the tests is left to the tester’s common sense. This may not be a very effective way of finding faults - which is what testing is all about.

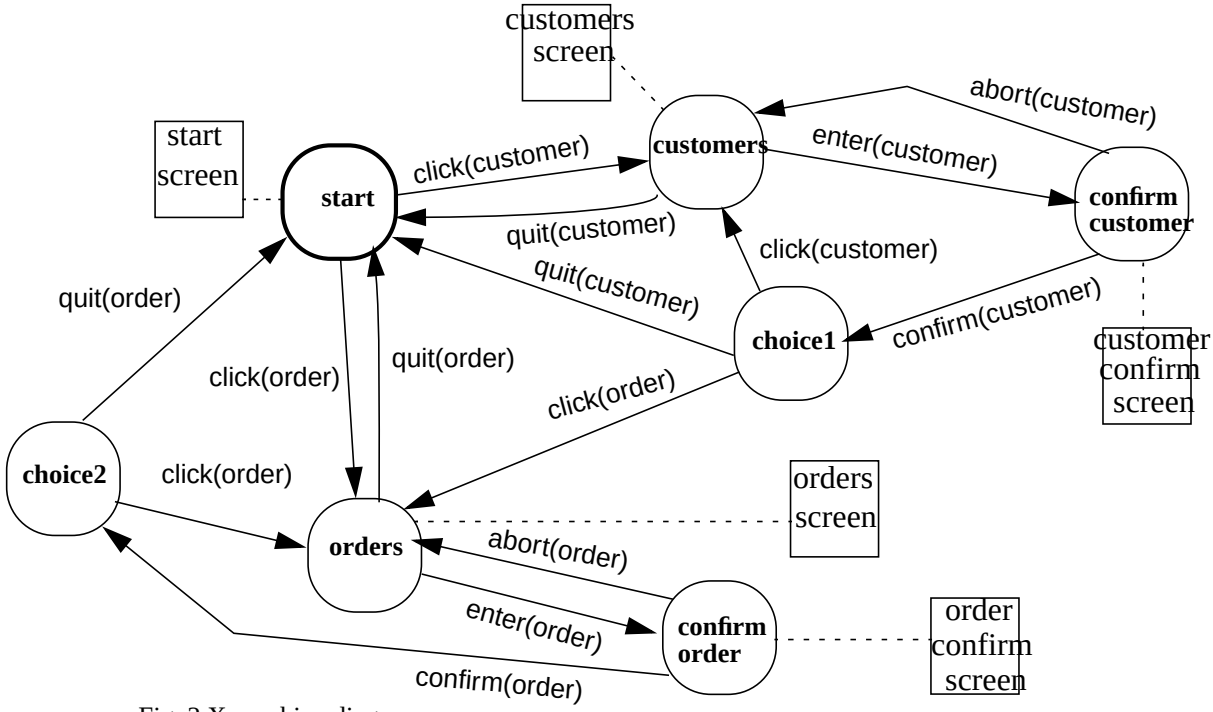


Fig. 2 X-machine diagram

2. Tests.

First, what is a *test*?

It is an application of some user or system input in a particular state of the system to establish if the system response is what is expected or is faulty in some sense - it might produce incorrect response or output, no output, or the system might crash.

Going back to our interface screens (fig. 1) we might consider the first screen and apply a test by clicking on the *customers* button. The expected result is a new screen, the customer's screen. That is all. We would not want the database to be deleted also!!

When in the customer's screen we might wish to test the data entry of the customer name and address. We have to specify the state that the test starts from and the data that we will insert. Now the data entry requirement will be based around some part of the software architecture, some classes or functions that accept user input and do things with it. The extent of the system testing that needs to be done is related to the level and extensiveness of the testing that has been carried out at the unit testing stage.

For example, if there are data integrity checks being carried out then these have to be tested. If there are table look ups, for example the system might check that a specific postal code exists by consulting an official list of these, or it might need to check that the format of the input is correct - letters where letters are expected and numbers also. No control characters etc. It might need to check whether the customer is already registered on the database and this will involve a query of the current database state.

The decision as to where the testing should be done - unit level or system level is not always clear. Obviously the more complete the testing at unit level the better but it is not possible to do all the things that are needed since inter-class communication and communication with databases and tables may not be possible at that point in the project.

We will have to include in our system testing, test cases which will expose faults in the data entry checking. We would do this by having test cases with *daft* input, for example. This might be invalid symbols or no symbols - perhaps just return - and so on.

We should also test the system under different conditions relating to the database, if any. For example, at the beginning the database is empty, we would test the system under these conditions, also when the database has some data in it and when it has a lot in it.

Where data entry has been carried out and hopefully stored in the database we need to establish that the data has been stored correctly. This needs either setting up queries as part of the test or writing a suitable script to pull the data out to check that it is OK.

So the test cases must reflect all of these things - what we put in and what we expect to see, and what we actually see.

If we can build a machine diagram like fig.2 and relate all the main requirements to it we can then create some very powerful test cases.

3. Testing strategy.

We identify paths from the start state and derive a test for each path. However, we do more than that. Notice that the paths through the machine involve driving the system between the states by carrying out the various functions that are available at each state.

So we could consider the path obtained by operating the following functions:

```
click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer) ; confirm(customer) ;
click(order) ; enter(order) ; abort(order) ; quit(order)
```

At each point in this sequence we will be submitting data values or mouse clicks and we will have to choose these to enable the test to be carried out. We expect certain things to happen and these have to be recorded and the test will be evaluated in respect of detecting what actually happened and seeing if this matches the expected behaviour. Did the buttons work, do the correct screens get displayed, was the correct data put into the database, were the correct error messages displayed (if appropriate) and the correct screen displayed subsequently etc.?

This test tests what should be there. However it often happens, particularly with OO programs that the system can do unexpected things which were not planned for. We need also to test that *the functions that are not supposed to be are not there!*

As an example, once we have reached state **customers** it should not be possible to use any of the order functions that are available for the **orders** part of the system. we could do this by trying to see if we can make these functions work as part of a test. So we ought to test that the data we entered for the customer does not also get put into some other part of the database dealing with orders.

Thus we can assemble a set of test cases based on paths of various lengths through the machine diagram and tests of the non-availability of functions in certain states. Here are some more examples.

click(customer)
click(customer) ; enter(order)
.
.
click(customer) ; enter(customer)
click(customer) ; enter(customer) ; enter(order)
.
.
click(customer) ; enter(customer) ; confirm(customer)
click(customer) ; enter(customer) ; confirm(customer) ; enter(order)
.
.
click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer)
click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer) ; enter(order)
.
.
click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer) ; abort(customer)
click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer) ; abort(customer) ; enter(order)
.
.
etc. etc.

Recall that the enter(order) test will look at the orders database to see that nothing has changed. The other functions of the system would also feature in a similar way.

Clearly this will lead to a lot of tests and automation is required to manage the size of the test set. In industry, sometimes very expensive, test tools and environments are available to generate tests, to apply tests and to analyse test results. Some of these can be found on the Internet and we have some test tools that support this approach to test set generation.

Another, rather draconian, general test is to reboot at an arbitrary point in the program, this is important since some users may panic and do this, we need to ensure that the minimum data is lost in this situation.

4.Test documentation.

It is vital that all the tests are properly documented so that testing can be carried out systematically and effectively. We also need to keep a record of the results so that the quality assurance can be convincing. Maintenance will also require information about the testing results.

For each requirement, which should be properly numbered in the requirements document, we will generate a set of tests. The details should be kept in a suitably designed spreadsheet.

Here is an example:

Table 1: Systems/acceptance test definitions

Require ment	Test reference	Test purpose	Test input	Constraints/ prerequisites	Expected output	final state	comments
1.1.1	1.1.1	test front page	load program	browser open	page loads	start	
1.1.2	1.1.2.1	load custom-ers page	click(custom-ers)	start page open	customers page displayed	customers	
	1.1.2.2	load custom-ers page	type random keyboard characters	start page open	no change in display	start	invalid input
	1.1.2.3	load custom-ers page	reboot	start page open	close down, data-base unaffected	-	invalid input
1.1.3	1.1.3.1	enter cus-tomer details	standard data entry1*	customers page open	data displayed	confirm	
	1.1.3.2	enter cus-tomer details	standard data entry2*	customers page open	data displayed	confirm	
	1.1.3.3	enter cus-tomer details	standard data entry3*	customers page open	data displayed	confirm	
	1.1.3.4	enter cus-tomer details	empty data entry	customers page open	error message	customers	invalid input

* The definitions of standard data entry1, standard data entry2, standard data entry3 need to be made somewhere in an appendix to this table.

The next stage is to try to automate the testing as far as possible. We need to create a file of test

Table 2: Test data file

Test ref.	Function sequence/path	Test sequence	Expected output	Final state
.....				
2.3.2.1	click(customer) ; enter(customer); enter(order)	click(customer); enter(standard data entry1); enter(order)	no change to d'base	confirm customer
.....				

inputs, one set of inputs for each test. These could be kept in a spreadsheet, the test data file - Table 2 - and a script written to extract these inputs and put them into a standard text file, one line per test. Another script would extract each input sequence and apply it to the code. This is sometimes easier to do in some cases than others. With GUI front ends it is sometimes difficult

to access the key parameters/events from inside like this and anyway one would want to test the overall programme as well. Test software is available to automate a lot of the interface interactions.

Table 3: Test results table - system version 1.0

Test ref.	Date/ personnel	Result pass/ fail	Fault	Action	Comments
1.1.2.1	12/3/02 Pete	P	-	-	-
1.1.2.2	12/3/02 Pete	F	system crash	debug	Jane alerted (13/3/02)
1.1.2.3	12/3/02 Pete	P	-	-	system closes - no losses

The test results file - Table 3 - is a vital resource which will have to kept up to date during testing. It describes what has been done, what has been fixed and what remains to be done.

5. Design for test.

Sometimes, it is hard to test a program because it has not been designed to make testing easy. This will usually result in a poor quality program since testing is very expensive - as you will find - and many software developers will stop testing, not when the system is suitable for release or delivery, but when they run out of money in the test budget. Often they take a risk that the cost of fixing the client's bugs later, or of supplying patches, is cheaper than continuing testing in-house. The client ends up doing some of the testing and they may not appreciate it!

In order to make the testing easier we introduce two strategies that will help. These are called *design for test* principles.

DFT1 Controllability.

This amounts to designing the X-machine of the system so that you can access any state with any value in the memory. It can be achieved by using a special test input to do this.

This issue mainly arises when there are functions that exist in several places of the machine. We wish to send data (inputs) directly to them without going through intermediate states which may change the internal memory in ways that will not allow the functions to be fully tested, for example preventing the preconditions to be satisfied or violated in some way.

See the diagram below:

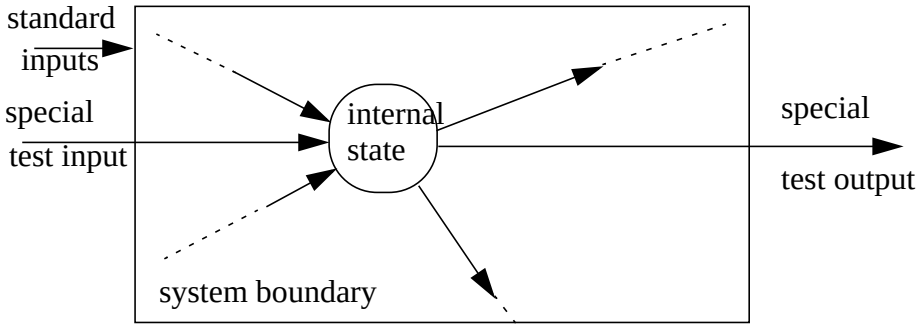


Fig. 3 Illustrating how to achieve design for test compliance.

We write special code to access this function under the conditions that we need, setting, for example, internal variables to suitable values.

DFT2 Observability.

This problem arises when we have carried out a test but we are not sure which function has operated and what it has done. The outputs might have been masked by other activity. The solution here is to define a special output value which is used to determine if the test has run properly. We therefore write some extra code which will print out, for example, some critical variable values, messages which will tell us what has happened etc. This is a common practice in programming where you often interrogate a variable to see what its value is etc. during debugging.

In both of these cases we have code in the implementation that is used only for testing and is not part of the original requirements. We can either leave it there or remove it - comment it out, for example - but whatever you do it needs to be done with care or else it might break the system!

6. Summary.

There are many aspects to testing, we have only just scratched the surface. Later we will look at unit testing and testing for non-functional requirements. For further information about the type of testing described here consult the following book: M. Holcombe & F. Ipate, “Correct systems - building a business process solution, 1988, Springer Verlag.

The testing of web sites is a specialist activity in itself and requires a lot of understanding of the technology and of the key issues at both the client end and at the server. For critical e-commerce business there are many security threats also. You have to know what you are doing!!