

# Test Design Techniques

SE401: Software Quality Assurance and Testing

# Outline

- The test development process
- Categories of test design techniques
- Case Study – Knight Capital
- Specification based testing (Black box)
- Black Box Testing Techniques
- General Testing
- Experience based testing

# Background

- Before we start testing, we need to know
  - What are we trying to test?
  - What are the inputs?
  - What are the results that should be produced by those inputs?
  - How do we prepare the tests?
  - How do we run the tests?
- To answer these questions we will look at
  - Test conditions
  - Test cases
  - Test procedures

# Background

- The test design process can be done in different ways, from very informal (little or no documentation), to very formal.
- The level of formality depends on the context of the testing, including:



# Test development process

1. Test analysis
2. Test design
3. Test implementation

# 1. Test analysis

- **The test basis documentation** is analyzed in order to determine *what* to test, i.e. to *identify the test conditions*.
- Test condition (*Def.*) = an item or event that could be verified by one or more test cases
- **Examples of test conditions**
  - A function
  - A transaction
  - A quality characteristic
  - Other structural elements (menus in web pages, etc.)

# Test possibilities

- **“Throw a wide net!”**
- First; identify as many test conditions as possible
- Second; select which one to develop in more detail
- We can't test everything (P2). We have to select a subset of all possible tests, but this subset must have a high probability of finding most of the defects in the system.
- We need a suitable test design technique to guide our selection and to prioritize the test conditions.

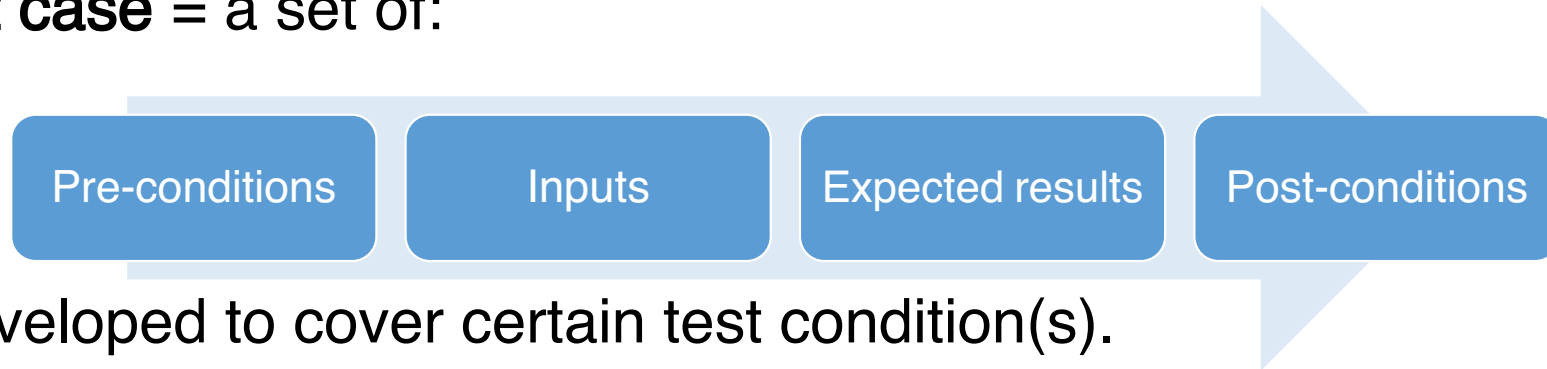
# 2. Test design

- During test design:



are created and specified.

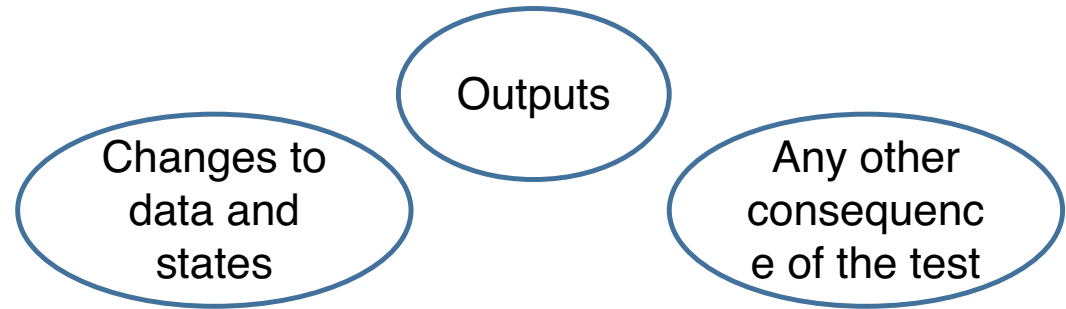
- **Test case** = a set of:



developed to cover certain test condition(s).

# Test oracle

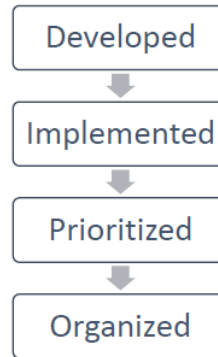
- In order to know what the system should do, we need to have a source of information about the correct behavior of the system an oracle
- Expected results include:



- If expected results have not been defined, then a plausible but erroneous result may be interpreted as the correct one
- Expected results should ideally be defined prior to test execution

# 3. Test implementation

- During test implementation the test cases are organized in the test procedures



- A manual test procedure
  - Specifies the sequence of action to be taken for executing of a test.
- An automated test procedure (test script)
  - If tests are run using a test execution tool , the sequence of action is specified in a test script

# 3. Test implementation

- The test execution schedule defines:
  - The order of execution of the test procedures & (possibly) automated test scripts
  - When they will be executed.
  - By whom to be executed.
- The test execution schedule will take into account such factors as:
  - risks
  - regression tests
  - prioritization
  - technical and logical dependencies

# 3. Test implementation

- Writing the test procedure is another opportunity to prioritize the tests, to ensure that the best testing is done in the time available.
- A good **rule of thumb** is ‘Find the scary stuff first’. However the definition of what is ‘scary’ depends on the business, system or project and on the risks of the project.

# Case Study – Knight Capital

## High Frequency Trading (HFT)

# Case Study – Knight Capital: High Frequency Trading (HFT)

- The ***Knight Capital Group*** is an American global financial services firm.
- Its high-frequency trading algorithms Knight was the largest trader in U.S. equities
  - with a market share of 17.3% on NYSE and 16.9% on NASDAQ.

# Case Study – Knight Capital

- Aug 1, 2012.
  - In the first 45 minutes, Knight Capital's computers executed a series of unusually large automatic orders.
    - “... spit out duplicate buy and sell orders, jamming the market with high volumes of trades that caused the wild swings in stock prices.”
- By the end of day: \$460 million loss
  - “Trading Program Ran Amok, With No ‘Off’ Switch”
- In two days, the company's market value plunged by 75%



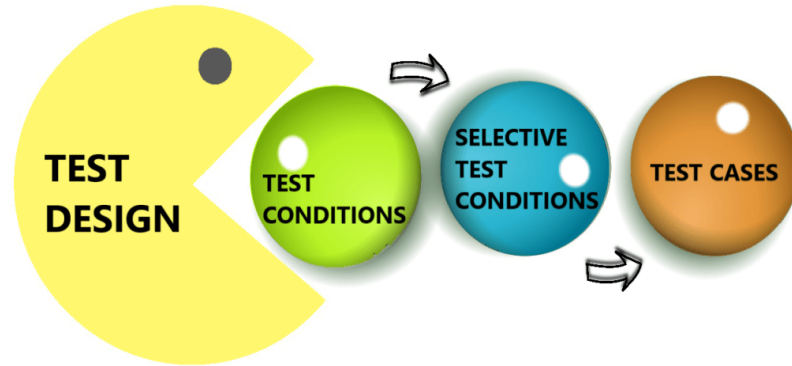
# Case Study – Knight Capital: What Happened?

## *"Zombie Software" Blamed for Knight Capital Trading Snafu*

- A new algorithmic trading program had just been installed, and began operation on Aug 1.
- A dormant legacy program was somehow "inadvertently reactivated"
- Once activated, the dormant system started multiplying stock trades by one thousand
  - Sent 4 million orders when attempting to fill just 212 customer orders
- “Knight’s staff looked through *eight* sets of software before determining what happened.”

# Case Study – Knight Capital: The Investigation and Findings

- SEC launched an investigation in Nov 2012. Findings:
  - Code changes in 2005 introduced defects. Although the defective function was *not meant to be used*, it was kept in.
  - New code deployed in late July 2012. The defective function was triggered under new rules. Unable to recognize when orders have been filled.
  - Ignored system generated warning emails.
  - Inadequate controls and procedures for code deployment and testing.
- Charges filed in Oct 2013
  - Knight Capital settled charges for \$12 million



# Testing Techniques

# Testing techniques

Recall the difference between static and dynamic test techniques:

**Static** testing



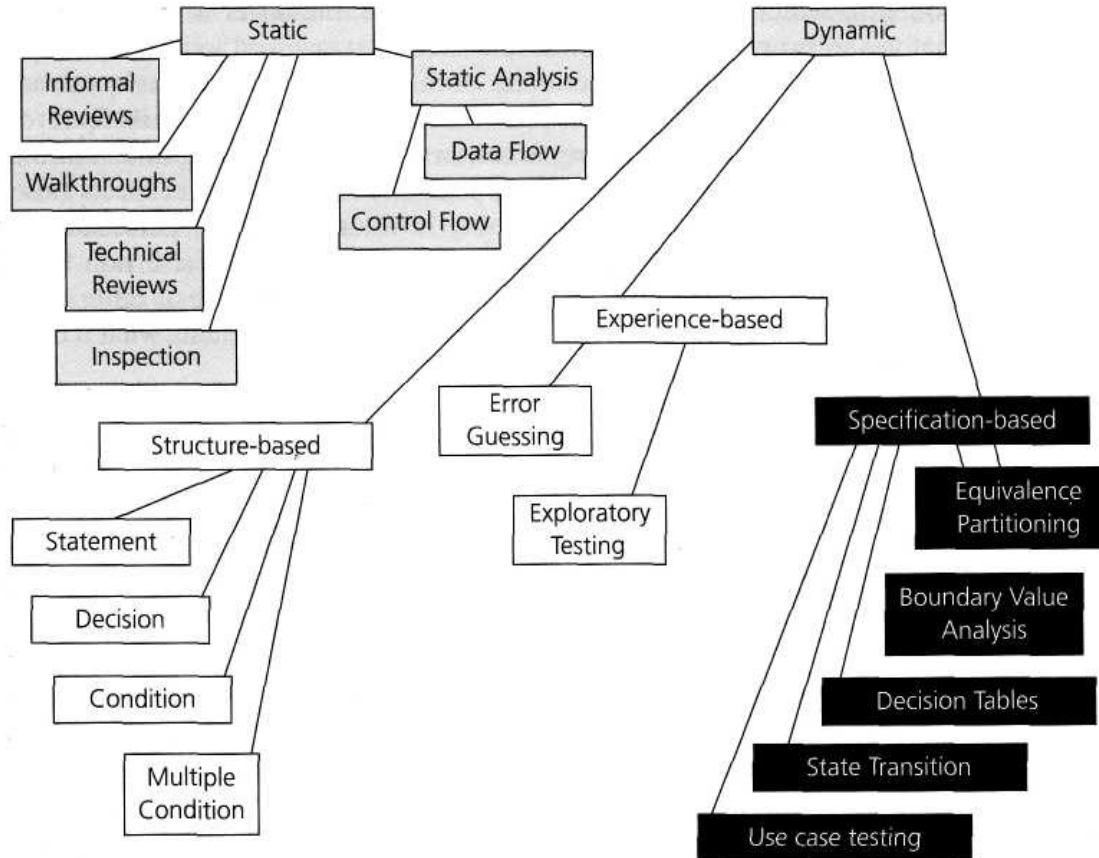
- Manual examination and automated analysis of **code** **without executing** it

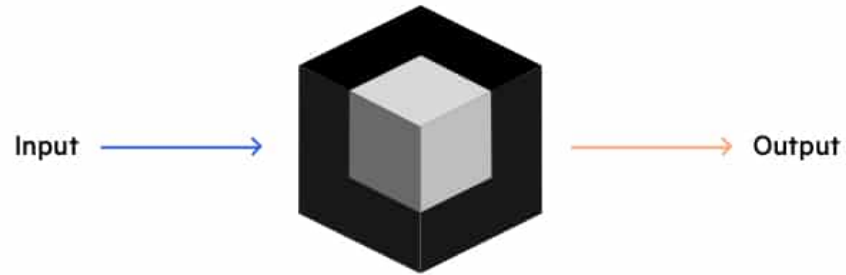
**Dynamic** testing



**code** to be **executed**

# Categories of test design techniques





# Specification-Based Testing

## Black Box Testing

# Functional Testing: A.k.a.: Black Box Testing

- Derive test cases from the *functional* specifications
  - *functional* refers to the source of information
  - not to what is tested
- *Also known as:*
  - specification-based testing (from specifications)
  - black-box testing (no view of the code)
- Functional specification = description of intended program behavior
  - either formal or informal

# Common features of black box techniques

- Derive systematically test cases from specification
- Test cases are derived from how the software is constructed for example: code and design.
- Using existing test cases, we can measure the test coverage of the software
- Further test cases can be derived systematically to increase the test coverage
- The test cases are derived from the knowledge and experience of people:
  - Knowledge of testers, developers, users and other stakeholders about the software, its usage and its environment
  - Knowledge about likely defects and their distribution

# Black Box Testing

- Testing software against a specification of its external behavior without knowledge of internal implementation details
  - Can be applied to software “units” (e.g., classes) or to entire programs
  - External behavior is defined in API docs, Functional specs, Requirements specs, etc.
- Because black box testing purposely disregards the program's control structure, attention is focused primarily on the information domain (i.e., data that goes in, data that comes out)
- The Goal: Derive sets of input conditions (test cases) that fully exercise the external functionality

# Black-box Testing Errors Categories

- Incorrect or missing functions
- Interface errors
  - Usability problems
  - Concurrency and timing errors
- Errors in data structures or external data base access
- Behavior or performance errors
- Initialization and termination errors
- Unlike white box testing, black box testing tends to be applied later in the development process

# Questions answered by Black-box Testing

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundary values of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

# The Information Domain: inputs and outputs

- **Inputs**

- **Individual input values**

- Try many different values for each individual input

- **Combinations of inputs**

- Individual inputs are not independent from each other
    - Programs process multiple input values together, not just one at a time
    - Try many different combinations of inputs in order to achieve good coverage of the input domain

- **Ordering and Timing of inputs**

- In addition to the particular combination of input values chosen, the ordering and timing of the inputs can also make a difference

# The Information Domain: inputs and outputs

- Defining the input domain
  - Boolean value
    - T or F
  - Numeric value in a particular range
    - $-99 \leq N \leq 99$
    - Integer, Floating point
    - Non-negative
  - One of a fixed set of enumerated values
    - {Jan, Feb, Mar, ...}
    - {Visa, MasterCard, Discover, ...}
  - Formatted strings
    - Phone numbers
    - File names
    - URLs
    - Credit card numbers
    - Regular expressions

# Why Black Box Testing?

- Early.
  - can start *before* code is written
- Effective.
  - find some classes of defects, e.g., missing logic
- Widely applicable
  - any description of program behavior as spec
  - at any level of granularity, from module to system testing.
- Economical
  - less expensive than structural (white box) testing

The base-line technique for designing test cases

# Early Black Box Testing

- Program code is not necessary
  - Only a description of intended behavior is needed
  - Even incomplete and informal specifications can be used
    - Although precise, complete specifications lead to better test suites
- Early test design has side benefits
  - Often reveals ambiguities and inconsistency in spec
  - Useful for assessing testability
    - And improving test schedule and budget by improving spec
  - Useful explanation of specification
    - or in the extreme case (as in XP), test cases are the spec

# Functional versus Structural: Classes of faults

- Different testing strategies (functional, structural, fault-based, model-based) are most effective for different classes of faults
- Functional testing is best for *missing logic* faults
  - A common problem: Some program logic was simply forgotten
  - Structural (code-based) testing will never focus on code that isn't there!

# Functional vs. Structural Test

- Functional test is applicable in testing at all granularity levels:
  - Unit test (from module interface spec)
  - Integration test (from API or subsystem spec)
  - System test (from system requirements spec)
  - Regression test (from system requirements + bug history)
- Structural test is applicable in testing relatively small parts of a system:
  - Unit test

# Steps: From specification to test cases

## 1. Decompose the specification

- If the specification is large, break it into *independently testable features* to be considered in testing

## 2. Select representatives

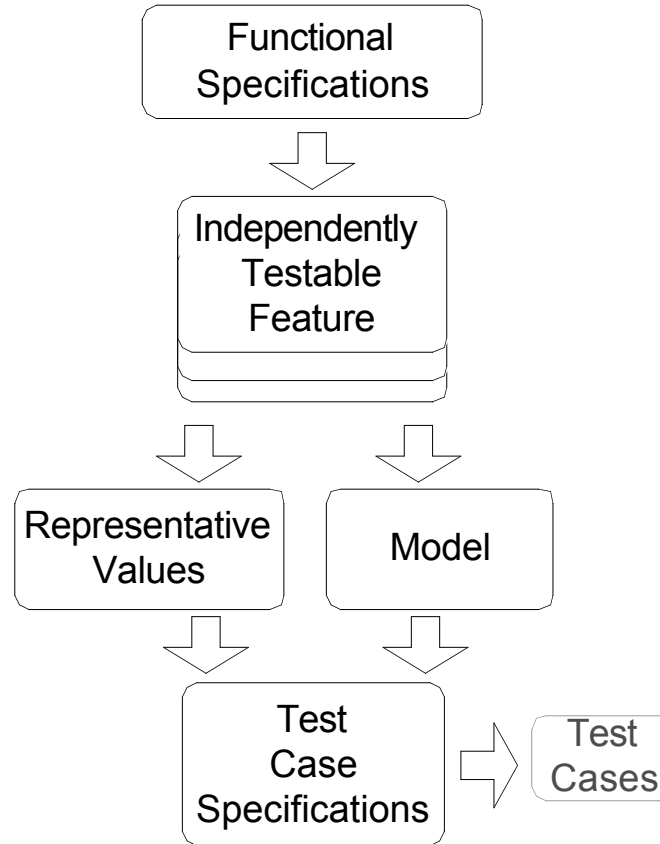
- Representative values of each input, or
- Representative behaviors of a *model*
  - Often simple input/output transformations don't describe a system. We use models in program specification, in program design, and in test design

## 3. Form test specifications

- Typically: combinations of input values, or model behaviors

## 4. Produce and execute actual tests

# From specification to test cases



# An Example: Postal Code Lookup



The screenshot shows the USPS website's ZIP Code Lookup interface. At the top left is the USPS logo. Below it is a cartoon mail carrier holding a letter and a sign that says 'U.S. MAIL'. To the right of the mail carrier is the text 'ZIP Code Lookup'. Below this are four buttons: 'Search By Address >>', 'Search By City >>', 'Search By Company >>', and 'Find'. Below the buttons is the instruction 'Find a list of cities that are in a ZIP Code.' followed by a section titled '\* Required Fields' which contains a single input field labeled '\* ZIP Code'. At the bottom of the form is a 'Submit >' button.

- Input: ZIP code (5-digit US Postal code)
- Output: List of cities

What are some representative values to test?

# Example: Representative Values

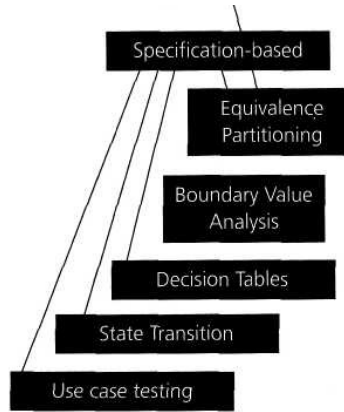
Simple example with  
one input, one output

- Correct zip code
  - With 0, 1, or many cities
- Malformed zip code
  - Empty; 1-4 characters; 6 characters; very long
  - Non-digit characters
  - Non-character data

Note prevalence of boundary values (0 cities, 6 characters) and error cases



The screenshot shows the United States Postal Service ZIP Code Lookup interface. At the top is the USPS logo. Below it is a cartoon mail carrier character holding a mailbag and a letter. The title "ZIP Code Lookup" is displayed. There are three search options: "Search By Address >>", "Search By City >>", and "Search By Company >>". A "Find" button is also present. Below the search options, the instruction "Find a list of cities that are in a ZIP Code." is shown. Underneath, there is a "Required Fields" section with a red asterisk and the label "ZIP Code" next to an empty text input field. A "Submit >" button is located at the bottom of the form.



# Black Box Testing Techniques

# Functional Testing Concepts

**Single Defect Assumption:** Failures are rarely the result of the simultaneous effects of two (or more) defects.

The four key concepts in functional testing are:

- Precisely identify the domain of each input and each output variable
- Select values from the data domain of each variable having important properties
- Consider combinations of special values from different input domains to design test cases
- Consider input values such that the program under test produces special values from the domains of the output variables

# Equivalence Classes

- *The basic idea is to divide a set of test conditions into sub groups or sub sets (partitions) that can be considered the same .*
- *It is important that the different partitions do not have common elements*
- *We need only to test one condition from each partition , because all the conditions in the same partition will be treated in the same way by the software.*

# Equivalence Partitioning

- A black-box testing method that divides the input domain of a program into classes of data from which test cases are derived
- An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed
- Test case design is based on an evaluation of equivalence classes for an input condition
- An equivalence class represents a set of valid or invalid states for input conditions
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercised at once

# Equivalence Partitioning

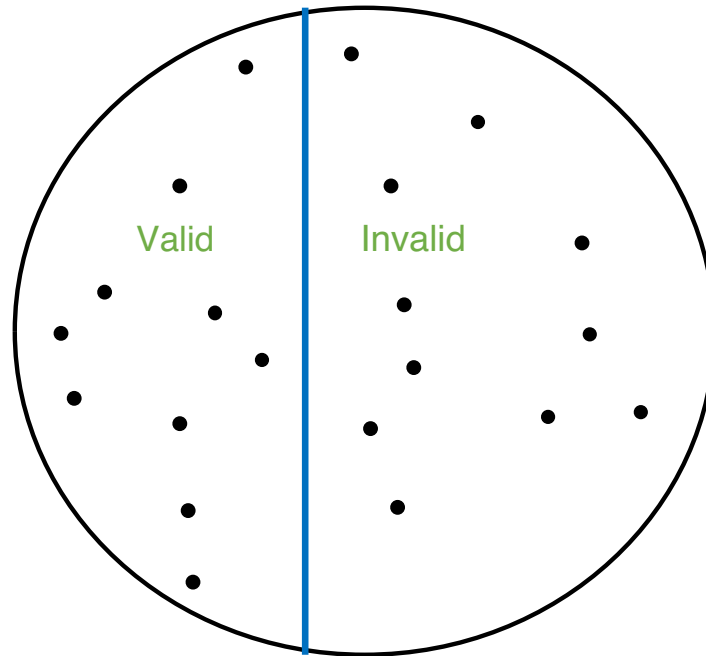
- Typically the universe of all possible test cases is so large that you cannot try them all
- You have to select a relatively small number of test cases to actually run
- Which test cases should you choose?
- Equivalence partitioning helps answer this question

# Equivalence Partitioning

- Partition the test cases into "equivalence classes"
- Each equivalence class contains a set of "equivalent" test cases
- Two test cases are considered to be equivalent if we expect the program to process them both in the same way (i.e., follow the same path through the code)
- If you expect the program to process two test cases in the same way, only test one of them, thus reducing the number of test cases you have to run

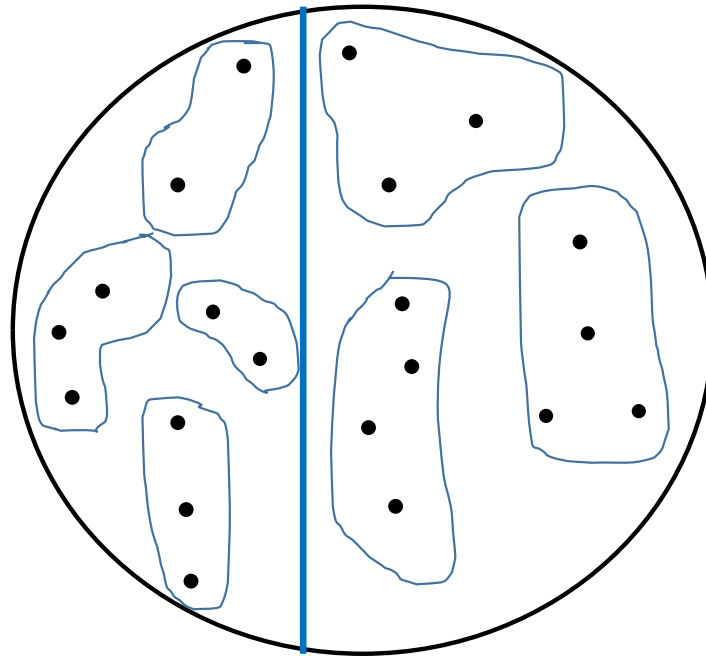
# Equivalence Partitioning

- First-level partitioning: Valid vs. Invalid test cases



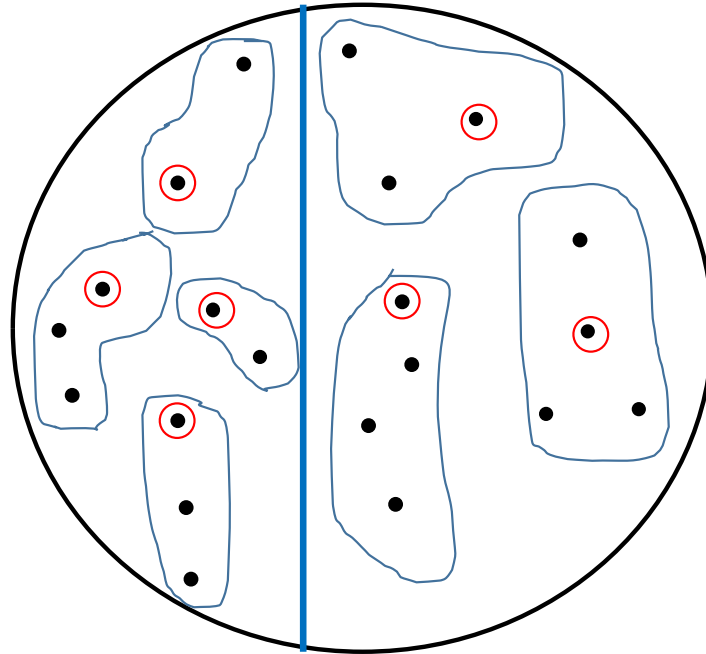
# Equivalence Partitioning

- Partition valid and invalid test cases into equivalence classes



# Equivalence Partitioning

- Create a test case for at least one value from each equivalence class



# Equivalence Partitioning

- When designing test cases, you may use different definitions of “equivalence”, each of which will partition the test case space differently
  - Example: `int Add(n1, n2, n3, ...)`
    - Equivalence Definition 1: partition test cases by the number of inputs (1, 2, 3, etc.)
    - Equivalence Definition 2: partition test cases by the number signs they contain (positive, negative, both)
    - Equivalence Definition 3: partition test cases by the magnitude of operands (large numbers, small numbers, both)
    - Etc.

# Equivalence Partitioning

- When designing test cases, you may use different definitions of “equivalence”, each of which will partition the test case space differently
  - Example: string Fetch(URL)
    - Equivalence Definition 1: partition test cases by URL protocol (“http”, “https”, “ftp”, “file”, etc.)
    - Equivalence Definition 2: partition test cases by type of file being retrieved (HTML, GIF, JPEG, Plain Text, etc.)
    - Equivalence Definition 3: partition test cases by length of URL (very short, short, medium, long, very long, etc.)
    - Same host
    - Etc.

# Equivalence Partitioning

- Test multiple values in each equivalence class.
- Often you're not sure if you have defined the equivalence classes correctly or completely, and testing multiple values in each class is more thorough than relying on a single value.

# Guidelines for Defining Equivalence Classes

- If an input condition specifies **a range**, one valid and two invalid equivalence classes are defined
  - Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}
- If an input condition requires **a specific value**, one valid and two invalid equivalence classes are defined
  - Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}
- If an input condition specifies **a member of a set**, one valid and one invalid equivalence class are defined
  - Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- If an input condition is **a Boolean value**, one valid and one invalid class are defined
  - Input: {true condition} Eq classes: {true condition}, {false condition}

# Determining Equivalence Classes

- Look for ranges of numbers or values
- Look for memberships in groups
- Some may be based on time
- Include invalid inputs
- Look for internal boundaries
- Don't worry if they overlap with each other —
  - better to be redundant than to miss something
- However, test cases will easily overlap with boundary value test cases

# Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	?	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

# Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: $-99 \leq N \leq 99$	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

# Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: $-99 \leq N \leq 99$	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, ...} Non-numeric strings {junk, 1E2, \$13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

# Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: $-99 \leq N \leq 99$	$[-99, -10]$ $[-9, -1]$ 0 $[1, 9]$ $[10, 99]$	$< -99$ $> 99$ Malformed numbers {12-, 1-2-3, ...} Non-numeric strings {junk, 1E2, \$13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	555-5555 (555)555-5555 555-555-5555 $200 \leq \text{Area code} \leq 999$ $200 < \text{Prefix} \leq 999$	?

# Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: $-99 \leq N \leq 99$	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, ...} Non-numeric strings {junk, 1E2, \$13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	555-5555 (555)555-5555 555-555-5555 $200 \leq \text{Area code} \leq 999$ $200 < \text{Prefix} \leq 999$	Invalid format 5555555, (555)(555)5555, etc.. Area code < 200 or > 999 Area code with non-numeric characters <i>Similar for Prefix and Suffix</i>

# Equivalence Partitioning

- Technique
  - Inputs/outputs/internal values of the software are divided into groups that are expected to exhibit similar behavior .
- Examples
  - People (woman / man)
  - Students (bachelor / master)
  - Tickets (children / youth / adults /older)
  - Vehicles (gasoline / diesel / electric)

# Equivalence Partitioning

- Technique
  - Inputs/outputs/internal values of the software are divided into groups that are expected to exhibit similar behavior .
- Equivalence partitions can be found for both valid data and invalid data, i.e. values that should be rejected.
- Notes
  - Tests can be designed to cover more than one partitions
  - Equivalence partitioning is applicable at all levels of testing.
  - Equivalence partitioning as a technique can be used to achieve input and output coverage

# Boundary Value Testing

- Test values, sizes, or quantities near the design limits
  - value limits, length limits, volume limits
  - null strings vs. empty strings
- Errors tend to occur near the extreme values of inputs (off by one is an example)
- Robustness:
  - How does the software react when boundaries are exceeded?
- Use input values
  - at their minimum, just above the minimum
  - at a nominal value,
  - at the maximum, just below the maximum

# Boundary Value Analysis

- A greater number of errors occur at the boundaries of the input domain rather than in the "center"
- Boundary value analysis is a test case design method that complements equivalence partitioning
  - It selects test cases at the edges of a class
  - It derives test cases from both the input domain and output domain

# Guidelines for Boundary Value Analysis

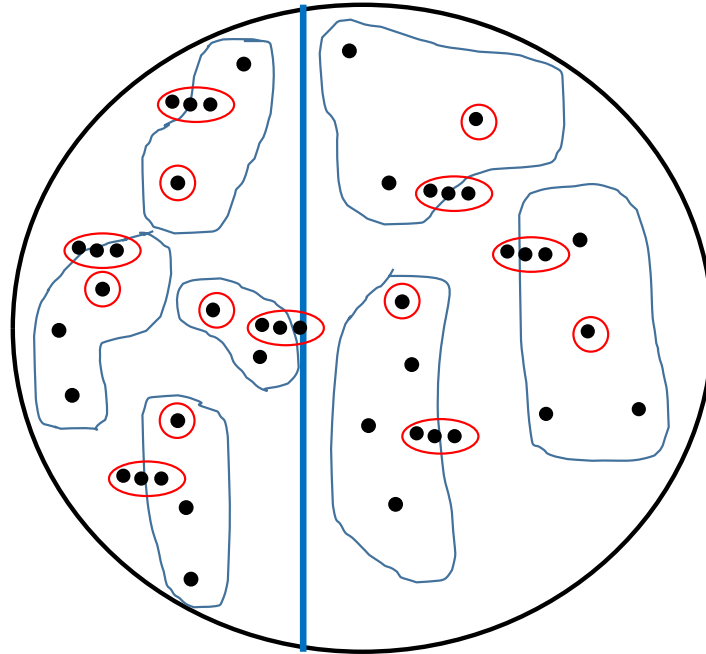
1. If an input condition specifies a range bounded by values  $a$  and  $b$ , test cases should be designed with values  $a$  and  $b$  as well as values just above and just below  $a$  and  $b$
  2. If an input condition specifies a number of values, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested
- Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above
  - If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries

# Boundary Value Analysis

- When choosing values from an equivalence class to test, use the values that are most likely to cause the program to fail
- Errors tend to occur at the boundaries of equivalence classes rather than at the "center"
  - If `(200 < areaCode && areaCode < 999) { // valid area code }`
  - **Wrong!**
  - If `(200 <= areaCode && areaCode <= 999) { // valid area code }`
  - Testing area codes 200 and 999 would catch this error, but a center value like 770 would not
- In addition to testing center values, we should also test boundary values
  - Right on a boundary
  - Very close to a boundary on either side

# Boundary Value Analysis

- Create test cases to test boundaries of equivalence classes



# Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: -99 <= N <= 99	?
Phone Number Area code: [200 .. 999] Prefix: [200 .. 999] Suffix: Any 4 digits	?

# Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: $-99 \leq N \leq 99$	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200 .. 999] Prefix: [200 .. 999] Suffix: Any 4 digits	?

# Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: $-99 \leq N \leq 99$	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200 .. 999] Prefix: [200 .. 999] Suffix: Any 4 digits	Area code: 199, 200, 201 Area code: 998, 999, 1000 Prefix: 200, 199, 198 Prefix: 998, 999, 1000 Suffix: 3 digits, 5 digits

# Boundary Value Analysis - examples

- Numeric values are often entered as strings which are then converted to numbers internally [int x = atoi(str);]
- This conversion requires the program to distinguish between digits and non-digits
- A boundary case to consider: Will the program accept / and : as digits?

Char	/	0	1	2	3	4	5	6	7	8	9	:
ASCII	47	48	49	50	51	52	53	54	55	56	57	58

# Mainstream usage testing

- Don't get so wrapped up in testing boundary cases that you neglect to test "normal" input values
  - Values that users would typically enter during mainstream usage

# Limitations of Boundary Value Testing

- Doesn't require much thought
- May miss internal boundaries
- Usually assumes the variables are independent
- Values at the boundary may not have meaning

# Boundary value analysis

- Notes

- Boundary value analysis can be applied at all test levels .
- Its relatively easy to apply and its defect finding capability is high
- Detailed specifications are helpful.
- This technique is often considered as an extension of equivalence partitioning
- Boundary values are used for test data selection

# Equivalence partitioning and boundary

- Why do both equivalence partitioning and boundary value analysis?
- Boundary values are usually extreme values
- To gain confidence to the system we also want to test it under normal circumstances
- Rule of thumb (Closed, valid partitions)
  - Pick two boundary values (min and max), and one value from the middle of the partition.

# Guidelines and observations

- Equivalence Class Testing is appropriate when input data is defined in terms of intervals and sets of discrete values.
- Equivalence Class Testing is strengthened when combined with Boundary Value Testing
- Strong equivalence takes the presumption that variables are independent. If that is not the case, redundant test cases may be generated

# Decision Table testing

- Decision tables are a good way
  - to capture system requirements that contain logical conditions
  - to document internal system design
  - to record complex business rules that a system is to implement
- Recall truth tables in mathematical logic :

	<b>P</b>	<b>Q</b>	<b><math>P \wedge Q</math></b>
Case 1	T	T	T
Case 2	T	F	F
Case 3	F	T	F
Case 4	F	F	F

# Decision Table testing

- When creating decision tables, the specification is analyzed, and actions of the system are identified.
- If the input conditions and actions are stated in a way where they are either be true or false (Boolean), decision tables can be useful
- The decision table contains the triggering conditions
  - all combinations of true and false for all input conditions, and
  - the resulting actions for each combination of conditions.

# Decision Table testing

- Decision tables
  - Cause-effect table
  - Used when input and actions can be expressed as Boolean
  - Systematic way of stating complex business rules
  - Help tester identify effects of combination of different input
  - Effective approach to reveal faults in the requirements

<b>Conditions</b>	R1	R2	R3	R4
Condition_1	T	T	F	F
Condition_2	T	F	T	F
<b>Actions</b>				
Action_1	?	?	?	?
Action_2	?	?	?	?

# Decision Table testing

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
<b>Conditions</b>								
<i>Condition 1</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>Condition 2</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>Condition 3</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>
<b>Actions</b>								
<i>Action 1</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>Action 2</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>Action 3</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>Action 4</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>

# Decision Table testing

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
<b>Conditions</b>								
<i>Condition 1</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>Condition 2</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>Condition 3</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>
<b>Actions</b>	<i>Action 3</i>	x	<i>Action 1</i>	<i>Action 2</i>	<i>Action 4</i>	x	x	x

- The **coverage standard** commonly used with decision table testing is to have at least one test per column, which typically involves covering all combinations of triggering conditions.

# Decision Table testing

	Rule 1	Rule 2 & 6	Rule 3	Rule 4	Rule 5	Rule 7 & 8
<b>Conditions</b>						
<i>Condition 1</i>	<i>True</i>	----	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>Condition 2</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>
<i>Condition 3</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>	---
<b>Actions</b>	<i>Action 3</i>	x	<i>Action 1</i>	<i>Action 2</i>	<i>Action 4</i>	x

The **coverage standard** commonly used with decision table testing is to have at least one test per column, which typically involves covering all combinations of triggering conditions.

# Decision Table testing

- **Example-Decision table for credit-card**
  - If you are a new customer opening a credit card account, you will get a 15% discount on all your purchases today.
  - If are an existing customer and you hold a loyalty card, you get 10% discount.
  - If you have a coupon, you can get 20% off today (but it can't be used with a 'new-customer' discount)
  - Discount are added, if applicable.

# Decision Table testing

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
<b>Conditions</b>								
<i>New customer (15%)</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>Loyalty card (10%)</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>Coupon (20%)</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>
<b>Actions</b>								
<i>Discount</i>	x	x	20 %	15%	30%	10%	20%	0%

# Decision Table testing

	Rule 1 & 2	Rule 3 & 7	Rule 4	Rule 5	Rule 6	Rule 8
<b>Conditions</b>						
<i>New customer (15%)</i>	<i>True</i>	<i>---</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>Loyalty card (10%)</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>
<i>Coupon (20%)</i>	<i>---</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>
<b>Actions</b>						
<i>Discount</i>	x	20 %	15%	30%	10%	0%

# Decision Table testing

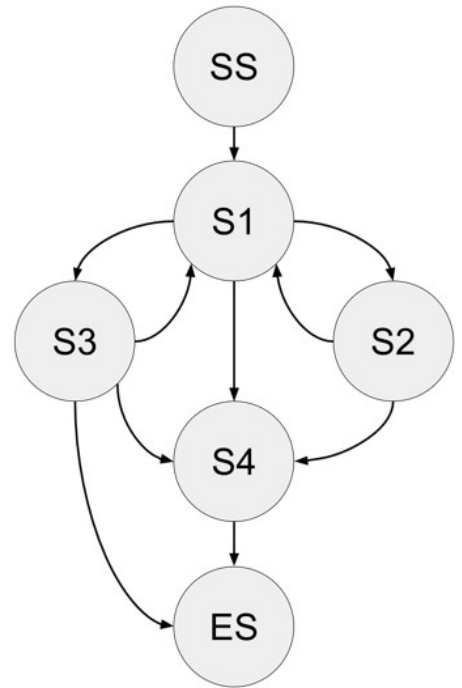
- **Decision tables** are a good way to:
  - capture system requirements that contain *logical conditions*
  - to document internal system design.
- The input conditions and actions are most often stated in such a way that they can be either true or false (Boolean).
- The strength of decision table testing is that it creates combinations of conditions that might not otherwise have been exercised during testing.
- It may be applied to all situations when the action of the software depends on several logical decisions.

# State transition testing

- **A system** can be in a finite number of different states. This aspects of the system can be described as a *'finite state machine'* ; a *state diagram*.
- Any system where you get a different output for the same input, depending on what has happened before, is a finite state system.
- The transition from one state to another are determined by the rules of the 'machine'.

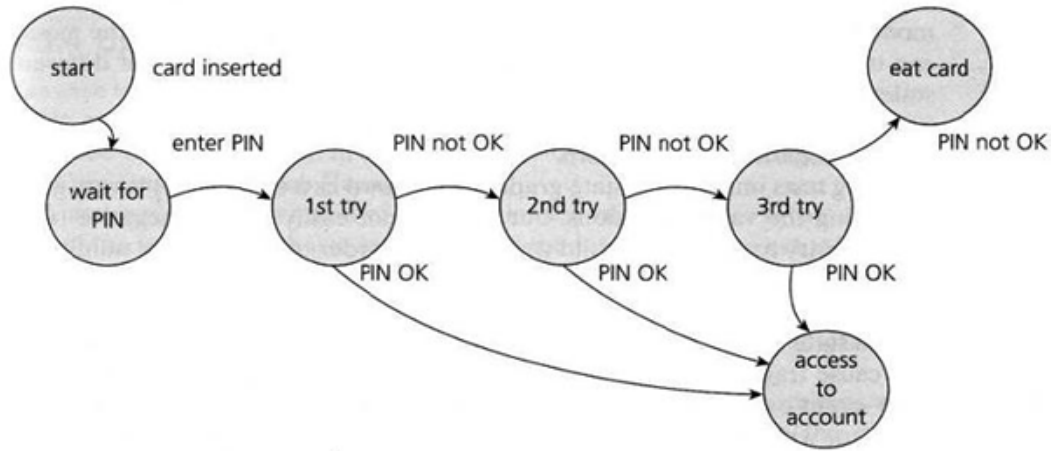
# State transition testing

- State transition testing
  - System can be in a finite number of different states
- Elements of state transition models
  - States (The software may occupy)
    - Open/closed, active/inactive
  - Transitions (From one state to another)
    - Not all transitions are allowed
  - Events (Causing state transition)
    - Closing a files/withdrawing money
  - Actions (Action resulting from transitions)
    - Error message



# State transition testing

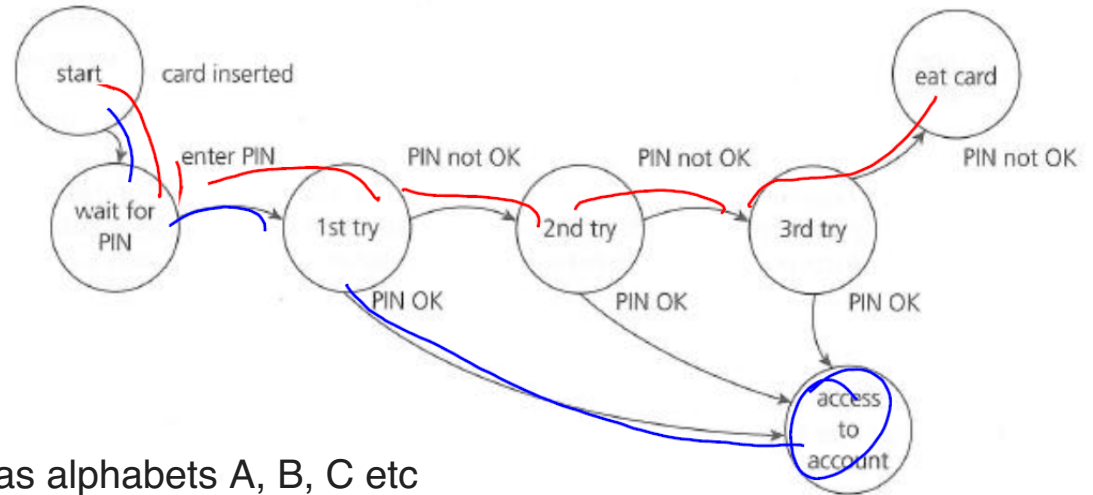
- A 'finite state machine' is often shown as a state diagram
- ATM PIN example.



- The states of the system under test are separate, identifiable and finite in number.

# State transition testing

- How many tests do we need to exercise every state?



**States:** States can be numbered as S1, S2 or as alphabets A, B, C etc

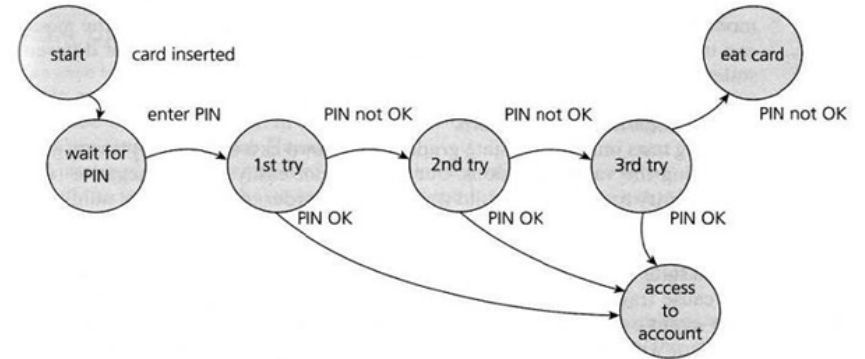
S1:Start, S2:Wait for Pin, S3: 1st try, S4: 2nd Try, S5: 3rd Try, S6: access to account, S7: eat card

**Events:**

Event1:Card inserted, Event 2: enter Pin, Event 3: Pin OK, Event 4: Pin not OK

**Actions :** (not shown in the above example) could be : Messages on the screen – error or otherwise.

# State transition testing



	Event 1 (Insert card)	Event 2 (Enter Pin)	Event 3 (Pin OK)	Event 4 (Pin not OK)
S1:Start	S2	-	-	-
S2:Wait for Pin	-	S3	-	-
S3: 1st try	-	-	S6	S4
S4: 2nd Try	-	-	S6	S5
S5: 3rd Try	-	-	S6	S7
S6: access to account	-	-	-	-
S7: eat card	S1	-	-	-

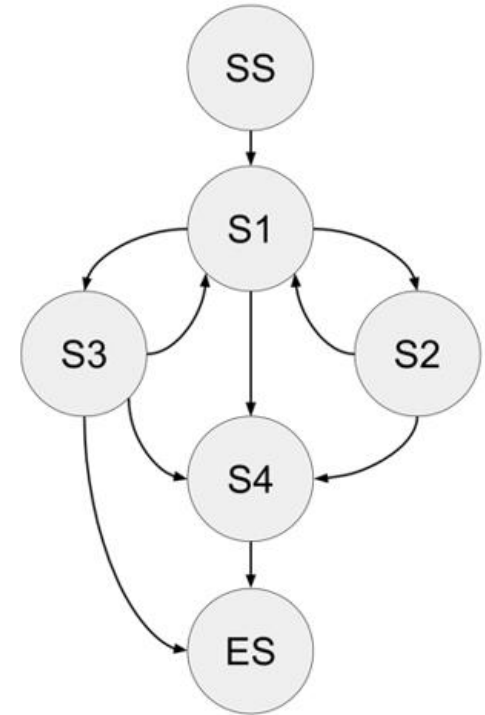
- Invalid or Null Transitions are represented as '-' in red in the table above.

# State transition testing

- Why state transition testing?
  - Because a system may exhibit a different response depending on current conditions or previous history.
- State transition testing allows the tester to view:
  - the software in terms of its states
  - transitions between states
  - the inputs or events that trigger state changes (transitions)
  - the actions which may result from those transitions

# State transition testing

- Tests can be designed
  - to cover a typical sequence of states
  - to exercise specific sequences of transitions
  - to cover every state
  - to exercise every transition
  - to test invalid transitions
- State transition testing is much used within the software industry and technical automation in general.



# Use case testing

- Use case describes interactions between actors (users and the system), which produce a result of value to a system user

## Example

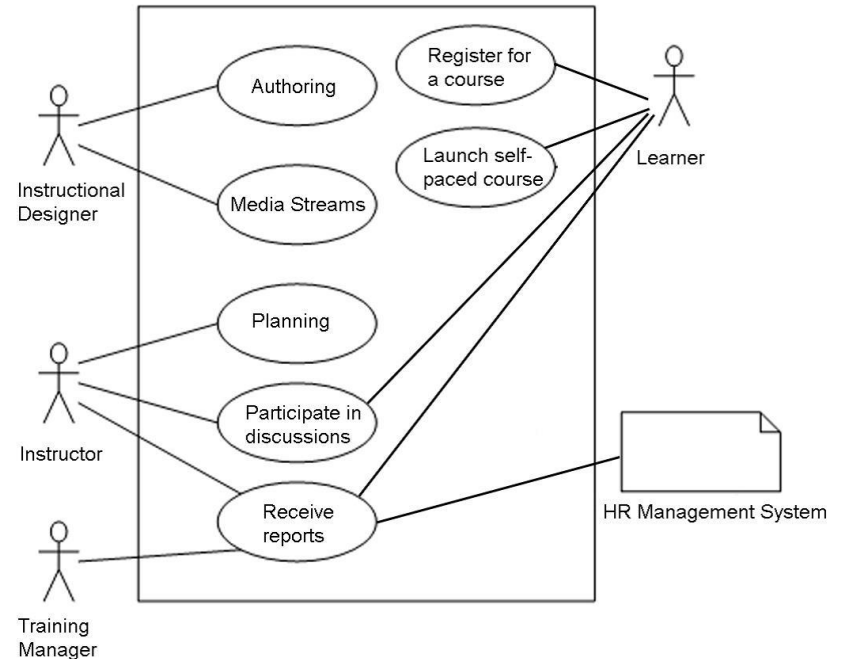
An on-line training website:

User 1: the learner

User 2: the tutor (instructor)

User 3: the training manager

User 4: the instructional designer



# Use case testing

- Identify test cases that exercise the whole system on a transaction by transaction basis from start to finish.
- Describe interactions between actor and system
- Use the language and terms of the business rather than technical terms, especially when the actor is a business user.
- Can uncover integration defects, that is, defects caused by the incorrect interaction between different components.

Use case name	<name>	
Actor(s)	<actor1>, ...	
Pre-conditions	<cond1>, ...	
Post-conditions	<cond1>, ...	
Main Success Scenario	Step	Description
	1	A: <action>
	2	S: <response>
	3	A: <action>
	4	S: <response>
	...	...
Extensions	Step	Description
	S.X	<cause> S: <response>
	S.Y	<cause> S: <response>

# Use case testing

- Each use case has pre-conditions, which need to be met for a use case to work successfully.
- Each use case terminates with post-conditions, which are the observable results and final state of the system after the use case has been completed.
- A use case usually has a *mainstream* (i.e. most likely) scenario, and sometimes *alternative branches*.

# Use case testing

	Step	Description
<b>Main Success Scenario</b>  A: Actor S: System	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
<b>Extensions</b>	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

# Use case testing

- Very useful for designing acceptance tests with customer/user participation.
- Describe the 'process flows' through a system based on its actual likely use.
- Derived from use cases are most useful in uncovering defects in the process flows during real-world use of the system.
- Help uncover integration defects caused by the integration and interference of different components, which individual testing would not see.
- Designing test cases from use cases may be combined with other specification-based test techniques.

# Summary: Key Concepts

- Black-box testing
  - vs. random testing, white-box testing
- Black box testing techniques
  - Equivalence class
  - Boundary value testing
  - Decision tables
  - State transition
  - Use case testing

# General Testing

# Testing for race conditions and other timing dependencies

- Many systems perform multiple concurrent activities
  - Operating systems manage concurrent programs, interrupts, etc.
  - Servers service many clients simultaneously
  - Applications let users perform multiple concurrent actions
- Test a variety of different concurrency scenarios, focusing on activities that are likely to share resources (and therefore conflict)
- "Race conditions" are bugs that occur only when concurrent activities interleave in particular ways, thus making them difficult to reproduce
- Test on hardware of various speeds to ensure that your system works well on both slower and faster machines

# Performance Testing

- Measure the system's performance
  - Running times of various tasks
  - Memory usage, including memory leaks
  - Network usage (Does it consume too much bandwidth? Does it open too many connections?)
  - Disk usage (Is the disk footprint reasonable? Does it clean up temporary files properly?)
  - Process/thread priorities (Does it play well with other applications, or does it hog the whole machine?)

# Performance Testing

- Tools
  - IBM Rational Performance Tester
  - Apache Jmeter
  - WebLOAD
  - LoadRunner

# Limit Testing

- Test the system at the limits of normal use
- Test every limit on the program's behavior defined in the requirements
  - Maximum number of concurrent users or connections
  - Maximum number of open files
  - Maximum request size
  - Maximum file size
  - Etc.
- What happens when you go slightly beyond the specified limits?
  - Does the system's performance degrade dramatically, or gracefully?

# Stress Testing

- Test the system under extreme conditions (i.e., beyond the limits of normal use)
- Create test cases that demand resources in abnormal quantity, frequency, or volume
  - Low memory conditions
  - Disk faults (read/write failures, full disk, file corruption, etc.)
  - Network faults
  - Unusually high number of requests
  - Unusually large requests or files
  - Unusually high data rates (what happens if the network suddenly becomes ten times faster?)
- Even if the system doesn't need to work in such extreme conditions, stress testing is an excellent way to find bugs

# Security Testing

- Any system that manages sensitive information or performs sensitive functions may become a target for intrusion (i.e., hackers)
- How feasible is it to break into the system?
- Learn the techniques used by hackers
- Try whatever attacks you can think of
- Hire a security expert to break into the system
- If somebody broke in, what damage could they do?
- If an authorized user became disgruntled, what damage could they do?

# Security Testing

- Tools:
  - Metasploit
  - W3af
  - Zed Attack Proxy (ZAP)
  - Wapiti

# Usability Testing

- Is the user interface intuitive, easy to use, organized, logical?
- Does it frustrate users?
- Are common tasks simple to do?
- Does it conform to platform-specific conventions?
  
- Get real users to sit down and use the software to perform some tasks
- Watch them performing the tasks, noting things that seem to give them trouble
- Get their feedback on the user interface and any suggested improvements
  
- Report bugs for any problems encountered

# Usability Testing

- Tools:
  - Crazy Egg
  - Optimizely
  - Usabilla

# Recovery Testing

- Try turning the power off or otherwise crashing the program at arbitrary points during its execution
  - Does the program come back up correctly when you restart it?
  - Was the program's persistent data corrupted (files, databases, etc.)?
  - Was the extent of user data loss within acceptable limits?
- Can the program recover if its configuration files have been corrupted or deleted?
- What about hardware failures? Does the system need to keep working when its hardware fails? If so, verify that it does so.

# Configuration Testing

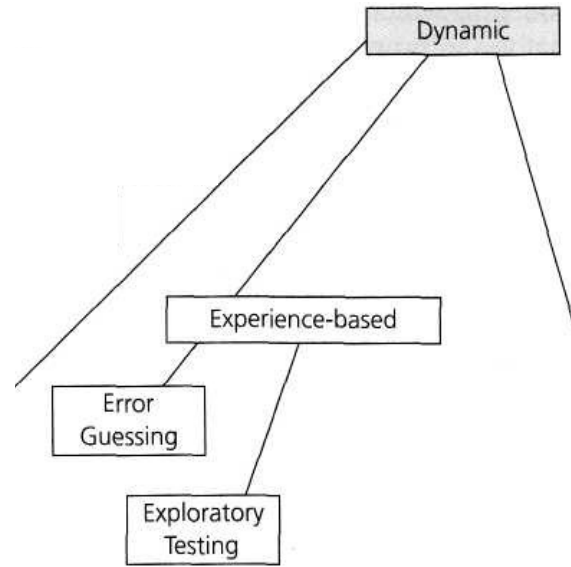
- Test on all required hardware configurations
  - CPU, memory, disk, graphics card, network card, etc.
- Test on all required operating systems and versions thereof
  - Virtualization technologies such as VMWare and Virtual PC are very helpful for this
- Test as many Hardware/OS combinations as you can
- Test installation programs and procedures on all relevant configurations

# Compatibility Testing

- Test to make sure the program is compatible with other programs it is supposed to work with
  - Ex: Can Word 12.0 load files created with Word 11.0?
  - Ex: "Save As... Word, Word Perfect, PDF, HTML, Plain Text"
  - Ex: "This program is compatible with Internet Explorer and Firefox"
- Test all compatibility requirements

# Documentation Testing

- Test all instructions given in the documentation to ensure their completeness and accuracy
- For example, “How To ...” instructions are sometimes not updated to reflect changes in the user interface
- Test user documentation on real users to ensure it is clear and complete



# Experience-based techniques

# Experience-based techniques

- Tests are derived from the tester's skill and intuition and their experience with similar applications and technologies.
- When used to augment systematic techniques, experienced based testing can be useful in identifying special tests not easily captured by formal techniques, especially when applied after more formal approaches.
- May yield widely varying degrees of effectiveness , depending on the testers experience.

# Experience-based techniques

- Error guessing = a commonly used experienced-based technique.
- Generally testers anticipate defects based on experience.
- A structured approach to the error guessing technique is to enumerate a list of possible errors and to design tests that attack these errors.
- This systematic approach is called fault attack.

# Experience-based techniques

- Exploratory testing = concurrent test design, test execution, test logging and learning,
- *based on a test charter containing test objectives, and carried out within time-boxes.*
- It is most useful ...
  - where there are few or inadequate specifications
  - under severe time pressure
  - to complement other, more formal testing
  - It can serve to help ensure that the most serious defects are found.

# Special Value Testing

- The most widely practiced form of functional testing
- The tester uses his or her domain knowledge, experience, or intuition to probe areas of probable errors
- Other terms: “hacking”, “out-of-box testing”, “ad hoc testing”, “seat of the pants testing”, “guerilla testing”

# Uses of Special Value Testing

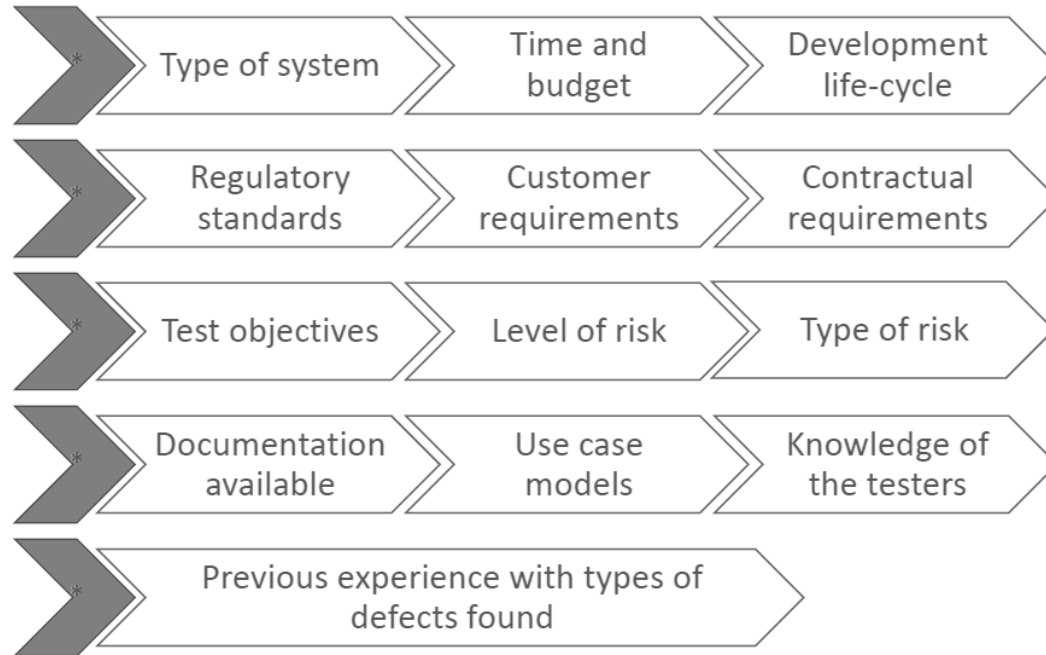
- Complex mathematical (or algorithmic) calculations
- Worst case situations (similar to robustness)
- Problematic situations from past experience
- “Second guess” the likely implementation

# Characteristics of Special Value Testing

- Experience really helps
- Frequently done by the customer or user
- Defies measurement
- Highly intuitive
- Seldom repeatable
- Often, very effective

# Choosing test techniques

- The choice of which test techniques to use depends on a number of factors, including:



# Summary

- Case Study – Knight Capital
- Testing techniques
- Black Box Testing
- Black Box Testing Techniques
- General Testing
- Experience based testing