COMP 3111
SOFTWARE ENGINEERING

LECTURE 14
TESTING

WHITE BOX TESTING: CONDITION TESTING

***Condition testing* executes the true and false value of each simple logical condition in a component.**

Errors in a condition result from errors in:

simple condition: (a rel-op b) where rel-op={<, ≤, =, ≠, ≥, >}
may be negated with NOT, e.g., a≤b; NOT(a≤b)

compound condition: two or more simple conditions connected with AND, OR, e.g., (a>b) AND (c<d)

relational expression: (E_1 rel-op E_2) where E_1 and E_2 are arithmetic expressions, e.g., ((a*b+c)>(a+b+c))

Boolean expression: non relational expressions (e.g., NOT A)

Due to incorrect/missing/extra:

- Boolean operator
- Boolean variable
- parenthesis
- relational operator
- arithmetic expression

WHITE BOX TESTING: CONDITION TESTING (CONT'd)

1. Branch testing

For a compound condition C, test true and false branches of C and every simple condition of C.

e.g., for C = (a>b) AND (c<d) we test for:

C	TRUE and FALSE
a>b	TRUE and FALSE
c<d	TRUE and FALSE

Basis path testing already covers these cases.

2. Domain testing

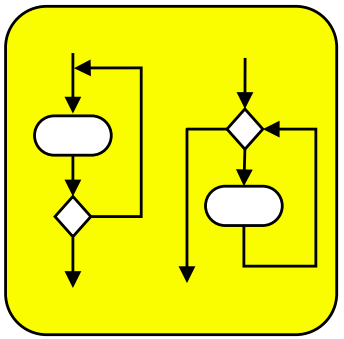
For an expression E_1 rel-op E_2, we test using values that make: E_1 greater than E_2, E_1 equal to E_2, and E_1 less than E_2.

- This guarantees detection of rel-op error if E_1 and E_2 are correct.
- To detect errors in E_1 and E_2, the difference between E_1 and E_2 for the tests that make E_1 greater than E_2 and E_1 less than E_2 should be as small as possible.

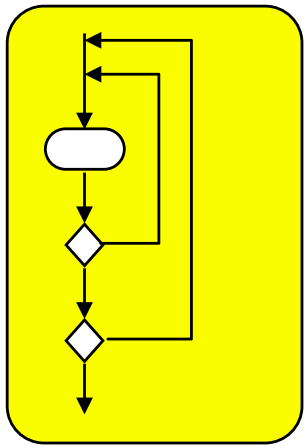
WHITE BOX TESTING: LOOP TESTING

Loop testing **executes loops at and within their bounds.**

1. **Simple Loops (n iterations)**
   1. 0, 1, 2 passes through the loop.
   2. m passes through the loop where m<n.
   3. n-1, n, n+1 passes through the loop.



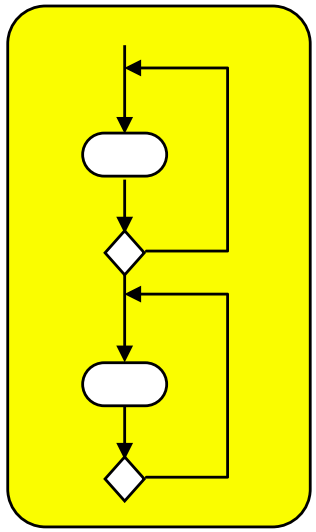
2. **Nested Loops**
   1. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration values.
   2. Work outward, conducting simple loop tests for the next innermost loop.
   3. Continue until all the loops have been tested.



Tests grow geometrically with level of nesting!

WHITE BOX TESTING: LOOP TESTING (CONT'D)

3. Concatenated Loops



☞ If loops are independent
→ use simple loop testing

☞ If loops are dependent (i.e., a loop depends on a variable set in another loop)
→ use nested loop testing

4. Unstructured Loops

☞ **Refactor the code!!!**

These loop test have the highest likelihood of uncovering errors with the minimum amount of effort and test overlap.

WHITE BOX TESTING: DATA FLOW TESTING

Data flow testing **ensures that the value of a variable is correct at certain points of execution in the code.**

Select test paths according to the locations of definitions (S) and uses (S') of a variable (X).

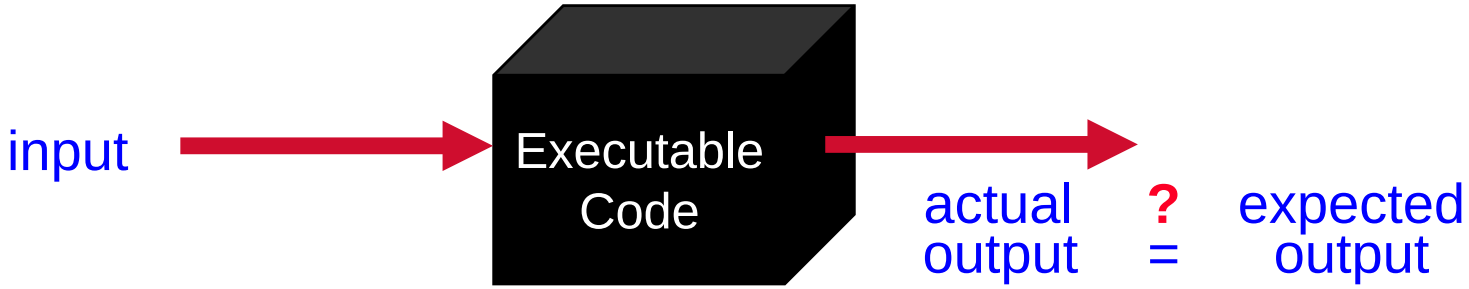
Definition Use (DU) Chain (X) is the set of [X, S, S´] where X is not redefined between S and S´.

S, S' are statement numbers

A testing strategy: Every DU chain must be covered once.

For every variable, do a test along the path from where the variable is defined to the statement(s) where the variable is used.
These tests can be combined with basis path testing.

DESIGN TESTS: BLACK BOX TESTING



Black box tests attempt to find:

- incorrect or missing functions
- interface incompatibility errors
- data structure or external database access errors
- performance errors
- initialization and termination errors

To achieve reasonable testing, a black box test case should cover a range of input or output values, not just a single value. That is, it should tell us something about the presence or absence of a *class of errors* (e.g., all character data is correctly/incorrectly processed).

EQUIVALENCE PARTITIONING

Equivalence partitioning **creates subdomains by grouping inputs and outputs by type to create test coverage of a class of errors.**

- Divide the input/output data of a component into partitions of equivalent data.
- The equivalence partitions are usually derived from the requirements specification.
- Test cases are designed to cover each equivalence partition at least once.

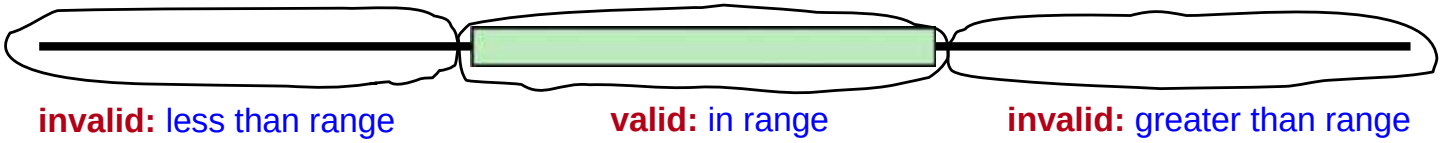
A COMMON PARTITIONING HEURISTIC

Select subdomains based on valid and invalid inputs/outputs.

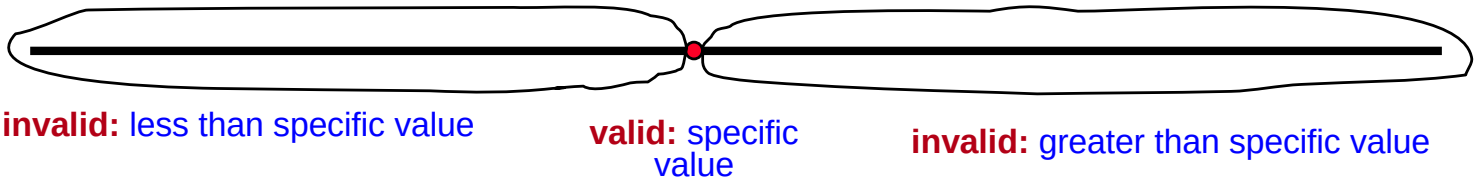
For testing, select "typical" values "inside" a subdomain.

EQUIVALENCE PARTITIONING: SUBDOMAIN SELECTION HEURISTICS

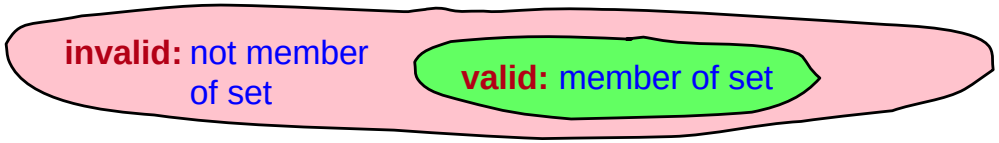
1. If the input is a range → one valid and two invalid subdomains:



2. If the input is a specific value → one valid and two invalid subdomains:



3. If the input is a set of related values → one valid and one invalid subdomain:



4. If the input is Boolean → one valid and one invalid subdomain:

BOUNDARY TESTING

More errors occur at the "boundaries" of a subdomain than in the "center".

Why?

- off-by-one bugs
- forget to handle empty container/null object
- overflow errors in arithmetic
- program does not handle aliasing of objects

- Small subdomains at the "boundaries" of the main subdomains have a high probability of revealing these common errors.

For testing, select values at the "boundaries" of a main subdomain.

BOUNDARY TESTING EXAMPLE

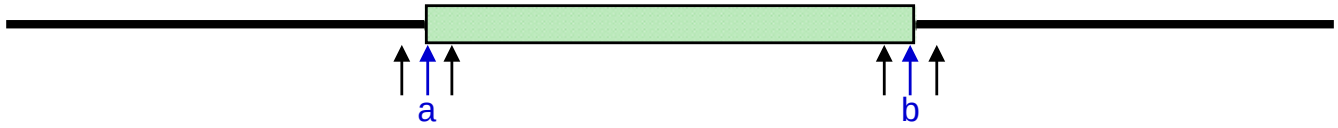
```
public int abs (int x)
// returns: |x|
```

- Tests for **abs**

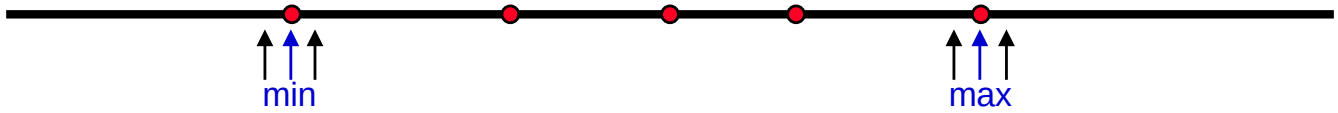
What are some values or ranges of **x** that might be worth testing?

BOUNDARY TESTING: TEST VALUE SELECTION HEURISTICS

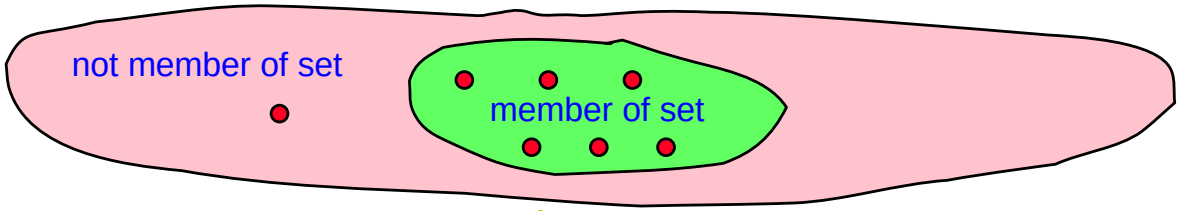
1. If the input is a range and is bounded by **a** and **b**, then use **a**, **b**, and values just above and just below **a** and **b**, respectively.



2. If the input is a number of discrete values, use the **minimum** and the **maximum** of the values and values just above and just below them, respectively. (Can also be applied to a single specific input value.)

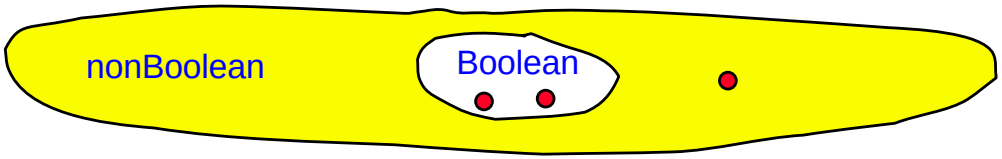


3. If the input is a set of values, test all values in the set (if possible) and one value outside the set.



BOUNDARY TESTING: TEST VALUE SELECTION HEURISTICS

4. If the input is a Boolean, test for both Boolean values (T, F) and for a non-Boolean value.



5. Apply guidelines 1 and 2 to create **output values** at the minimum and maximum expected values.

   E.g., if the output is a table, create a minimum size table (1 row) and a maximum size table.

6. If data structures have boundaries, test these boundary values and values just above and just below them, respectively.

   E.g., for an array with bounds 1 to 10 → test array index = 0, 1, 2, 9, 10, 11.

EXAMPLE ASU BLACKBOX TESTING

Student Record Search

In the ASU System, the *Maintain Student Information* use case uses a procedure that searches the database for the record of a given student. The input to the search procedure is a student ID. The output is either a success indication and the student record containing the given student ID or a failure indication and a message indicating the nature of the failure. The valid range of student IDs is from 1000000 to 9999999.

For each of the test categories below, what test cases and specific test values should be used to test whether the search procedure works correctly and accepts only valid student IDs?

(a) boundary

(b) typical

(c) other

EXAMPLE ASU BLACKBOX TESTING: TEST CASES AND TEST VALUES

Boundary values

	Test Case	Test Value
1.	Minimum range—valid student ID	1000000
2.	Minimum range—valid student ID	1000001
3.	Minimum range—invalid student ID	999999
4.	Maximum range—valid student ID	9999999
5.	Maximum range—valid student ID	9999998
6.	Maximum range—invalid student ID	10000000

Typical values

	Test Case	Test Value
7.	Mid-range—valid student ID; existing student record	5000000
8.	Mid-range—valid student ID; non-existent student record	5000001
9.	Below lower range—invalid student ID	500000
10.	Above upper range—invalid student ID	50000000

Other values

	Test Case	Test Value
11.	Nonnumeric value—invalid student ID	5647ABC

EXAMPLE ASU BLACKBOX TESTING: DESIGNING TEST CASES

Test case procedures: boundary values

Test case 1:	Minimum range—valid student ID
Setup:	Ensure that a record with student ID 1000000 exists in the data base
Test value:	1000000
Verification:	Successful retrieval of student record with ID 1000000

Test case 2:	Minimum range—valid student ID
Setup:	Ensure that a record with student ID 1000001 exists in the data base
Test value:	1000001
Verification:	Successful retrieval of student record with ID 1000000

Test case 3:	Minimum range—invalid student ID
Setup:	None
Test value:	999999
Verification:	Invalid student ID error message

Test case 4:	Maximum range—valid student ID
Setup:	Ensure that a record with student ID 9999999 exists in the data base
Test value:	9999999
Verification:	Successful retrieval of student record with ID 9999999

EXAMPLE ASU BLACKBOX TESTING: DESIGNING TEST CASES

Test case 5:	Maximum range—valid student ID
Setup:	Ensure that a record with student ID 9999998 exists in the data base
Test value:	9999998
Verification:	Successful retrieval of student record with ID 9999998

Test case 6:	Maximum range—invalid student ID
Setup:	None
Test value:	10000000
Verification:	Invalid student ID error message

Test case procedures: typical values

Test case 7:	Mid-range—valid student ID; existing student record
Setup:	Ensure that a record with student ID 5000000 exists in the data base
Test value:	5000000
Verification:	Successful retrieval of student record with ID 5000000

Test case 8:	Mid-range—valid student ID; non-existent student record
Setup:	Ensure that no record with student ID 5000001 exists in the data base
Test value:	5000001
Verification:	Unsuccessful retrieval and no such record error message

EXAMPLE ASU BLACKBOX TESTING: DESIGNING TEST CASES

Test case 9:	Below lower range—invalid student ID
Setup:	None
Test value:	500000
Verification:	Invalid student ID error message

Test case 10:	Above upper range—invalid student ID
Setup:	None
Test value:	50000000
Verification:	Invalid student ID error message

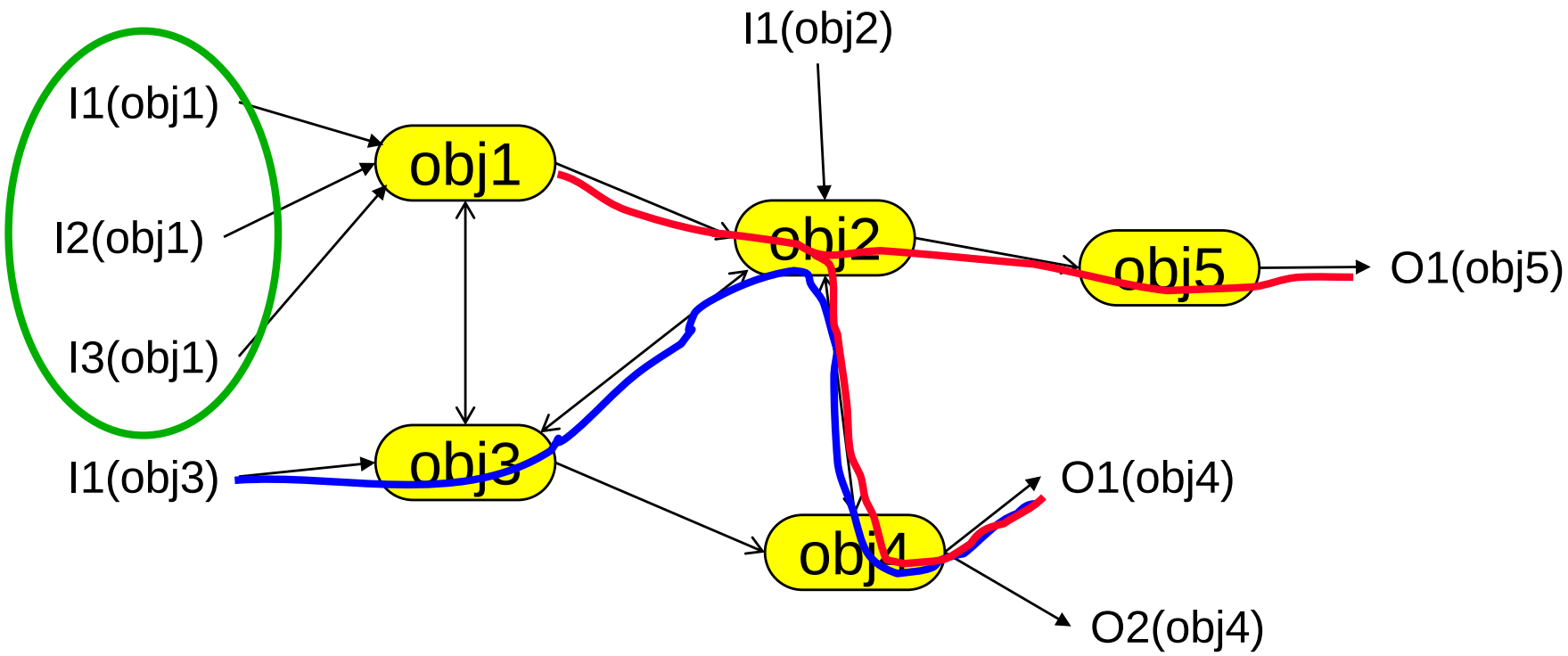
Test case procedures: other values

Test case 11:	Nonnumeric value—invalid student ID
Setup:	None
Test value:	5647ABC
Verification:	Invalid student ID error message

BLACK BOX TESTING: THREAD TESTING

- An event-based approach where tests are based on events which trigger system actions.
  - ☞ **Particularly appropriate for object oriented systems.**
- It is used after classes have been unit tested and integrated into subsystems.
- Need to identify and execute each possible processing thread.
  - ☞ **Examine the use cases.**
- However, it may not be possible to do complete thread testing.
  - ☞ **Since there are usually too many input/output combinations.**
- Therefore, we focus on the most commonly executed threads.
  - ☞ **These are the basic flow of events of use cases.**

BLACK BOX TESTING: THREAD TESTING (CONT'D)

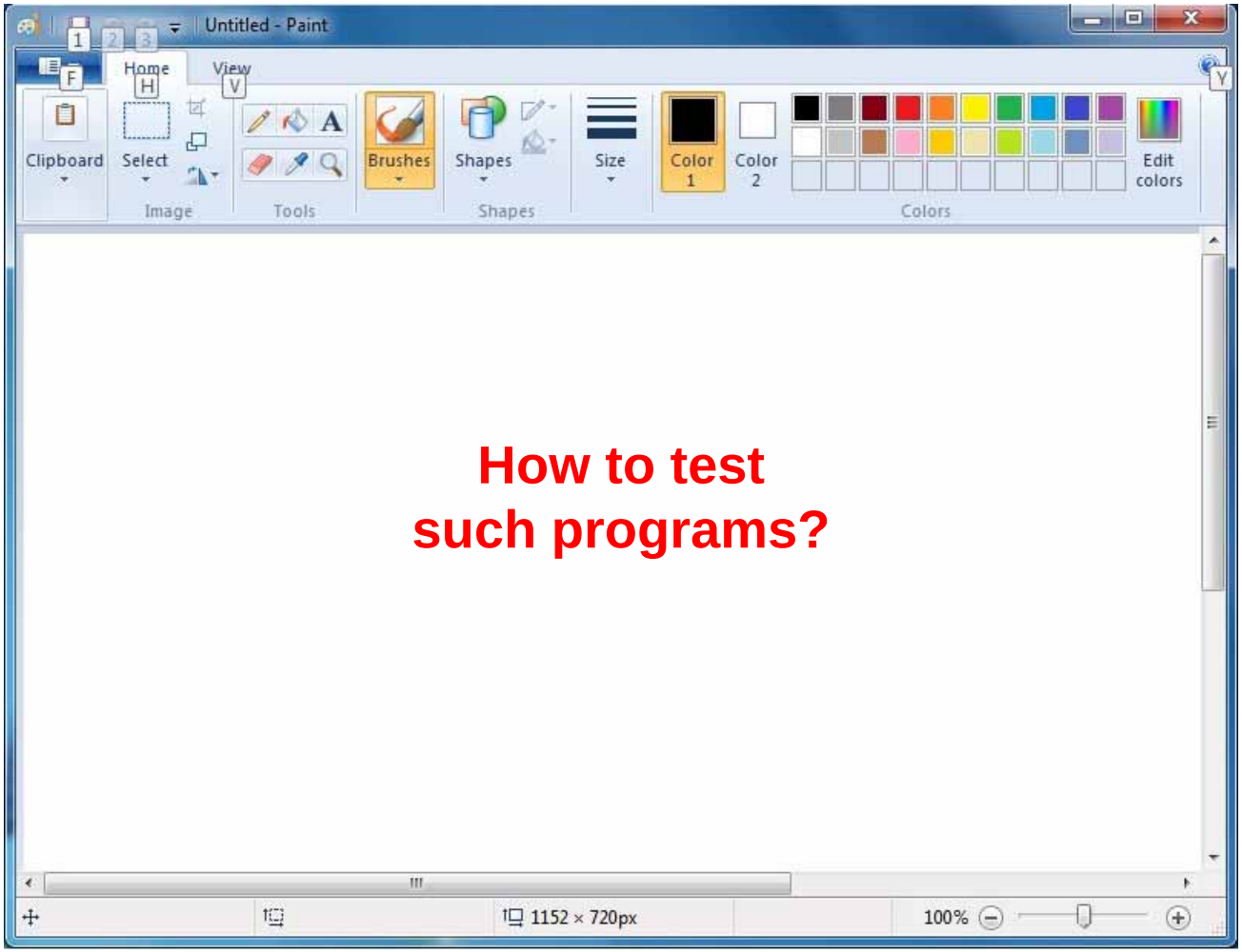


Types of thread tests include:

- single thread: obj3→obj2→obj4
- multiple thread: obj1→obj2→{obj4; obj5}
- multi-input thread: into obj1

BLACK BOX TESTING: STATE-BASED TESTING

Some programs can be in many different states simultaneously.



How to test such programs?

BLACK BOX TESTING: STATE-BASED TESTING

- State-based testing focuses on comparing the resulting state of a class with the expected state.

☞ **Derive test cases using state machine diagram for a class.**

- Derive a representative set of "stimuli" (events) for each transition (similar to equivalence testing).

- Check the attributes or links of the class after applying a "stimuli" to determine if the specified state has been reached.

☞ **We first need to put the class in the desired state before applying the "stimuli".**

REGRESSION TESTING

- **Whenever you find a bug:**
  - Reproduce it (before you fix it).
  - Record the input that produced that bug.
  - Record the correct output.
  - Put everything into the test suite.
  - Then, fix the bug and verify the fix.

- **Why is this a good idea?**
  - It helps to populate the test suite with good tests.
  - It protects against changes that reintroduce the bug.

☞ **It happened once, so it might happen again!**