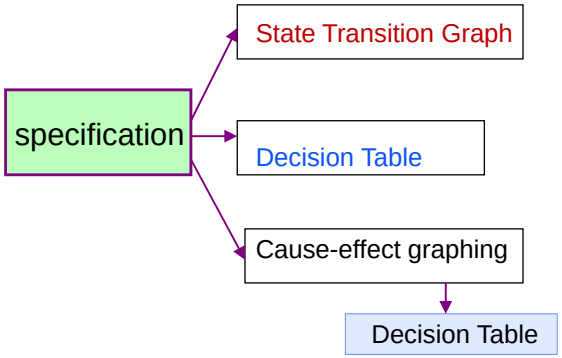
Course Structure

1. Software Quality Assurance
2. Testing Fundamentals
3. Code-based Techniques
4. Specification-based Techniques- Part II
5. Inspection Technique
6. Test Tools
7. Measuring Software Quality
8. TDD

Spec. Dependent Testing Techniques

Introduction
Spec-independent
▸ **Spec-dependent**
Other

If the specification can be expressed in a specific representation, then we can use one of the following techniques to identify test cases:



Do Finite State Machine testing

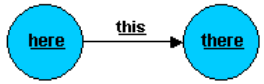
Do Decision Table testing

Do Cause-effect graphing testing (which actual uses decision table testing)

Finite State Model

Introduction
Spec-independent
Spec-dependent
▸ **FSM**
DT
CE
Other

A **model** is a description of a system's behavior. It helps us understand and predict the system's behavior.



A model

Ack: Harry Robinson

- A **finite state model** has finite number of **states, inputs/events** and **outputs/actions**.
- When an **input** or an **event** occurs, the FSM may **output** something or take an **action,** and may make **transition** to a new state.

Finite State Model

Introduction
Spec-independent
Spec-dependent
▸ **FSM**
DT
CE
Other

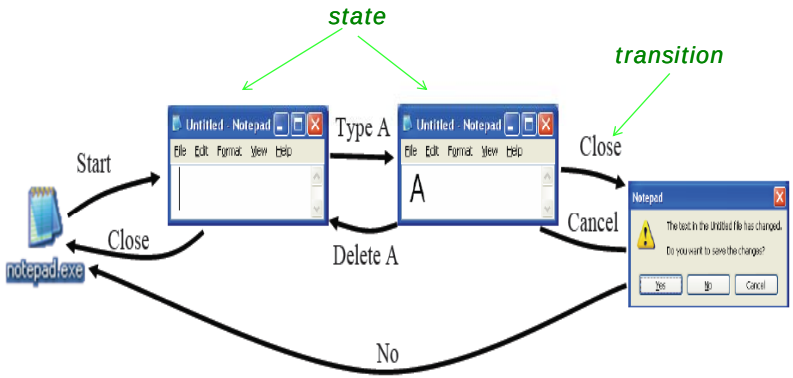
Many systems have the characteristic of 'memory'. That is, past behavior can influence future behavior.

Example: a pay phone must remember how much money has been inserted to determine how much service to provide.

In a finite state model or machine, its **states** serve as the memory.

The machine is always in exactly one state.

Example FSM: Notepad



Finite State Model Testing

State: a FSM has a fixed set of possible states. It is a condition in which the system is waiting for one or multiple events. e.g., state of a person: working, eating

Transition: change from one state to another. A transition is triggered by an input (or event) occurring in a particular state.

Reachable state: A state X is reachable from state Y if there is a sequence of inputs such that, when starting from Y, the FSM will end up in X. (There is a path from Y to X).

- Many computer applications can be modeled as a finite state model, because there is a limited number of different internal conditions (i.e., states) and rules which determine when they change from one state to another.
- FSM are good for testing
  - menu-driven applications (e.g., websites)
  - GUI
  - computer game
  - telecommunications systems,
  - communications protocols,
  - control systems,
  - embedded software, and
  - OO design.

Example: Telephone System

A telephone is a device that at any given time is in one of several states.

Example: the ***state*** can be *talking, ringing, dialing,* or *on hook.*

User actions cause the current state to change.

Example: a user picking up a ringing telephone causes the current state to change from ***ringing*** to ***talking***.

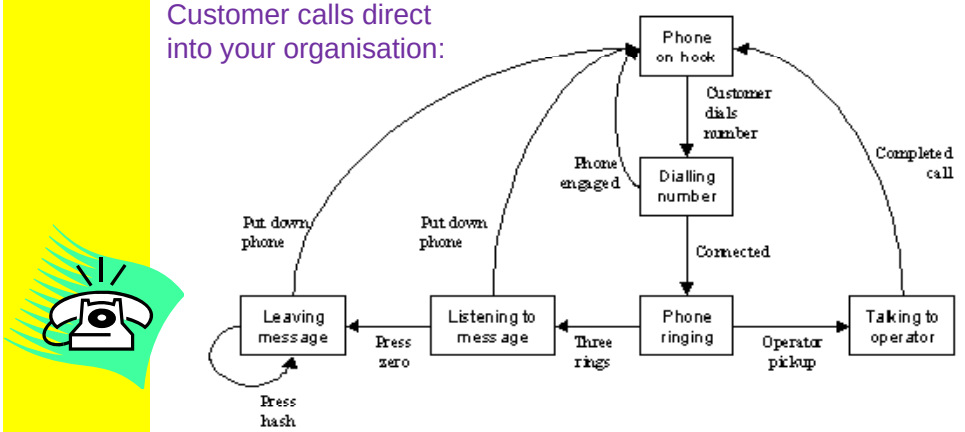
When the system obtains a new input, the machine moves to another state and performs an action, which may be used to produce an output.

States (shown as **nodes**) are connected by labeled directed arcs denoting transitions among the states.

Labels on the arcs consist of 2 parts:
- The **input / event** provoking the transition.
- The **output / action** preformed

State Transition Graph: Phone System

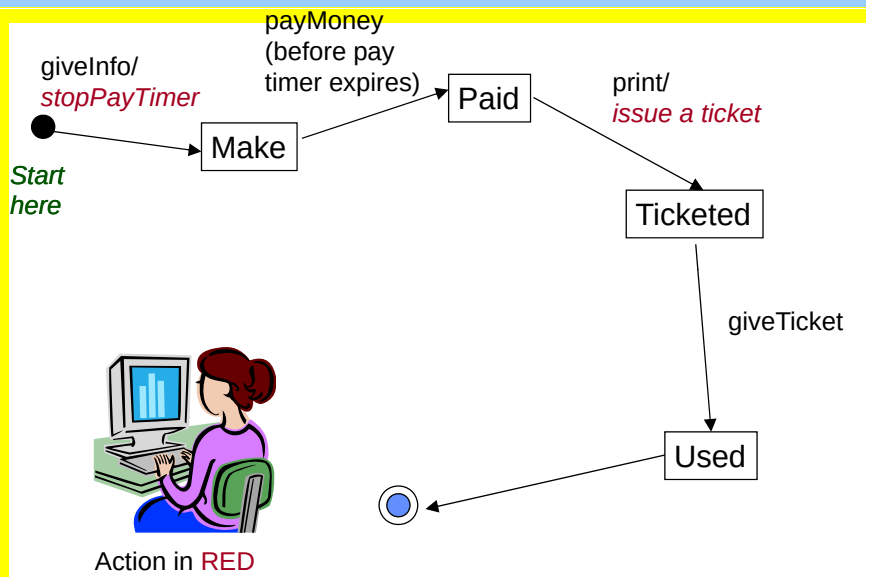
Customer calls direct into your organisation:



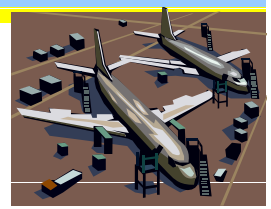
For this example, labels of arcs show the event only, and do not show the output/action.

This transition graph is strongly connected (i.e. each state can be reached from any other state).

Example: Airline Reservation

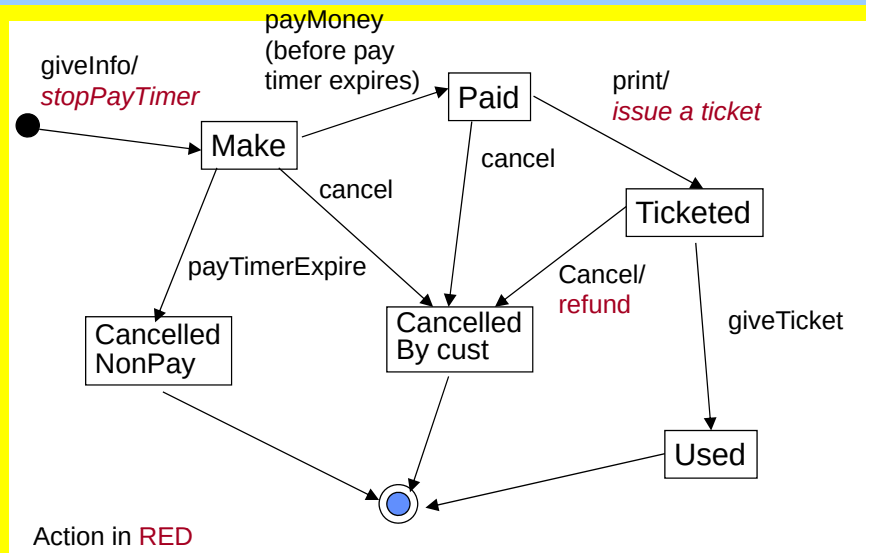


Example: Airline Reservation



- We have modelled the normal interaction or flow.
- ***What about cancellations?***
- Under which situation can the customer cancel their reservation?
  - Customer can cancel in Make state, Paid state, and Ticketed state
- Also, what if the reservation timer expires (i.e., the user took too long to complete the booking)?

Example: Airline Reservation



Example: Airline Reservation

State transition table:

Current state	Input/event	Output/Action	Next State
start	giveInfo	stopPayTimer	Make
Make	payMoney		Paid
Paid	print	Issue ticket	Ticketed
Ticketed	giveTicket		Used
Ticketed	cancel	Refund	CancellByCust
Paid	cancel		CancellByCust
Make	cancel		CancellByCust
Make	payTimerExpire		CancellNonPay

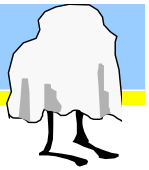
Common Problems in FSM

- ⌘ Missing transitions
- ⌘ Incorrect actions
- ⌘ Incorrect next state
- ⌘ Invalid path

What to test?

- ➔ Test every state
  Can we reach every exit state?
- ➔ Test every event
- ➔ Test every transition (transition coverage).
- ➔ Test every path

Testing Transition



Transition coverage =

$$\frac{\text{Number of transitions exercised}}{\text{Total number of transitions}}$$

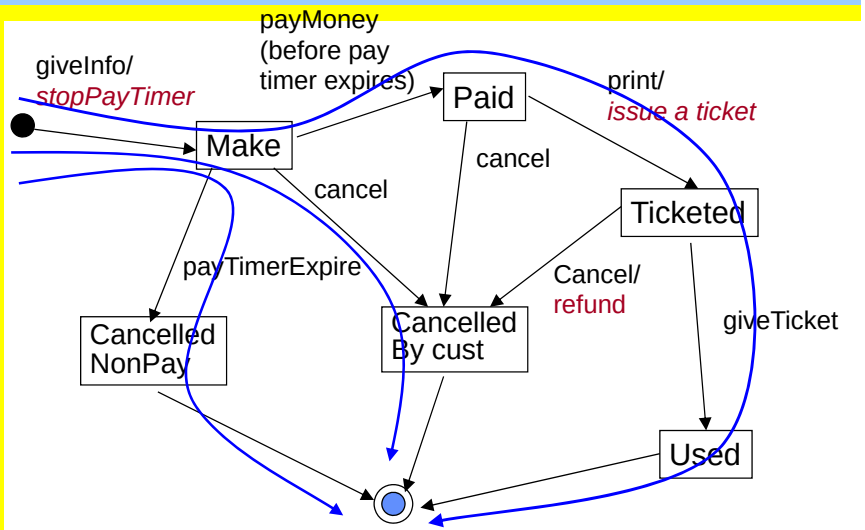
2-Transition coverage =

$$\frac{\text{Number of sequences of 2 transitions exercised}}{\text{Total number of sequences of 2 transitions}}$$

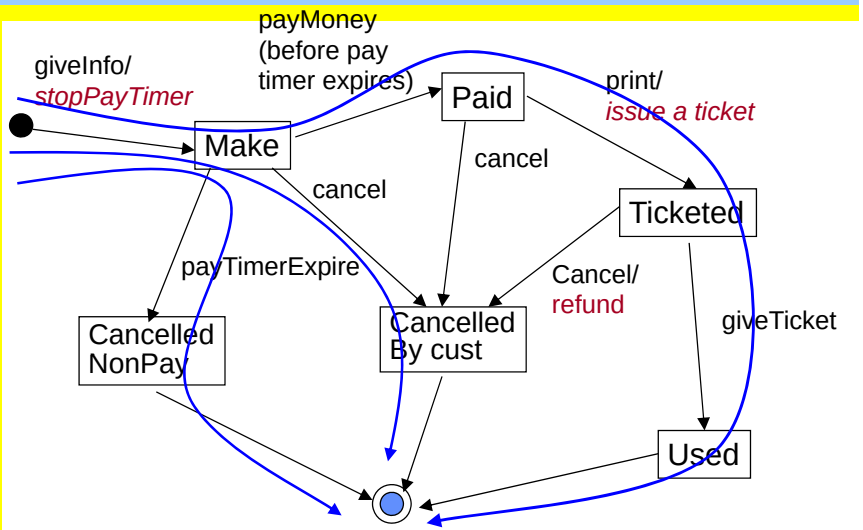
We can have **3-transitions**, **4-transitions**, …, **n-transitions** coverage.

Many companies require transition coverage as a minimum criterion.

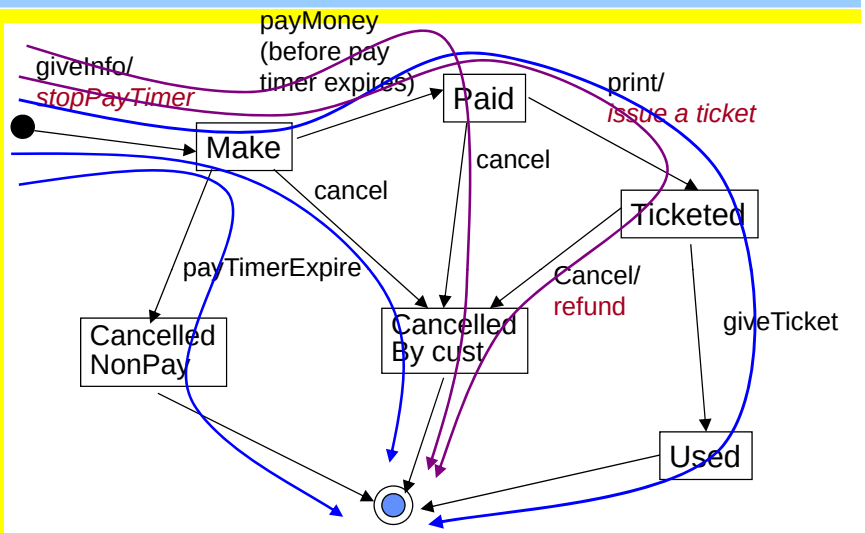
Test All States (every state executed)



Test All Events (every event executed)

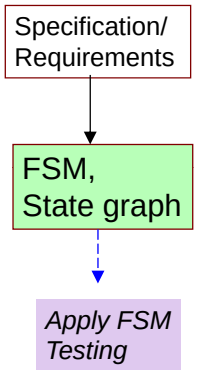


Test All Transitions



Also tested All Paths in this case!

FSM Testing

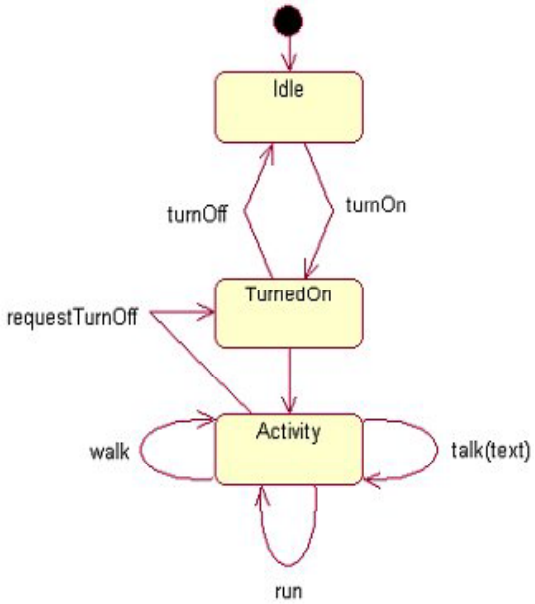


- Given a system specification, ***if*** we can express/model the functional specification in term of a FSM, we can use FSM testing
- We focus on how the system interacts with the world, and the order of events.
- Every test should start from the initial state, make a tour of various states to get to another state (the **target state**), and then return to the initial state or the final state.
- Try to achieve at least **all transitions testing**.

Exercise 1

Given the state graph (for a robot):

What test cases to achieve 100% transition coverage?



Steps for applying FSM Testing

Create a FSM

1. Identify all entities (e.g., turn on, turn off, output)
2. List all possible events (actions, rules) and states (operational models) associated with the entities.
3. Determine start state, input, output and finish state.
4. Draw the state transition graph
5. Review the state transition graph. Make sure it meets all the requirements and is complete.
6. Develop a state transition table as a further check for completeness.

Identify tests

1. Determine coverage level (all states, all events, all transitions, or all paths)
2. Define tests

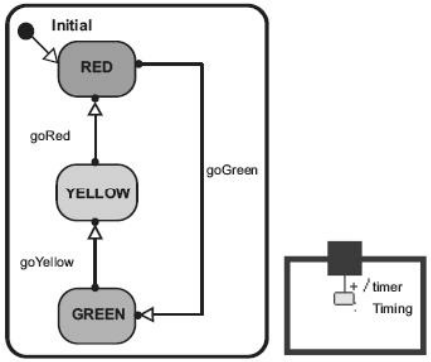
Example: Traffic Light

State No., Name	Description	Attributes
Initial	System not running	Timer event set to 30 seconds
Green	On timer time-out event go to(goYellow) Yellow	Timer event set to 45 seconds
Yellow	On timer time-out event go to(goRed) Red	Timer event set to 10 seconds
Red	On timer time-out event go to(goGreen) Green	Timer event set to 30 seconds

State Specification Template

Requirements for traffic light:

- The traffic light has 3 colors: red, yellow, and green.
- To change from green to yellow, an event of time expiring is needed.
- The events always happen in a fixed sequence of green to yellow to red.
- It is not allowed to go from green to red.



FSM Testing

⌘ The complexity of a FSM is given by its *state-input product* = number of states x number of inputs.

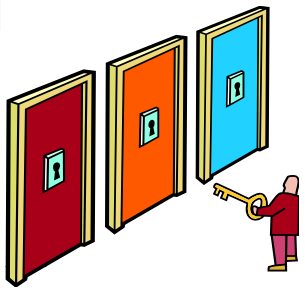
⌘ If this is high, then the FSM is very complicate and will take large effort to test.

State-input coverage = Number of state-input pairs exercised / state-input product

Limitation of FSM testing:

- Hard to apply to large state graphs
- No commercial tools yet.

Decision Table Testing



Decision Tables (1)

Conditions	Condition entries
Actions	Action entries

A decision table is used to record complex business rules.

A decision table consists of

⌘ **conditions** listing the conditions relevant in the decision situation,

⌘ **actions** containing a list of all possible actions,

⌘ **condition entries** containing the combinations of the states possible for the conditions, and

⌘ **action entries** that can be used to define the corresponding actions to be carried out.

Condition = cause action = effect = expected results

Decision Tables (2)

		rules							
		R1	R2	R3	R4	R5	R6	R7	R8
conditions	C1	T	T	T	T	F	F	F	F
	C2	T	T	F	F	T	T	F	F
	C3	T	F	T	F	T	F	T	F
actions	a1	x			x	x			x
	a2	x							x
	a3		x				x		
	a4			x	x			x	x
	a5	x			x				

values of conditions

actions taken

R1 says when all conditions are T, then actions a1, a2, and a5 occur .

Example Decision Table

Specification:

IF EMPLOYEE IS NOT ON FILE OR EMPLOYEE IS ON FILE AND CODE <=5 THEN "NOT OK"

On File	No	Yes	Yes
Code<=5	-	Yes	No
Action	Not OK	Not OK	???

This case has no information. Need to ask the customer on what to do

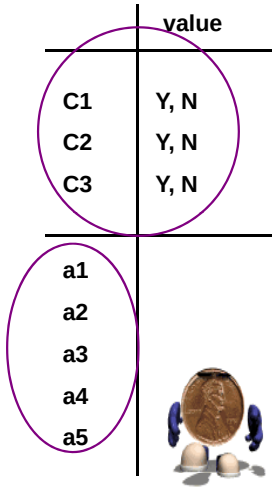
Decision Tables (3)

- The conditions in the DT may take on any number of values. When it is binary, then the DT conditions are like a **truth table set of conditions**.
- There is no particular order implied by the conditions and selected actions do not occur in any particular order.
- Rows specify all the conditions that the **input** may satisfy
- Columns specify different actions (or **output**) that may occur
- Entries in the table indicate whether the actions should be performed if a condition is satisfied.

Decision Tables (4)

- **Each column (or rule) suggests a significant test case** (conditions as inputs, and actions as outputs)
- DT lists the 'complete' combination of conditions. Thus, it produces a ***comprehensive set*** of test cases.
- DT allows us to look at and consider "dependence", "impossible" and "not relevant" situations and eliminate some test cases.
- Allow us to detect potential error in our specifications (e.g., missing cases).
- DT can represent an equivalence partitioning.

Creating a Decision Table (1)



1. The conditions and actions are defined and listed.
   - Write down the values of conditions
   - Group related conditions
   - Put the most dominating condition first
   - Put multi-valued conditions last
   - List all the actions
2. Calculate the number of possible combinations
   - ⌘ If all conditions are simply Y/N values: 2^{number}
   - ⌘ If 1 condition with 3 values and 3 with 2 values: $3^1 \times 3^2 = 24$

This give an idea about the number of columns (how big is our table)

Creating a Decision Table (2)

	R1	R2	R3	R4	R5	R6	R7	R8
C1	T	T	T	T	F	F	F	F
C2	T	T	F	F	T	T	F	F
C3	T	F	T	F	T	F	T	F
a1	x			x	x			x
a2	x							x
a3		x				x		
a4			x	x			x	x
a5	x			x				

3. All combinations of conditions that are formally possible are specified.
4. Identify which action should be produced in response to particular combinations of conditions.

Creating a Decision Table (3)

5. Reduce the table and check for ambiguous and incompleteness.

⌘ For indifferent combinations of condition – use "Don't care" (e.g., C1 = Y, C1 = N produce the same result, then use C1 = don't care)

⌘ Join columns where actions are identical

⌘ Check that we have covered all combinations?

For complex decision table, construct a hierarchy of decision tables

Reference: http://www.csm.uwe.ac.uk/~jharney/table.html

Another Decision Table Example: Credit Card Billing

⌘ Conditions
- **Account balance >=0**
- **Missed payment in last year**
- **Previous payment late**
- **Signed up for bonus rebate plan**

⌘ Actions
- **Charge interest (regular or high)**
- **Charge late payment fee**
- **Calculate cash back (i.e., rebate)**

Example: Credit Card Billing

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7
Conditions							
Balance	>=0	<0	<0	<0	<0	<0	<0
Missed payment	DC	Yes	Yes	No	No	No	No
Late last payment	DC	No	Yes	No	No	Yes	Yes
Bonus plan	DC	DC	DC	No	Yes	No	Yes
Actions							
Interest	None	High	High	Regular	Regular	Regular	Regular
Late fee	No	No	Yes	No	No	Yes	Yes
Rebate	No	No	No	No	Yes	No	Yes

DC: don't care (any value)

Test case

	Conditions				*Expected Results*		
Test Cases	Balance	Missed Payment	Late last payment	Bonus plan	Interest	Late fee	Rebate
...							
TC 2-1	-500	Yes	No	No	High	No	No
TC 2-2							
TC 3-1				**			
TC 3-2							

Decision table testing summary

#tests: moderate to large (depends on granularity of conditions/actions)

Usage: much study of requirement is needed

When to use:

- Assumes dependent inputs (variables)
- Applications with:
  - prominent If-then-else logic
  - logical relationships among input variables
  - calculations involving subsets of the input variables (possible computational errors)
  - cause and effect relationships between input/output
  - high cyclomatic complexity

Disadvantages:

- We can apply DT testing only if we can express the functional spec. in term of a decision table.
- Need a few iterations to identify all conditions.
- Checking completeness, redundancy, and consistency requires a great effort
- DT doesn't scale up very well (not suitable for complicate systems) – 2^n for n conditions!!

Cause-Effect Graphing 因果图

Cause-effect graphing looks at all possible ways that inputs can combine to produce outputs.

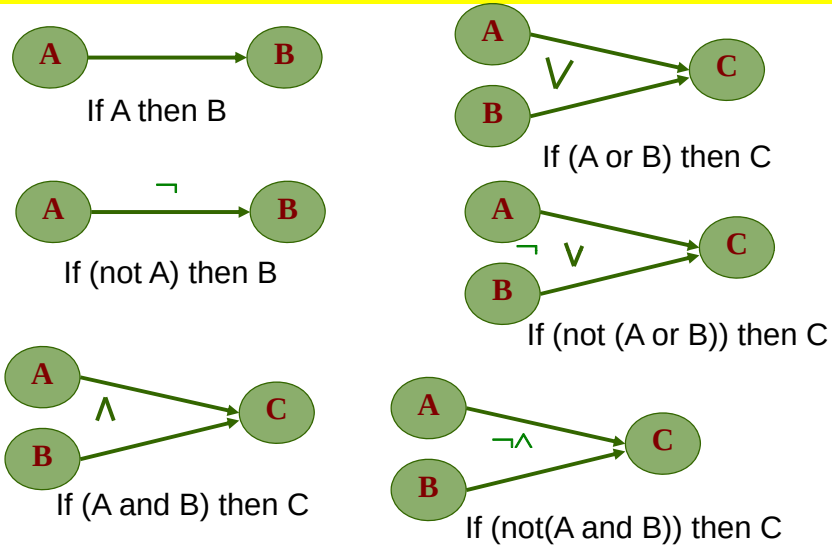
It aids in selecting test cases to check if the program will produce right effect for every possible combination of causes.

- A cause-effect graph is a directed graph that describes the logical combinations of **causes** and their relationship to the **effects** to be produced.
- Cause is like **condition**, and effect is like **action** in the Decision Table!

Method of Cause-Effect Graphing

1. Divide the program specification into pieces of workable size.
2. Identify causes (input conditions) and effects (actions or computation) in the specification.
3. Analyze the specification to determine the *logical relationship* among causes and effects, and express it as a cause-effect graph. *(Draw causes on the LHS, Draw effects on the RHS, Draw logical relationship between causes and effects as edges in the graph.)*
4. Identify syntactic or environmental constraints that make certain combinations impossible.
5. *Translate the graph into a decision table.*
6. *Select a test case for every column in the decision table.*

Drawing Cause-Effect Graphs



Cause-effect graphing example

Specification: Check-debit function

Inputs: new balance and account type (2 types: postal or counter)

Output: one of 4 possible actions:

- Process debit and send out letter.
- Process debit only.
- Suspend account and send out letter.
- Send out letter only.

The Check-debit function has the following requirements:

- If there are sufficient funds available in the account to be in credit, or the new balance would be within the authorized overdraft limit, then process the debit.
- If new balance is below the authorized overdraft limit, do not process the debit, and if the account type is postal, suspend the account.
- If a) the transaction has an account type of postal or
  b) the account type is counter and there are insufficient funds available in the account to be in credit, send out letter.

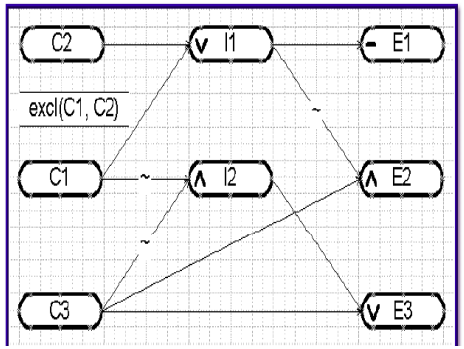
Cause-effect graphing example

The *causes* for the function are:

C1 - New balance is in credit.
C2 - New balance is in overdraft, but within the authorized overdraft limit.
C3 - Account type is postal.

The *effects* are :

E1 - Process the debit.
E2 - Suspend the account.
E3 - Send out letter.



C1 and C2 cannot be true at the same time.

Converting CE Graph -> DT

Each column of the decision table is a rule.

In the top part of the table, each rule is tabulated against the causes.

- A T indicates that the cause must be TRUE for the rule to apply
- An F indicates FALSE for the rule to apply.

In the bottom part, each rule is tabulated against the effects.

- A T indicates that the effect will be performed;
- An F indicates that the effect will not be performed;
- An asterisk (*) indicates that the combination of conditions is **infeasible** and so no effects are defined.

Decision Table

Rules	1	2	3	4	5	6	7	8
C1: New balance is in credit.	F	F	F	T	T	F	T	T
C2: New balance is in overdraft, but within the authorized limit.	F	F	T	F	F	T	T	T
C3: Account is postal.	F	T	F	F	T	T	F	T
E1: Process the debit.	F	F	T	T	T	T	*	*
E2: Suspend the account.	F	T	F	F	F	F	*	*
E3: Send out letter.	T	T	T	F	T	T	*	*

T: true ***F: false*** ****: infeasible***

- Only test cases 1-5 are required to provide 100% functional coverage.
- Rule No. 6 does not provide any new functional coverage that has not already been provided by the other 5 rules, so a test case is not required for No. 6.
- No test cases are generated for rule Nos. 7 and 8 because they describe infeasible conditions since C1 and C2 cannot be true at the same time.

Test Cases

Introduction
Spec-independent
Spec-dependent
FSM
DT
CE
Other

Final set of 5 test cases with sample data values :

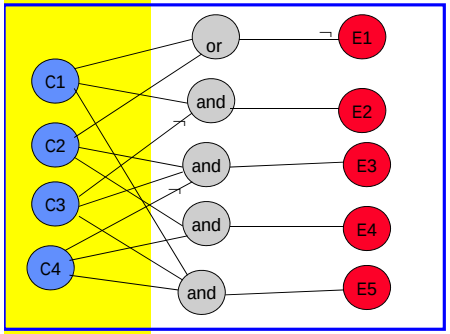
	CAUSES						EFFECTS
Test Case	*Current Balance*	*Debit Amount*	*Difference*	*Overdraft Limit*	*New Balance*	*Account Type*	*Action*
1	-$70	$50	-$120	-$100	-$70	Counter	**Send out letter.**
2	$420	$2,000	-$1,580	-$1,500	$420	Postal	**Suspend the account; send out letter.**
3	$650	$800	-$150	-$250	-$150	Counter	**Process the debit; send out letter.**
4	$2,100	$1,200	$900	-$1,000	$900	Counter	**Process the debit.**
5	$250	$150	$100	-$500	$100	Postal	**Process the debit; send out letter.**

Another CE Example: ATM

Introduction
Spec-independent
Spec-dependent
FSM
DT
CE
Other

- For a simple ATM banking transaction system
  - Causes (inputs)
    - C1: Command is credit
    - C2: command is debit
    - C3: account number is valid
    - C4: transaction amount is valid
  - Effects
    - E1: Print “invalid command”
    - E2: Print “ invalid account number”
    - E3: Print “debit amount not valid”
    - E4: debit account
    - E5: Credit account

Example: ATM Cause Effect Graph

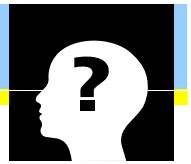


Don’t Care condition

Decision table

Cause\ effect	R1	R2	R3	R4	R5
C1	F	T	DC	DC	T
C2	F	DC	T	T	DC
C3	DC	F	T	T	T
C4	DC	DC	F	T	T
E1	T	F	F	F	F
E2	F	T	F	F	F
E3	F	F	T	F	F
E4	F	F	F	T	F
E5	F	F	F	F	T

Review Questions



1 Can you apply FSM testing to your current project?
2 Can you always find the boundary of the input domain?
3 You can find EC from the specification. Where else?
4 Compare Decision Table and Equivalent Partitioning testing.

Guidelines on black box testing

- Inputs dependent on each other?
  →decision table
- Suspect computational errors?
  →decision table or equivalence partition
- Suspect boundary errors?
  →boundary value testing

Scenario

⌘ A **scenario** is a description of a possible set of interactions between a system and a set of actors external to the system.
⌘ **Purpose:** to stimulate thinking about possible occurrences, assumptions relating these occurrences, possible opportunities and risks, and courses of action.
⌘ Scenarios can be used:
  ➔ to discover user needs that are not obvious
  ➔ to better describe the use of systems in real-life
  ➔ to systematically explore normal-case and exceptional behavior of a system and its environment.

Example Scenario of ATM

A typical scenario:
1 Insert bank card
2 Enter password
3 Make selection (one of the following:)
  - Withdraw [enter x$, take bank card, take x$, take receipt]
  - Transfer
  - Deposit
  - Look up balance
4 Finish transaction

A good source of scenario test is **Use Case**.

Use Cases

A use case is a sequence of actions performed by a system, which produce a result of value to a user.

Use cases are a type of process flow model.

The value of use cases is that they focus attention on the user, rather than on the actions the system performs.

Many applications – not just OO software – employ use uses to document system requirements.

The use cases serve as vehicle for transforming the user interface description into test cases.

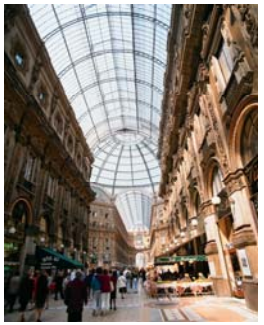
Each use case generates many test cases:

- The main success scenario gives the first test.
- Have at least one test for each alternative scenario.
- Different testers may add additional test cases from the same use case.

Example: Scenario Testing

A **Mall Guide system** in a shopping mall has the following stores:

- Women's clothes (6 stores)
- Men's clothes (2 stores)
- Children's clothes (4 stores)
- Women's shoes (3 stores)
- Men's shoes (3 stores)
- Flowers (2 store)
- Gifts (5 stores)
- Restaurants (8 stores)



Two types of actors interact with the Mall System:

- **Shoppers** who seek to find an appropriate store
- **Administrators** who update store names & locations.

User input

Shopper commands: main menu

1. Shopper touches a store category from a list and is given a list of stores from which to select.
2. Shopper touches map key and a map of the mall is provided on the screen.
3. Shopper touches help key and is provided with a help screen.

Shopper commands: list of stores

4. Shopper touches a particular store name and a map highlighting the store's location is provided on the screen.
5. Shopper touches main menu key and returns to the main menu.
6. Shopper touches help key and is provided with a help screen.

Shopper commands: maps

7. Shopper touches print key and a map is printed out.
8. Shopper touches "return to previous screen" key and the prior screen is displayed.
9. Shopper touches main menu key and returns to the main menu.
10. Shopper touches help key and is provided with a help screen.

Shopper commands: help

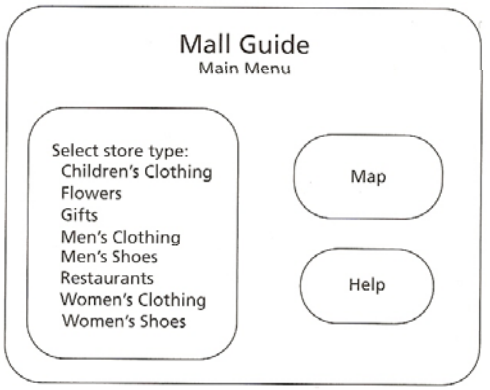
11. Shopper touches "return to previous screen" key and the prior screen is displayed.
12. Shopper touches main menu key and returns to the main menu.

Administrator commands

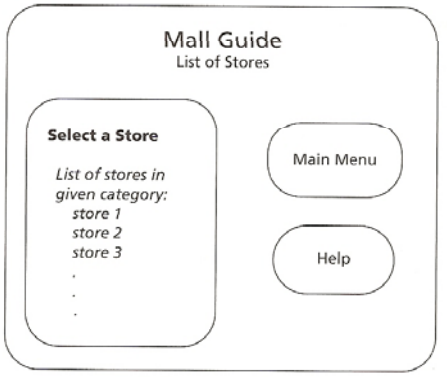
13. Administrator adds a new store name.
14. Administrator removes a store name.
15. Administrator modifies the mall map.

(will not look at these commands in detail)

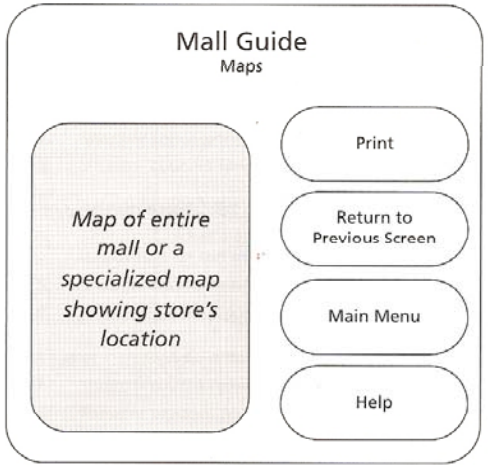
Main menu touch screen



Store list touch screen



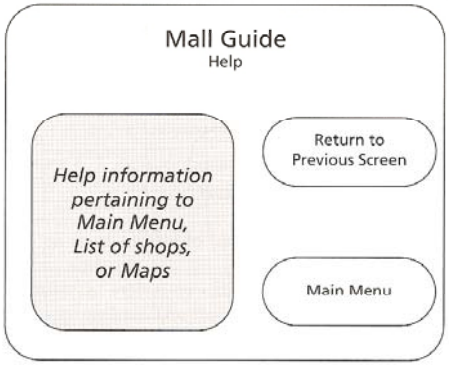
Map touch screen



User can request a printed map.

Help touch screen

Any screen allows the shopper to select the help feature.



Applying Use Cases

- We will consider 3 use cases.
- Each use case generates many test cases.

The main success **scenario**, which is the simplest and most common path through the use case, provides the information for the first use case and serves as the foundation for subsequent test cases.

At least one test case for each extension to ensure that the system properly handles that event.

Use case 1

Use case #1 Shopper wants to find a specific store in the mall.

Success guarantees

Shopper receives a printed map to the desired store.

Main success scenario

1. Shopper selects a store category.
2. Shopper selects a store name.
3. Shopper selects the print function.

Extensions

1a. Shopper wants help selecting a store category.
1b. Appropriate store category does not exist.
2a. Shopper wants help selecting a store name.
2b. Desired store does not exist.

Use case 2

Use case #2 Shopper wants to find all gift stores and map the best route to each store.

Success guarantees

Shopper receives a printed map for each gift store.

Main success scenario

1. Shopper selects a store category.
2. Shopper selects the first store name.
3. Shopper selects the print function.
4. Shopper returns to the list of stores.
5. Shopper selects the second store name.
6. Shopper selects the print function.

Extensions

(none)

Use case 3

Use case #3 Administrator wants to add a new store to the Mall Guide.

Success guarantees

New store name appears under List of stores.
New store location appears on the mall map.

Main success scenario

1. Administrator selects the store category.
2. Administrator adds the store name.
3. Administrator updates the map.

Extensions

1a. Appropriate category does not exist.
2a. Store name already exists.
3a. Location selected already assigned to another store.

Test case ID	Test source	Initial system state	Input	Expected results	Actual results
u1-1	use case #1	Start test from the main menu.	(a) select "Men's clothes" (b) select the first store name from the list. (c) hit print key	(a) The list of stores screen appears and lists 2 men's clothes stores. (b) Get a map to this store posted on the screen. (c) Get a printed copy of the map.	
u1-2	use case #1 extension 1a	Start test from the main menu.	(a) hit help key (b) hit "return to previous screen" key	(a) Get the help screen providing information about the main menu. (b) Return to main menu.	
u1-3	use case #1 extension 1b	Start test from the main menu.	[No input: Shopper wants to find a book store].	Main menu does not list books as a store category.	
u1-4	use case #1 extension 2a	Start test from the main menu.	(a) select "Restaurants" (b) hit help key	(a) The list of stores screen appears and lists 5 restaurants. (b) Get the help screen providing information about the list of stores menu.	
u1-5	use case #1 extension 2b	"Lorna's Shoes" is not a valid store name. Start test from the main menu.	Select "Women's shoes"	The list of stores screen appears and lists 2 women's shoes stores, but "Lorna's Shoes" is not included.	
u2-1	use case #2	Gifts stores listed are "Collectibles" and "The Gift Box". Start test from the main menu.	1. select "Gifts" 2. select "Collectibles" 3. hit print key 4. hit "return to previous screen" key 5. select "The Gift Box" 6. hit print key	1. The list of stores screen appears and lists 2 gift stores. 2. Get a map showing the location of "Collectibles". 3. Get a printed copy of the map. 4. Return to list of stores screen listing 2 gift stores. 5. Get a map showing the location of "The Gift Box". 6. Get a printed copy of this map.	

More Test cases

Introduction

Spec-independent

Spec-dependent

Other

▸ **Scenario**
Checklist
Risk-based

Test case ID	Test source	Initial system state	Input	Expected results	Actual results
u3-1	use case #3	Start test from the administrator menu.	1. select "Men's clothes" 2. type in "The Cave" 3. enter location of store onto map 4. run test #u1-1 to verify addition of new store	1. List containing 2 store names appears. 2. System accepts the store name. 3. Map updated. 4. Get list of 3 stores, including new name; get map showing new store location.	
u3-2	use case #3 extension 1a	Start test from the administrator menu.	1. create new store category "Books" 2. add new store "The Reading Room" to the books category 3. enter location of store onto map 4. use Shopper interface to confirm addition of new store	1. New category created. 2. System accepts the store name. 3. Map updated. 4. New book category contains the new store name; get map showing new store location.	
u3-3	use case #3 extension 2a	Gifts stores listed are "Collectibles" and "The Gift Box". Start test from the administrator menu.	1. select the Gifts category 2. type in "Collectibles"	1. List containing 2 store names appears. 2. Get error: name already exists.	
u3-4	use case #3 extension 3a	"Red Roses" is the one flower store listed. Start test from the administrator menu.	1. select "Flowers" 2. type in "Flower Shoppe" 3. specify the location currently occupied by "Red Roses"	1. List containing 1 store name appears. 2. System accepts the store name. 3. Get error: location already occupied.	

Cross-reference matrix: Check the test cases

Input option \ Test case	u1-1	u1-2	u1-3	u1-4	u1-5	u2-1	u3-1	u3-2	u3-3	u3-4
1	✓			✓	✓	✓	✓	✓		
2										
3		✓								
4	✓					✓	✓	✓		
5										
6				✓						
7	✓					✓	✓			
8						✓				
9										
10										
11		✓								
12										
13							✓	✓	✓	✓
14										
15							✓	✓		✓

Did the test cases exercise every user input option?

- 4 use cases execute input option 13 (add a new store name), but no distinction exists between successful addition, missing store category, or invalid store name.

The matrix detects the need for additional test cases.

Introduction

Spec-independent

Spec-dependent

Other

▸ **Scenario**
Checklist
Risk-based

What other test cases we should use to test this system?

Hint: White-box and black-box techniques

Summary: Scenario Testing

- ➔ *User-centered view:* Concentrates on **what the customer does**, not what the product does (consists of many transactions)
- ➔ Scenario tests tend to exercise multiple subsystems in a single test (exactly that's what users do). **We should test each feature in isolation before testing scenarios**
- ➔ More complex and more realistic than tests from other techniques
- ➔ Will cover the higher visibility interaction defects, and expose coding errors because scenario tests combine many features and much data.
- These test cases are good for Acceptance Tests

Good Scenario Tests

- ➔ **Realistic.** A real user would attempt this scenario. We check this from use case analysis, monitoring of actual use of the system, or discussion from customers.
- ➔ **Complex.** It combines several features and uses them in a way that seems to challenge the system. It may involve ***a complex set of data***.
- ➔ **Easy to determine pass/fail.** If it takes a significant effort to determine the pass/fail, we will take shortcuts and guess the system is OK.
- ➔ **At least one stakeholder considers it a serious failure** if the scenario does not work.

Checklist (1)

- ⌘ A manual technique
- ⌘ Well-constructed checklists lead the tester to areas which likely have defects.
- ⌘ Can be simple yes-no questions, or more complex probing questions.
- ⌘ Effective at a very minimal cost.
- ⌘ Require minimal skills to use checklists
- For projects of low importance, it's okay to use checklists.
- For projects of high importance, highly formalized test cases are a must.

- Checklists are simple test cases that assume that the test case executor knows the application well, or can figure out how the application should work.

Example Checklist: GUI Testing

Checking Navigation:

1. Can the screen be accessed correctly from the menu?
2. Can the screen be accessed correctly from the toolbar?
3. Can the screen be accessed correctly by double clicking on a list control on the previous screen?
4. Can all screens accessible via buttons on this screen be accessed correctly?
5. Can all screens accessible by double clicking on a list control be accessed correctly?

Reference: http://members.tripod.com/~bazman/checklist.html

Checking Usability:

1. Are all the dropdowns on this screen sorted correctly? Alphabetic sorting is the default unless otherwise specified.
2. Is all date entry required in the correct format?
3. Have all pushbuttons on the screen been given appropriate Shortcut keys?
4. Do the Shortcut keys work correctly?
5. Have the menu options which apply to your screen got fast keys associated and should they have?

…

Checklist (2)

Advantages:

- ☺ provides an extensive set of probing questions
- ☺ limits the scope of the test
- ☺ generates consistent test from application to application
- ☺ uses to document the test results

Disadvantages:

- ☹ Test team may rely too much on the checklist and fail to test other areas
- ☹ the checklist may not be appropriate for the system being tested.

Risk-Based Testing

We cannot test everything.

We need to make decisions about where to focus the depth and intensity of testing.

We know from experience that

the top 10% - 15% of the test cases uncover 75-90% of the significant defects.

How to select these test cases?

Focus on what's important

- Core functionality – the parts that are critical or popular – before looking at the "nice to have" feature.
- Risky areas.

Product Risks depend on

- ⌘ Which functions and attributes are critical (for the success of the product)?
- ⌘ How visible is a problem in a function? (for customers, users, outside people)
- ⌘ How often is a function used?

Risk-Based Testing

What is the minimum testing effort that we should invest to maximize risk reduction?

Testing is a mean to reduce risks related to software.

We want to use our limited time and resources to test the most important things.

Objective: find the most important defects as early as possible at the lowest price

A **risk** is an unwanted event that has negative impact.

A **problem** is something that is a fact (happened or will happen)

A risk is something that **might happen** in the future.

Risk-Based Testing

The essence of risk-based testing:

- Imagine how the product could fail
- Design tests to expose these risks (potential failures).

⌘ In risk-based testing, we create tests for vulnerable areas of the program.

Like error-based or fault-based testing

Risky Areas:

- Customer or user impact
- Business operational risk
- Prior defect history
- What's new/modified
- Heavily used features
- Complex features
- Exceptions
- Poorly built portions

Ref. Cem Kaner & James Bach's notes

Risk-Based Test Management

⌘ List all areas of the program that could require testing

⌘ On a scale of 1-5, assign a probability-of-failure estimate to each

⌘ On a scale of 1-5, assign a severity-of-failure estimate to each

Estimated Risk = estimated probability x estimated severity

⌘ Test the program areas in order of the estimated risk

Example

Function	Prob.	Severity	Risk
1	4	3	**12**
2	2	5	10
3	3	1	3

Component risk matrix

Component	Risk	Categories
Printing	Normal	popular,
Report generation	Higher	new, strategic, critical
Installation	lower	popular, changes
Library	Lower	complex

Higher risk items get 2x the effort as normal items, which in turn get 2x the effort as the components that are lower risk.

2 Heuristic for Risk Based Testing

1. Inside-out approach

- Begin with situation details
- Then perform risk identification

- Study the system and ask "what can go wrong here?" with each component:
  - **Vulnerabilities**: what weaknesses or possible failures are there in this component?
  - **Threats**: what inputs or situations could there be that might exploit a vulnerability and trigger a failure in this component?
  - **Victims**: who or what would be impacted by potential failures and how bad would that be?

*These are heuristics because they are fallible but useful guides. We have to exercise our own judgment about **which** to use **when**.*

2 Heuristic for Risk Based Testing

1. Inside-out approach

Ask:

- What if this function fail?
- What error checking do you do here?
- Is this a complete picture? What have you left out?
- How to do the integration testing?
- What worry you?
- What do you think I should test?
- Can this function be invoked at the wrong time?

2 Heuristic for Risk Based Testing

2. Outside-in approach

- Use a predefined risk list
- Check any of those risks apply to our project.
- React to risks in current circumstance
- A more general approach than inside-out and somewhat easier.

Generic Risk Lists

- Generic risks are risks that are universal to any system.

Examples:

- Complex: anything disproportionately large, intricate, or convoluted
  Example:
  - Adding a new feature/module that is highly complex
  - Modifying a new feature/module that is highly complex
  - Modifying a feature/module with a large number of lines of code
- New: anything that has no history in the product
- Changed: anything that has been changed or improved
- Critical: anything whose failure could cause substantial damage
- Popular: anything that will be used a lot
- Recent failure: anything with a recent history of failure

Project Risk : Where to Look for Errors

- New things: Less likely to have revealed its defects yet.
- New technology: same as new thing, plus the risks of unanticipated problems.
- Learning curve: People make more mistakes while learning.
- Changed things: same as new things, but changes can also break old code.
- Poor control: without Software Configuration Management, files can be overridden or lost.
- Late change: rushed decisions, rushed or demoralized staff lead to mistakes.
- Rushed work: some tasks or projects are chronically under funded and all aspects of work quality suffer.

- Publicity: anywhere failure will lead to bad publicity.
- Liability: anywhere that failure would justify a lawsuit.
- Critical: anything whose failure could cause substantial damage.
- Precise: anything that must meet its requirements exactly.
- Easy to misuse: anything that requires special care or training to use properly.
- Popular: anything that will be used a lot, or by a lot of people.
- Strategic: anything that has special importance to our business.
- VIP: anything used by particularly important people.

Risks associated with the project as a whole, or with the staff, or management of the project can guide our testing.

Project Risk : Where to Look for Errors

- **Fatigue: tired people make mistakes.**
- **Distributed team: a far flung team communicates less.**
- **Other staff issues: alcoholic, mother died, two programmers who won't talk to each other (neither will their code)…**
- **Surprise features: features not carefully planned may have unanticipated effects on other features.**
- **Third-party code: external components may be much less well understood than local code, and much harder to get fixed.**
- **unbudgeted: unbudgeted tasks may be done shoddily.**

- **Ambiguous: ambiguous descriptions lead to incorrect or conflicting implementations.**
- **Conflicting requirements: ambiguity often hides conflict.**
- **Mysterious silence: when something interesting or important is not described or documented, it may have not been thought through, or designer may be hiding problems.**
- **Unknown requirements: requirements surface throughput development. Failure to meet a legitimate requirement is a failure of quality for that stakeholder.**
- **Evolving requirements: people realize what they want as the product develops. Adhering to a start-of-the project requirements list will fail.**

Project Risk : Where to Look for Errors

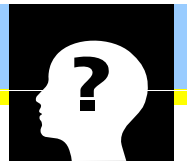
- **Buggy: anything known to have lots of problems has more.**
- **Recent failure: anything with a recent history of problems.**
- **Upstream dependency: may cause problems in the rest of the system.**
- **Downstream dependency: sensitive to problems in the rest of the system.**
- **Distributed: anything spread out in time or space, that must work as a unit.**
- **Open-ended: any function or data that appears unlimited.**
- **Complex: what's hard to understand is hard to get right.**
- **Language-typical errors: such as wild pointers in C.**

- **Little system testing: untested software will fail.**
- **Little unit testing: programmers normally find most of their bugs.**
- **Previous reliance on narrow testing strategies: can yield a many-version backlog of errors not exposed by those techniques.**
- **Weak test tools: if tools don't exist to help identify / isolate a class of error (e.g. wild pointers), the error is more likely to survive after testing.**
- **Unfixable bugs: when they were first reported, no one knew how to fix them in the time available.**
- **Untestable: anything that requires slow, difficult or inefficient testing is probably under tested.**

A strategy of Doing RBT

1. Identify risk ideas
2. For each important idea, *determine test activities, prepare tests (that have power against that idea), spend time* to gather information on it.
3. Maintain traceability between risks and tests.
4. Monitor and report the status of the risks as the project goes on and we learn more about them.
5. Assess coverage of the testing effort program, given a set of risk-based tests.
6. If a risk is determined to be small enough, stop testing against it.
7. On retesting an area, evaluate our test experiences so far to determine what risks they were testing for and whether more powerful variants can be created.
8. Do *some* non-risk-based testing too.
9. Build lists of defect histories, configuration problems, tech support requests and obvious customer confusions – enlarge our lists of failure modes.

Review Questions



Introduction

Spec-independent

Spec-dependent

Other

1 True or False?

a. Testing is about running the program until we get the right results.

b. Thorough testing is too expensive for most projects.

c. Testing ensures quality.

d. Error-free program is always the testing goal.

2 How to find scenarios? Give 5 methods.

3 What is the key objective of black-box testing?

4 Can you tell whether a test case is a black-box test or a white-box test?

Which Color is Better?

(White or Black)

Why do Both Code-based & Spec-based Testing?

Different developers may generate different, yet functionally equivalent, source code from the same set of low-level requirements.

Example: A low-level requirement to assign to x twice the input value y may be coded as

```
x = 2 *y
x = y + y
x = y/0.5
```

Structural coverage analysis is required to assure that the as-implemented code structure has been adequately tested and does not contain any unintended functionality.

- Structural testing happens late, because it is based on the implementation.

Why do Code-based & Spec-based Testing?

- There might be parts of code which are not fully exercised by spec-based tests.
  e.g., a Boolean expression (e.g., a true/false condition) in the specification may be implemented as many LOC.
- Code that is implemented without being linked to requirements may not be exercised by spec-based tests. Such code could result in unintended functions.
- Code-based testing can find some errors which have been missed by spec-based testing
  - ➔ misinterpretation of the specification
  - ➔ simple typographical mistakes
  - ➔ ignorance, incorrect assumptions

Why do Code-based & Spec-based Testing?

- Code-based testing is testing against the implementation (the program) and will discover **faults of commission**, indicating the faults in the implementation.
- It cannot guarantee that the complete specification has been implemented.
- It may suffer from inadequate specification coverage.

- Specification-based testing is testing against the specification and will discover faults of omission, indicating that part of the specification has not been implemented.
- It cannot guarantee that all parts of the implementation have been tested.
- It may suffer from inadequate code coverage.

Spec-based testing shows that all intended functions are properly implemented, and structural coverage analysis demonstrates that all existing code is reachable and adequately tested.

Together they provide a higher confidence that there are no defects.

Summary (1)

1. We need high quality specification to support better testing.
2. Some testing techniques focus on testing the input, while others focus on testing the functions.
3. EP testing selects one test case for each EC.
4. BV testing selects at least three test cases for each boundary.
5. Classification tree testing combine different equivalence classes together to form test cases
6. Output range testing is similar to input domain testing except that the focus is on the output range.
7. Special-value testing depends on the experience of the test team.
8. State machine testing requires a state machine model of the system. It aims to achieve transition coverage.

Summary (2)

9. DT is more sophisticated than EP and BV testing, as we need to consider data and logical dependencies.
10. Risk-based testing focus on the risk of the software.
11. White box testing should be performed in unit test, while black box testing tends to be applied during later stages (integration test, system test, UAT), but it can also be applied at unit test level.
12. Combinatorial testing makes an excellent trade-off between test effort and test effectiveness.
13. Pairwise testing can often reduce the number of tests dramatically, but it can still detect faults effectively.

The value of Test Case Design

In practice, the design of test cases, including the review of the requirements specification, can reveal many defects!

Summary (3)

	Unit Test	Int. Test	System Test	UAT
Done by	Developer	Tester	Tester	User
Test Environment	*Internal*		*Lab*	Actual envion.
Test Techniques	code-based & spec-based	code-based & spec-based	spec-based	spec-based

	Black box (functional)	White box (structural)
Dynamic	Decision table, BV Equivalence Partition OA FSM CE	Dataflow testing Domain testing Path-based testing Basis path testing Mutation analysis
Static	Specification proving	Code Walkthroughs Inspections Program proving Symbolic execution Anomaly analysis

When to use which test technique?

Test Techniques	Requir't	Design	Code & Unit Test	Int. Test	Sys. Test
Boundary-value analysis					
Control-flow analysis					
Corrective proof					
Coverage based					
Decision table					
Domain testing					
Data flow					
Equivalent partition					
Error-base					
Fault-base					
Inspection					
Mutation testing					
Symbolic execution					
State-base					
CT Scenarios testing					

References

- Sloane, Neil J. A. A Library of Orthogonal Arrays. Information Sciences Research Center, AT&T Shannon Labs. 9 Aug. 2001 http://www.research.att.com/~njas/oadir/
- Daich, G.T., New Spreadsheet tool helps determine minimal set of test parameter combinations, Crosstalk, August 2003.
- Berger, Bernie (2001) "The dangers of use cases employed as test cases," STAR West conference, San Jose, CA. www.testassured.com/docs/Dangers.htm.
- Kaner, C. (2003) An introduction to scenario testing, http://www.testingeducation.org/articles/scenario_intro_ver4.pdf
- Stale Amland, Risk Based Testing, http://www.amland.no/WordDocuments/EuroSTAR99Paper.doc
- Whittaker, J. (2003), *How to Break Software*
- Bach, J. Risk based testing, STQEMagazine, Vol 1, No. 6, www.stqemagazine.com/featured.asp?stamp=1129125440
- http://en.wikipedia.org/wiki/Model-based_testing
- Combinatorial testing, http://csrc.nist.gov/acts

Murphy's Laws of Computing

- *When computing, whatever happens, behave as though you meant it to happen.*
- *The first place to look for information is in the section of the manual where you least expect to find it.*
- *When the going gets tough, upgrade.*
- *For every action, there is an equal and opposite malfunction.*
- *He who laughs last probably made a back-up.*
- *A complex system that does not work is invariably found to have evolved from a simpler system that worked just fine.*
- *A computer program will always do what you tell it to do, but rarely what you want to do.*