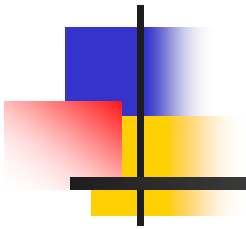
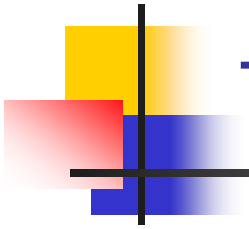
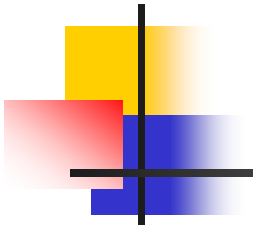


Testing Basics



Testing

Testing is the process of determining if a program has any errors.



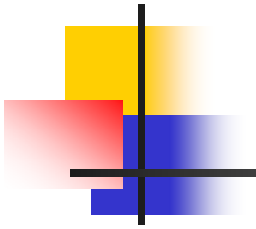
Test case/data

A test case is a pair consisting of test data to be input to the program and the expected output. The test data is a set of values, one for each input variable.

A test set is a collection of zero or more test cases.

Sample test case for sort:

Test data: <"A'' 12 -29 32 >
Expected output: -29 12 32



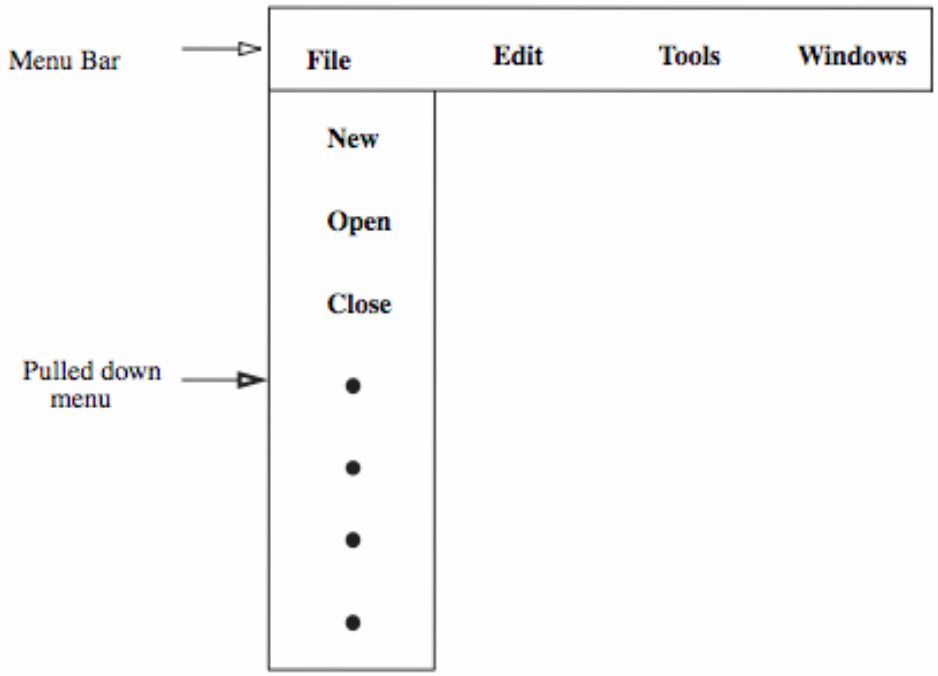
Program behavior

Can be specified in several ways: plain natural language, a state diagram, formal mathematical specification, etc.

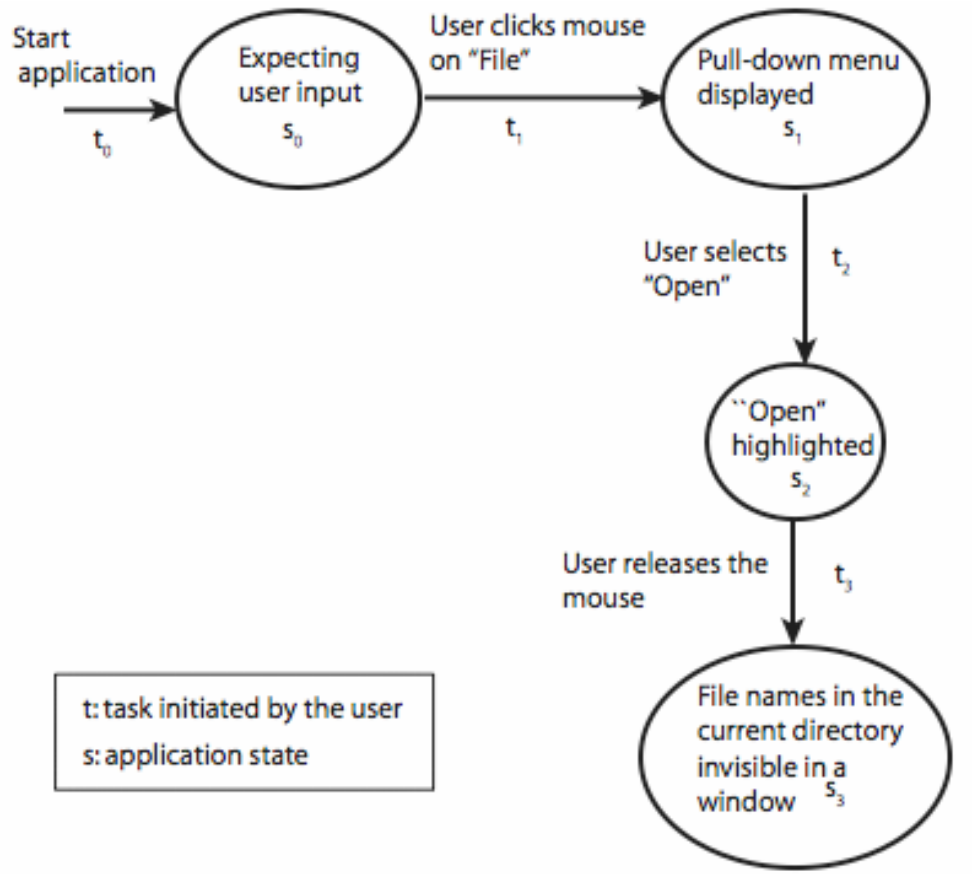
A state diagram specifies program states and how the program changes its state on an input sequence. inputs.

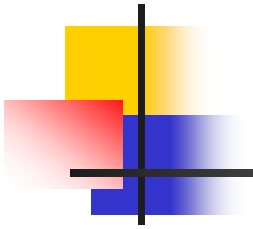
Program behavior: Example

Consider a menu driven application.



Program behavior: Example (contd.)





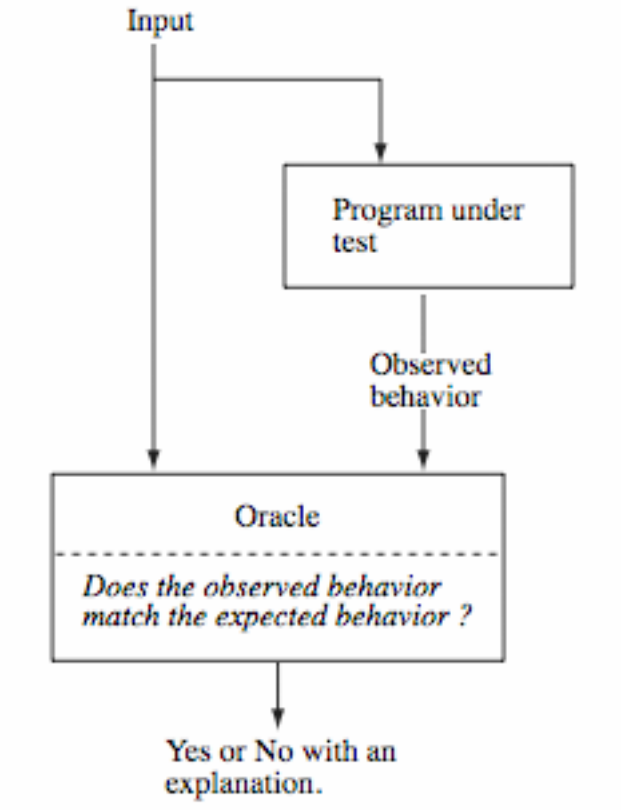
Behavior: observation and analysis

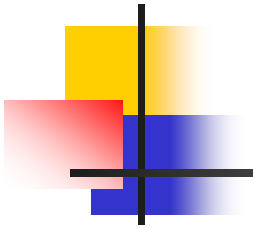
In the first step one observes the behavior.

In the second step one analyzes the observed behavior to check if it is correct or not. Both these steps could be quite complex for large commercial programs.

The entity that performs the task of checking the correctness of the observed behavior is known as an oracle.

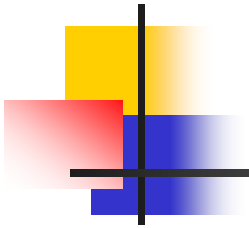
Oracle: Example





Oracle: Programs

Oracles can also be programs designed to check the behavior of other programs.



Types of testing

One possible classification is based on the following four classifiers:

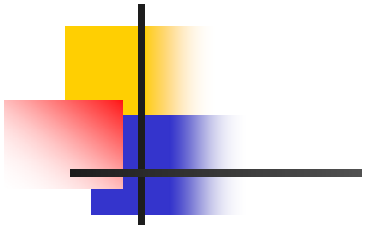
C1: Source of test generation.

C2: Lifecycle phase in which testing takes place

C3: Goal of a specific testing activity

C4: Characteristics of the artifact under test

C1: Source of test generation



Artifact	Technique	Example
Requirements (informal)	Black-box	Ad-hoc testing Boundary value analysis Category partition Classification trees Cause-effect graphs Equivalence partitioning Partition testing Predicate testing Random testing
Code	White-box	Adequacy assessment Coverage testing Data-flow testing Domain testing Mutation testing Path testing Structural testing Test minimization using coverage
Requirements and code	Black-box and White-box	
Formal model: Graphical or mathematical specification	Model-based Specification	Statechart testing FSM testing Pairwise testing Syntax testing
Component interface	Interface testing	Interface mutation Pairwise testing

C2: Lifecycle phase in which testing takes place

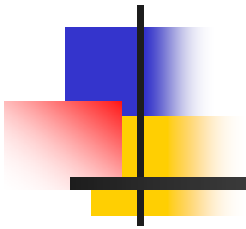
Phase	Technique
Coding	Unit testing
Integration	Integration testing
System integration	System testing
Maintenance	Regression testing
Post system, pre-release	Beta-testing

C3: Goal of specific testing activity

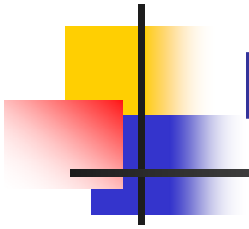
Goal	Technique	Example
Advertised features	Functional testing	
Security	Security testing	
Invalid inputs	Robustness testing	
Vulnerabilities	Vulnerability testing	
Errors in GUI	GUI testing	Capture/plaback
		Event sequence graphs
		Complete Interaction Sequence
Operational correctness	Operational testing	Transactional-flow
Reliability assessment	Reliability testing	
Resistance to penetration	Penetration testing	
System performance	Performance testing	Stress testing
Customer acceptability	Acceptance testing	
Business compatibility	Compatibility testing	Interface testing
		Installation testing
Peripherals compatibility	Configuration testing	

C4: Artifact under test

Characteristics	Technique
Application component	Component testing
Client and server	Client-server testing
Compiler	Compiler testing
Design	Design testing
Code	Code testing
Database system	Transaction-flow testing
OO software	OO testing
Operating system	Operating system testing
Real-time software	Real-time testing
Requirements	Requirement testing
Software	Software testing
Web service	Web service testing



Functional Testing



Learning Objectives

- Equivalence class partitioning
- Boundary value analysis
- Test generation from predicates

Essential black-box techniques for generating tests for functional testing.

Equivalence class partitioning

Equivalence Class Testing

- Complete testing
- Avoiding redundancy

Equivalence classes form a partition of a set.

Partition: collection of mutually disjoint subsets whose union is the entire set.

Equivalence testing: use one element from each equivalence class.

Key: choice of equivalence relation.

Program: f(a,b,c) with input domains A, B, and C.

A = A1 U A2 U A3

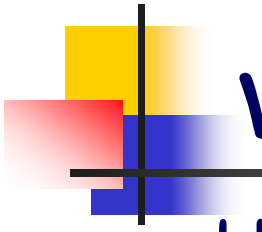
B = B1 U B2 U B3 U B4

C = C1 U C2

Elements of partition denoted as: $a1 \in A1$

$b3 \in B3$

$c2 \in C2$

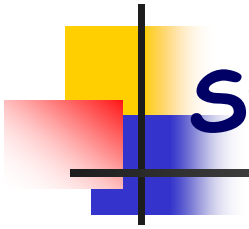


Weak Equivalence Class Testing

- Use one variable from each equivalence class in a test case.

Test Case	a	b	c
1	a1	b1	c1
2	a2	b2	c2
3	a3	b3	c1
4	a1	b4	c2

#test cases = #classes in the partition with the largest numbering of subsets.



Strong Equivalence Class Testing

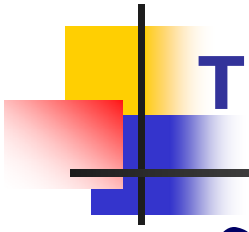
- Based on Cartesian product of the partition subsets.

  A X B X C will have 3 X 4 X 2 = 24 elements

  (a1,b1,c1),(a1,b1,c2),(a1,b2,c1)…..

We cover all the equivalence classes and we have one of each possible combination of inputs.

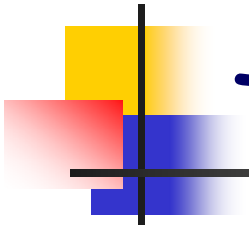
Generalization: equivalence classes on outputs



Traditional Equivalence Class Testing

Given the valid and invalid sets of inputs, the traditional equivalence testing strategy identifies test cases as follows:

- For valid inputs, use one value from each valid class.
- For invalid inputs, a test case will have one invalid value and the remaining values will all be valid.



The Nextdate Program

It is a function that returns the date of the day after the input date. The month, day and year values in the input date have numerical values with the following constraints.

1<= month <= 12

1 <= day <= 31

1812 <= year <= 2012

Note: A year is a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400. So 2000 is a leap year while the year 1900 is not a leap year.

e.g., valid ranges for next date problem

1<= month <= 12; 1 <= day <= 31;

1812 <= year <= 2012

invalid ranges

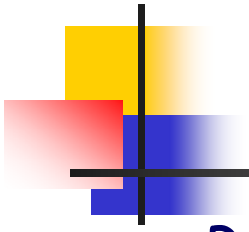
day>31; day<1; month<1; month>12;

year>2012; year<1812

Traditional Equivalence Class Test Cases for Next Date Function

Test Case	Month	Day	Year	Expected Output
1	6	15	1912	6/16/1912
2	-1	15	1912	Invalid Input
3	13	15	1912	Invalid Input
4	6	-1	1912	Invalid Input
5	6	32	1912	Invalid Input
6	6	15	1811	Invalid Input
7	6	15	2013	Invalid Input

- Equivalence relation defines the class of elements that should be treated in the same way.
- Deficiency of traditional approach: same treatment at valid/invalid level.
- Better Equivalence relation?
  - Look at the functionality of the program, that is, what must be done to input date?



Postulate the following equivalence classes:

M1 = {month: month has 30 days}

M2 = {month: month has 31 days}

M3 = {month: month is February}

D1 = {day: 1<=day<=28}

D2 = {day: day=29}

D3 = {day: day=30}

D4 = {day: day=31}

Y1 = {year: year = 1900}

Y2 = {year: 1812<=year<=2012 AND (year!=1900)
AND(year=0 mod 4)}

Y3 = {year: (1812<=year<=2012 AND year!=0 mod 4}

Weak Equivalence Class Test Cases

Test Case	Month	Day	Year	Expected Output
1	6	14	1900	6/15/1900
2	7	29	1912	7/30/1912
3	2	30	1913	Invalid Input
4	6	31	1900	Invalid Input

Strong Equivalence Class Test Cases

(m1,m2,m3) X (d1,d2,d3,d4) X (y1,y2,y3)

3 x 5 x 3 = 45 test cases

Equivalence classes for variables: range

Eq. Classes	Example	
	Constraints	Classes
One class with values inside the range and two with values outside the range.	speed $\in$[60..90]	{50}, {75}, {92}
	area: float area$\geq$0.0	{{-1.0}, {15.52}}
	age: int	{{-1}, {56}, {132}}
	letter:bool	{{J}, {3}}

Equivalence classes for variables: strings

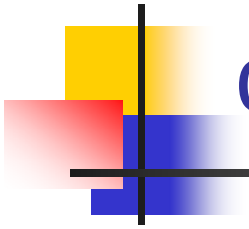
Eq. Classes	Example	
	Constraints	Classes
At least one containing all legal strings and one all illegal strings based on any constraints.	firstname: string	{{ε}, {Sue}, {Loooong Name}}

Equivalence classes for variables: enumeration

Eq. Classes	Example	
	Constraints	Classes
Each value in a separate class	autocolor:{red, blue, green}	{{red,} {blue}, {green}}
	up:boolean	{{true}, {false}}

Equivalence classes for variables: arrays

Eq. Classes	Example	
	Constraints	Classes
One class containing all legal arrays, one containing the empty array, and one containing a larger than expected array.	int [ ] aName: new int[3];	{[ ]}, {[-10, 20]}, {[-9, 0, 12, 15]}

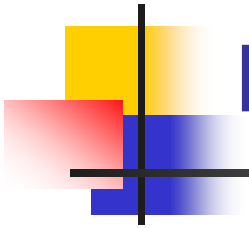


Equivalence classes for variables: compound data type

Arrays in Java and records, or structures, in C++, are compound types. Such input types may arise while testing components of an application such as a function or an object.

While generating equivalence classes for such inputs, one must consider legal and illegal values for each component of the structure.

Boundary value analysis

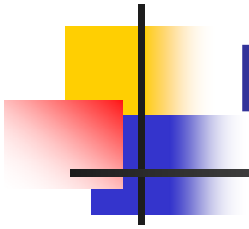


Errors at the boundaries

Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes.

For example, suppose that method M is required to compute a function f1 when $x \leq 0$ is true and function f2 otherwise. However, M has an error due to which it computes f1 for $x<0$ and f2 otherwise.

Obviously, this fault is revealed, though not necessarily, when M is tested against $x=0$ but not if the input test set is, for example, {-4, 7} derived using equivalence partitioning. In this example, the value $x=0$, lies at the boundary of the equivalence classes $x \leq 0$ and $x>0$.

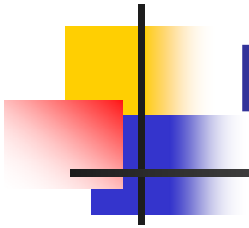


Boundary value analysis (BVA)

Boundary value analysis is a test selection technique that targets faults in applications at the boundaries of equivalence classes.

While equivalence partitioning selects tests from within equivalence classes, boundary value analysis focuses on tests at and near the boundaries of equivalence classes.

Certainly, tests derived using either of the two techniques may overlap.



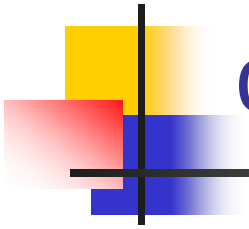
BVA: Procedure

1 Partition the input domain using unidimensional (weak) partitioning. This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning. We will generate several sub-domains in this step.

2 Identify the boundaries for each partition. Boundaries may also be identified using special relationships amongst the inputs.

3 Select test data such that each boundary value occurs in at least one test input.

BVA: Example: 1. Create equivalence classes



Assuming that an item code must be in the range 99..999 and quantity in the range 1..100,

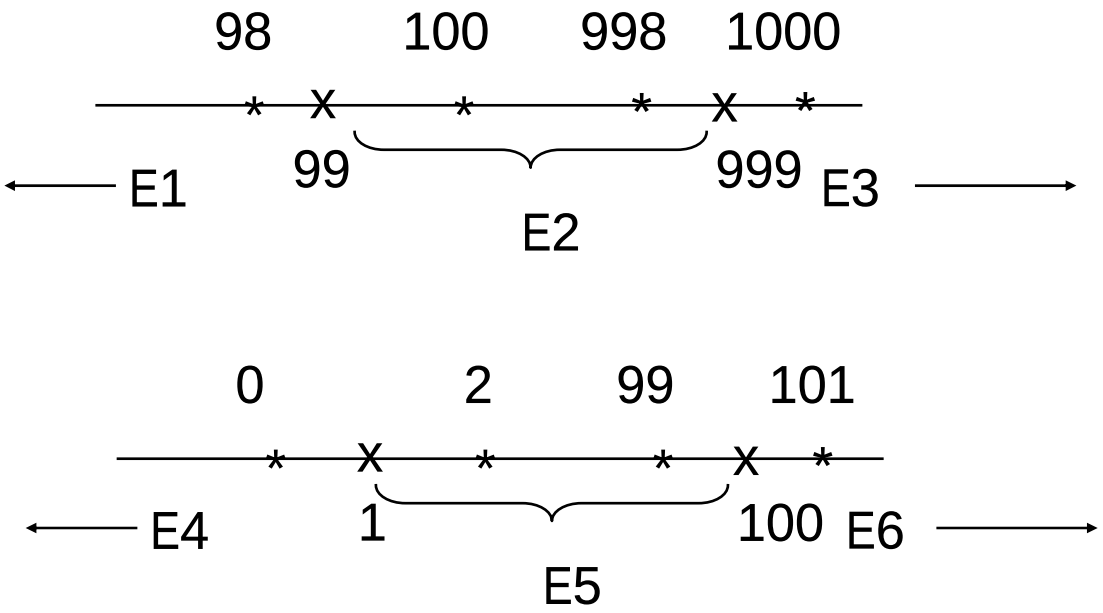
Equivalence classes for code:

- E1: Values less than 99.
- E2: Values in the range.
- E3: Values greater than 999.

Equivalence classes for qty:

- E4: Values less than 1.
- E5: Values in the range.
- E6: Values greater than 100.

BVA: Example: 2. Identify boundaries

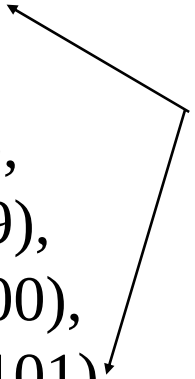


Equivalence classes and boundaries for findPrice. Boundaries are indicated with an x. Points near the boundary are marked *.

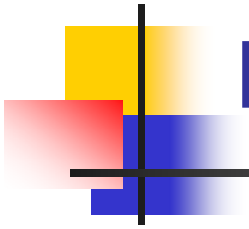
BVA: Example: 3. Construct test set

Test selection based on the boundary value analysis technique requires that tests must include, for each variable, values at and around the boundary. Consider the following test set:

T={ t1: (code=98, qty=0),
t2: (code=99, qty=1),
t3: (code=100, qty=2),
t4: (code=998, qty=99),
t5: (code=999, qty=100),
t6: (code=1000, qty=101)
}



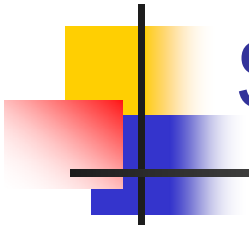
Illegal values of code and qty included.



BVA: In-class exercise

Is T the best possible test set for findPrice? Answer this question based on T's ability to detect missing code for checking the validity of age.

Is there an advantage of separating the invalid values of code and age into different test cases?



Summary

Equivalence partitioning and boundary value analysis are the most commonly used methods for test generation while doing functional testing.

Given a function *f* to be tested in an application, one can apply these techniques to generate tests for *f.*