Soft. Quality Assurance — Lecture 01

18.06.22

When is software ready for release ?

Software bug → functional deficiency

(a function not working as it should be)

faulty software ⟹ monetary loss, reputation loss, legal reprocussions

What is "good" software ? — defined by quality metrics

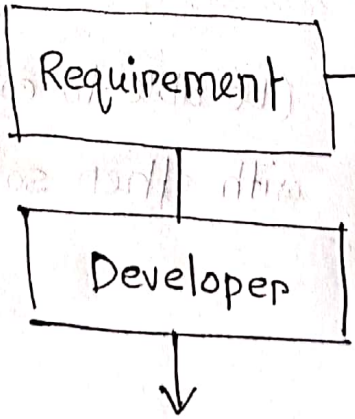
- dependability
- performance
- scalability
- availability
- security

Software Engineering

↳ idea about software, life cycle, how to develop

System Design

↳ designing a system from scratch keeping requirements in mind



(check requirements)

Quality Assurance Engineers / Testers

Core idea of testing / QA:

"Verification and Validation"

When is software ready for release?

- can't find anymore bugs
- finished testing
- high quality

Quality software — runs quickly

Quality attribute * dependability

- availability
- modifiability → (be able to change + add new things)
- testability
- Interoperability (be able to communicate with other softwares properly)
- usability
- performance
- security
- scalability

- resilience
- supportability

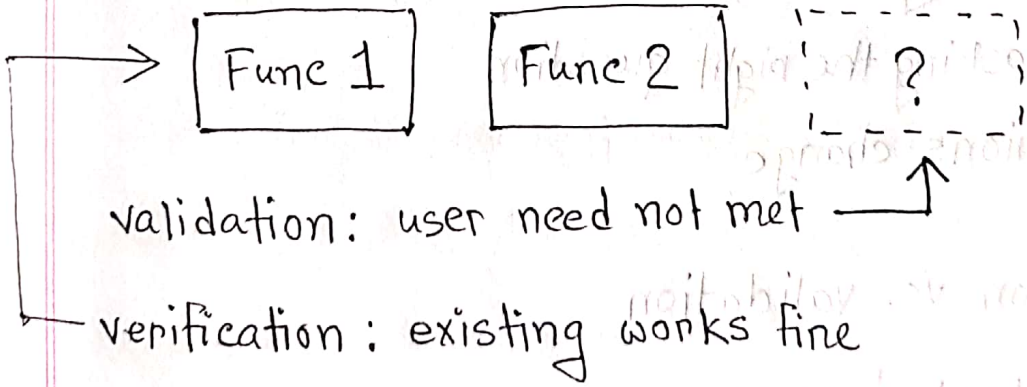
Verification → check if requirements are fulfilled
set by system designer

validation → check if USER requirements are fulfilled
- meet customers true requirements, needs, expectation

by Barry Boehm

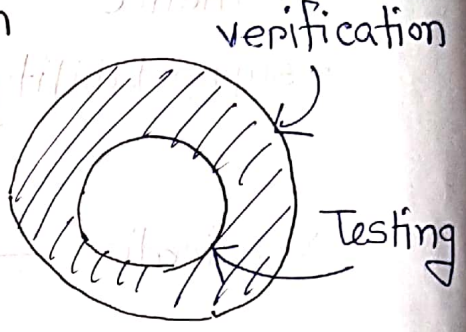
verification: "are we building the product right?"

validation: "are we building the right product?"



Soft. Quality Assurance – Lecture 02

Testing is a means of verification



but also,

– detailed design & high level arch

– design & requirements

Testing:

system under test

Test Input → SUT → output → Test oracle (expected out)

verdict (pass/fail) ←

validation → much harder than verification

- not asking the right person
- not asking the right question
- opinions change

verification vs. validation

Required level of V&V:

- software purpose
- user expectation

- marketing environment

When does V&V start?

- from the start-/inception

static verification: verifying without running code

↳ code review

dynamic verification: after running code

- testing
- ~~nan~~ fuzzing
- taint analysis

cheaper and can be automated but all possible inputs cannot be tested.

The trade-off game: better, faster, cheaper (pick 2)

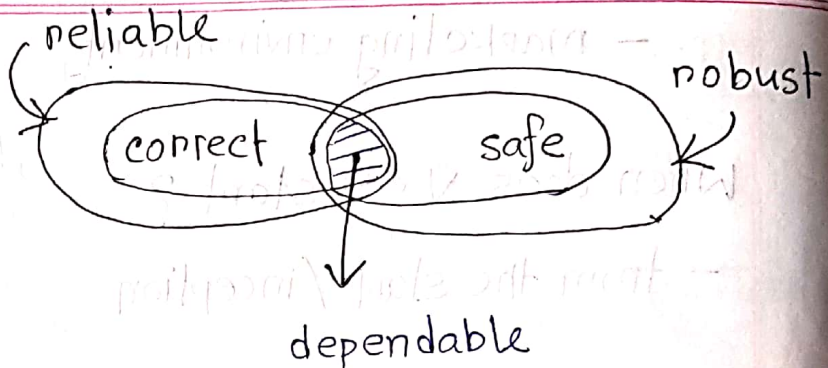
perfect verification — not possible

must accept some inaccuracy

- pessimistic inaccuracy
- optimistic inaccuracy
- property complexity

Dependability:

- correct
- reliable
- seafe
- robust



safety → ability to avoid | hazards |

defined undesirable situation

* Generally serious problem

Metrics of ~~availability~~ reliability:

- POFOD

Soft. Quality Assurance — Lecture 03

25.06.22

Quality attributes and Measurment:

dependability — Reliability
— Correctness
— Safety
— Robustness

correctness is hard to prove if requirement is strong.
safety can be measured but not effective.
Robustness works in environments that is not normal
not concerned for normal situations.

correctness is binary. not a good measure.
30% correct → no such thing

Reliability can be measured!

Probality of failure-free operation for a specified time in a specified environment for a given purpose

— specified time: fixed period

10 hours, 1000 request, 10 failed → 99% success
1% failure rate

Specified environment:

reliability can differ from user type to user type

Removing Bugs

Removing X% of faults ≠ X% of improvement in reliability

Example— removing 60% bug led to 3% improvement

* Reliability can be quantified and measured

Software → design wrong
Hardware → design correct, component wrong
} when bug found

$$\text{Availability} := \frac{\text{uptime}}{\text{total time observed}}$$

1 day = 1440 minutes

availability, A = 90% = 0.9 → down 10% time

= 144 hours

≈ 2.5 hours

Industry grade → 5 to 7 nines ⇒ 0.99999
0.999999 etc

0.9 → 144 minutes

0.99 → 84 seconds

0.999 → 8.4 seconds

0.9999 → 0.84 seconds

downtime = 10 hours

total time 365 day

≈ 8000 hours

$$Ava. = \frac{8000 - 10}{8000}$$

Probability of Failure on Demand (POFOD):

$$POFOD = \frac{\text{failures}}{\text{requests over observed time}}$$

POFOD = 0.001 means,
1 failure every 1000 requests.

$$POFOD = \frac{\text{failed attempts}}{\text{total attempts}}$$

Rate of Occurance of Fault (ROCOF):

$$ROCOF = \frac{\text{number of failures}}{\text{total time observed}} = \frac{\text{failed attempts}}{\text{total time}}$$

100 failures in 10 hours

$$ROCOF = \frac{100}{10} = 10 \text{ failures/hour}$$

(request frequency এর উপর depend করে।)

Mean time Between Failures (MTBF)

Time 10:00 12:20 14:00

Failure 1 ⟷ 2 ⟷ 3

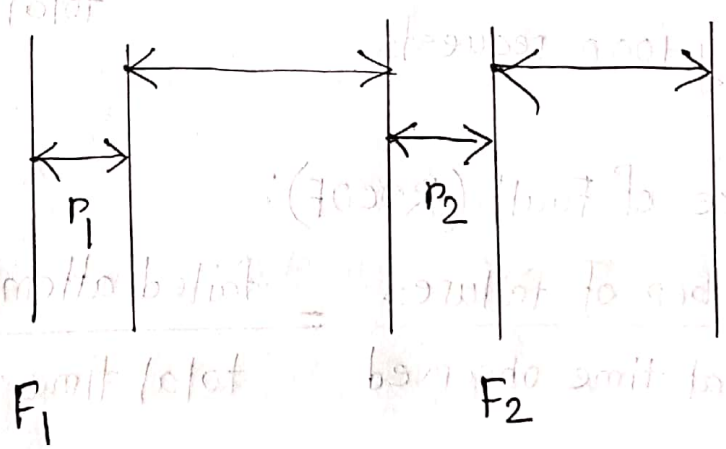
2:20 2:40

$$MTBF = \frac{2.33 + 2.67}{2} = 2.5 \text{ hours}$$

Probabilistic availability

↳ alternate definition

Probability that system will provide a service within required bounds over a specified time interval



$$\text{Availability} = \frac{MTBF}{MTBF + MTTR}$$

Example 1 : $\frac{(35+5)}{10,000} = 0.0004$

Example 2 : $\text{RoCOF} = \frac{6}{144}$

$$\text{POCOF} = \frac{6}{6000,000}$$

Example 3 : $1 \text{ year} = 365 \times 24 = 8760$

$\therefore$ total failures $= 8760 \times 0.001$

$= 8.76$

$\therefore$ downtime $= 8.76 \times 3 \text{ hr} = 26.28 \text{ hr}$

$\therefore$ Availability $= \frac{8760 - 26.28}{8760}$

$= 99.7\%$

RoCOF $= 0.001$ fail/hour

$= 0.001 \times 8760$ fail/year

$= 8.76$ fail/year

Example 4: $POFOD = \frac{64}{972} = 0.065$

$$ROCOF = \frac{64}{7 \times 24} = 0.38 \text{ / hour}$$

$$= (0.38 \times 8) \text{ / } 8 \text{ hours}$$

$$= 3.047 \text{ / } 8 \text{ hours}$$

$$\text{Availability} = \frac{168 - (37 \times 2)/60}{168}$$

$$= 0.9926 = 99.26\%$$

downtime $= (37 \times 2)/60 = 1$ hour 14 min

can't calculate MTBF

Soft. Quality Assurance — Lecture 05

02.07.22

Test Oracle → A predicate/function that determines whether a program is correct or not.

Oracle Information

Embedded information used to judge

Oracle Procedure

```
if (actual value != expected value){
    fail(...);
}
```

03 types of oracle
- expected value oracle
- property based oracle
- implicit oracle

expected value: checks with a stored expected value

property based: checks whether a defined property is true or not. → checking sorted array.
- usually written at "function" level
- limited ~~num~~ by number & complexity of property

implicit oracle: checks properties expected from system.

doesn't envolve logic; just runs the system and checks different properties.

Testing stages :

class → unit → subsystem (API) → system

The V-model of development

Architectural design → graphical prototype

Detailed design → defining the classes

Unit testing :

testing smallest 'unit' that can be tested

oracle → expected value oracle

Integration Testing (Subsystem)

checking system APIs

subsystem ⟹ API

GUI testing
Exploratory testing
} system tests
↳ reflects end-to-end user journeys

Based on user scenarios/journeys

- applying for leave
- submitting assignment

⇒ done by human or machine

GUI → specific (like DFS)

Exploratory → open-ended (like BFS)

Testing Percentage

Unit test → 70% (bulk/maximum)

system (Integration) test → 20%

GUI/exploratory test → 10%

Acceptance Testing

- Alpha team testing → small group of people/experts
- Beta team testing → In the wild/general people
- Formal acceptance testing

Formal acceptance testing

↳ runs like a loop with multiple iterations

developer & customer must negotiate

Soft. Quality Assurance — Lecture 06

05.07.22

System Testing :

- tests the integration of all the units as a whole
- test through different interfaces

Interface type
- Parameter Interface
- Procedural Interface
- Shared memory Interface
- Message Passing Interface

System Level Test case : How to create

1. Identify an Independently Testable Function
2. ↳ Identify choices
3. ↳ Identify Representative Input Values
4. ↳ Generate Test case Specifications
5. ↳ Generate Test cases

} 5-step method

Class Test #01

19th July, 2022

Slides no. 01, 02, 03

Software Quality Assurance — Lecture 08

23.07.22

Creating system level tests:

→ identify/isolate independetly testable function

↳ identify choices

↳ idify representative input values

↳ generate test case specifications

↳ generate test cases

```
int getMax(int a, int b)
```

int a — a > 0, a = 0, a < 0

int b — b > 0, b = 0, b < 0

choice: the value of the integer (for this case)

ch representative values

also, a > b, a = b, a < b

test cases getMax(a>0, b>0, a>b)

specification " (a>0, b>0, a<b)

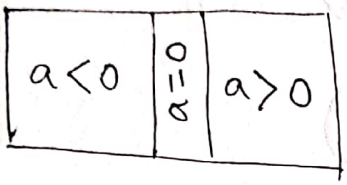
"

generate test case

↳ getMax(5, 10)

" (5, -10)

Input Partitioning → dividing the input space



03 partitions

* equivalence class

substr (string str, int index)

substr ("Assurance", 4) = "rance"

possible partitions : index > 0

index = 0

index < 0

length of str > ø index

length of str = ø index

length of str < ø index

getBuildingType (floor layout, country)

South Asia	offices
Europe	Apartments
Africa	Shops

* timing, operating systems, data structure

What are the input partitions for:

max (int a, int b) returns (int c)

Forming specification (int N, list A)

insert Postal code (

— Choice: int N

5 digit int between 10000 and 99999

rep values: <10000, 10000-99999, >100000

— Choice: list A

list of length 1-10

rep values: empty list

list [1]

list [2-10]

list [>10]

insert (10000-99999, list [1])

test: insert (56789, {12345})

Soft. Quality Assurance — Lecture 10

26.07.22

Computer configuration example

Check (Model, configuration)

```
Model {
    string modelNo;
    List <Slot> slots_required;
    List <Slot> optionalslots;
}
```

```
configuration {
    List <Slot, Component> mapping
}
```

```
Model {
    modelNo : "ABC-1234",
    reqSlots : {
        slot1,
        slot 2,
        slot 3
    }
    optionalSlots : {
        slot1,
        slot 2
    }
}
```

```
Configuration {
    mappings : {
        (slot1, comp1),
        (slot1, comp2),
        (slot3, comp3)
    }
}
```

total test case

configuration $3^7 \times 5^2 \times 4$

$= 2,18,700$

Removing constraints

choice: str len

len = 0

len = 1

len >= 2

choice: index

value < 0 error

value = 0

value = 1

value >= 2

choice: str contents

contains letters and numbers

contains special characters

empty

Single

error + single: 3 x 2 x 3 + 1 + 1 = 20

Zero len = True

len option	str content	ind
1	1	3

Zero len = False

len option	str content	ind
2	1	3

IF constraints

total: 1 x 1 x 3 + 2 x 1 x 3 + 1 + 1

= 11

RS many	OS many	
0	0	$2 \times 1 \times 1 \times 1 \times 1 \times 1 \times 2 \times 2 \times 3 \times 3 = 72$
0	1	$2 \times 1 \times 1 \times 1 \times 1 \times 1 \times 2 \times 3 \times 3 \times 3 = 36$ ~~36~~ 108
1	0	$2 \times 1 \times 1 \times 1 \times 1 \times 1 \times 3 \times 2 \times 3 \times 3 = 36$ ~~36~~ 108
1	1	$2 \times 1 \times 1 \times 1 \times 1 \times 1 \times 3 \times 3 \times 3 \times 3 = 18$ ~~18~~ 162

total = 450

error + single = 14

464

total test cases → 464

Tuesday → class test 02 : slide 4 and 5

Software Quality Assurance — lecture 11

Combinatorial Interaction Testing

Bandwidth mode — 3

Language — 4

Fonts — 3

Advertising — 4

Screen size — 3

Full set = $4 \times 4 \times 3 \times 3 \times 3 = 432$

Cover all K-way interactions ($K < N$)

choice 1	choice 2	choice 3
R_1	L_1	K_1
R_2	L_2	K_2
R_3	L_3	K_3

2-way combination :

$R_1\ L_1\ K_1$
$R_1\ L_2\ K_{1.2}$
$R_1\ L_3\ K_3$
$R_2\ L_1\ K_2$
$R_2\ L_2\ K_3$
$R_2\ L_3\ K_1$
$R_3\ L_1\ K_3$
$R_3\ L_2\ K_1$
$R_3\ L_3\ K_2$

reduced to 9 test specification

Line	Indent	Para
L_1	I_1	P_1
L_2	I_2	P_2
L_3		

total $= 2 \times 2 \times 3 = 12$

2-way combination : reduced to = 6

$L_1\ I_1\ P_1$

$L_2\ I_1\ P_2$

$L_3\ I_1\ P_1$

$L_1\ I_2\ P_2$

$L_2\ I_2\ P_1$

$L_3\ I_2\ P_2$

2-way combination for 5 choices $(3 \times 3 \times 3 \times 3 \times 3)$

$A_1\ B_1\ C_1\ D_1$

$A_1\ B_2\ C_2\ D_2$

$A_1\ B_3\ C_3\ D_3$

$A_2\ B_1\ C_2$

$A_2\ B_2\ C_3$

$A_2\ B_3\ C_1$

$A_3\ B_1\ C_3$

$A_3\ B_2\ C_2$

$A_3\ B_3\ C_1$

$A_1 - A_4$

$B_1 - B_3$

$S_1 - S_3$

$L_1 - L_4$

$F_1 - F_3$

L_1 A_1 B_1 S_1

L_1 A_2 B_2 S_2

L_1 A_3 B_3 S_3

L_1 A_4

L_2 A_1

L_2 A_2 B_1 S_2

L_2 A_3 B_2 S

L_2 A_4 B_3 S

L_3 A_1 B_3 S

L_3 A_{12}

L_3 A_{23} B_1 S

L_3 A_{34} B_2 S

L_4 A_{41} B_2 S

L_4 A_{x2} B_3 S

L_4 A_3

L_4 A_4 B_1 S

Cl (1-3), P(1-2), C(1-3), Ao(1-2), At(1-2), F(1-2)

Cl_1 C_1 P_1 Ao_1 At_2 F
Cl_1 C_2 P_2 Ao_2 At_1 F
Cl_1 C_3

Cl_2 C_1
Cl_2 C_2 P_1 Ao_2 At_1 F
Cl_2 C_3 P_2 Ao_1 At_2 F

Cl_3 C_1 P_2 Ao_1 At_1 F
Cl_3 C_2
Cl_3 C_3 P_1 Ao_2 At_2 F

Flight data structure (Assignment)

flight code : BMN-0333
originity : DHK
Or depart : 29-07-22 11:00:00
dest : MUM
dest App : 29-07-22 12:00:00

Software Quality Assurance — lec 12

02.08.22

terminate Membership (set<student> club, Student s)

Parameter : club

choice : length of set ~~char~~ club

Parameter : s

choice : is student in club ?

choice : is student valid ?

* University of South Carolina

Agile method

↳ validation happens before completion

Software Quality Assurance

Quality attributes

- dependability
- availability
- modifiability
- testability
- performance
- security
- scalability
- Interoperability
- Usability

Test input → SUT → Output → Test Oracle → Verdict

Verification
- Static
- Dynamic
  - testing
  - fuzzing
  - taint analysis

Three dimensions of inaccuracy
- Pessimistic inaccuracy
- Optimistic inaccuracy
- Property complexity

Safe — no optimistic inaccuracy,
only accept if correct with respect to property

Sound — if the technique returns true only when the program does meet the property.

if, true = correct and technique is sound

$\Rightarrow$ also safe

if, true = incorrect and technique is sound

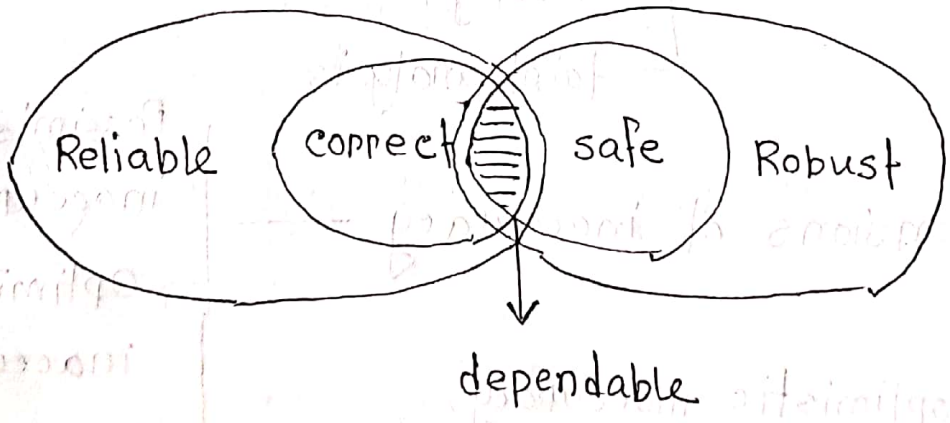
$\Rightarrow$ optimistic (allow)

$\Rightarrow$ pessimistic (disallow)

Complete — always returns true when the program does satisfy the property.

if true = correct then complete only admits optimistic inace.

dependable — correct, reliable, safe, robust



Probability of failure-free operation for a specified time in a specified environment for a given purpose.

$$\text{Availability} = \frac{\text{uptime}}{\text{total observed time}}$$

POFOD (Probability of Failure on Demand)

$$\text{POFOD} = \frac{\text{failed attempts}}{\text{total attempts}}$$

ROCOF (Rate of occurance of fault)

$$\text{ROCOF} = \frac{\text{failed attempts}}{\text{total observed time}}$$

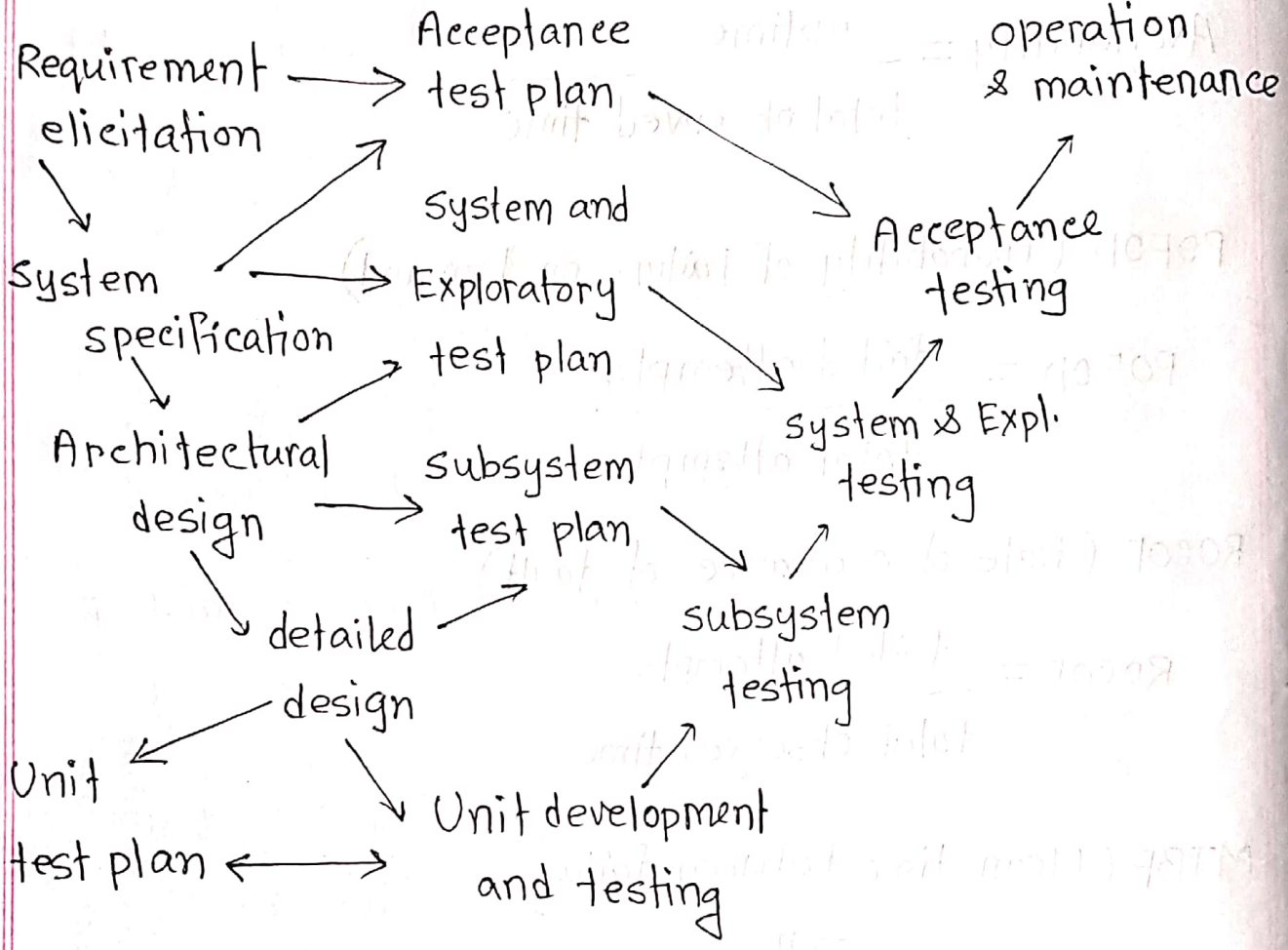
MTBF (Mean time between failue)

$$\text{MTBF} = \frac{1}{n} \sum_{i=1}^{i=n} \text{time\_between\_failure}_i$$

$$\text{MTBF} = \frac{1}{n-1} \sum_{i=1,j=2}^{n-1,n} \text{time between failuere } i \text{ and } j$$

MTTR → mean time to repair

$$\text{Probabilistic availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$



V-MODEL

```
if (On = Expected (On)) {
    pass();
}
else
    fail();
```

Question No. 01 :

download = 1 hour

during peak hours → 10 ~ 150

down — (avg) 2 times/week

3 min/fail

Soft. Quality Assurance — lecture 13

16.08.22

Unit Testing

Unit → A class

testing the smallest "unit" that can be tested

↳ often a class and its methods

Tested in isolation from all other classes

Test input = method calls

Test oracle = assertions on output/class variables

Unit tests should cover

- Set and check class variables
  setter and getter
- Each job performed by the class
  action/verb methods

Account

- name
- personnumber
- balance

withdraw()
deposit()
checkname()

we should also test constructors

Software Quality Assurance — lecture 14

20.08.22

Test scaffolding — The code written for test automation (not part of product; temporary)

Test scaffolding components — driver : test cases
harness
stub

Unit

```
class {
    var
    method()
}
```

```
testMethod()
{
    ... method(3,5)
}
```

driver substitutes for a main or calling program.

Harness : Initiallizing/setting up the environment.

Example — Wanting to withdraw 1000tk when balance is less than 1000. Harness : depositing <1000tk.

```
class Account {
    balance;
    personNum;
    name;
    withdraw(double amount);
```

withdraw(double amount) — Successfully withdrawn
Insufficient
Negative

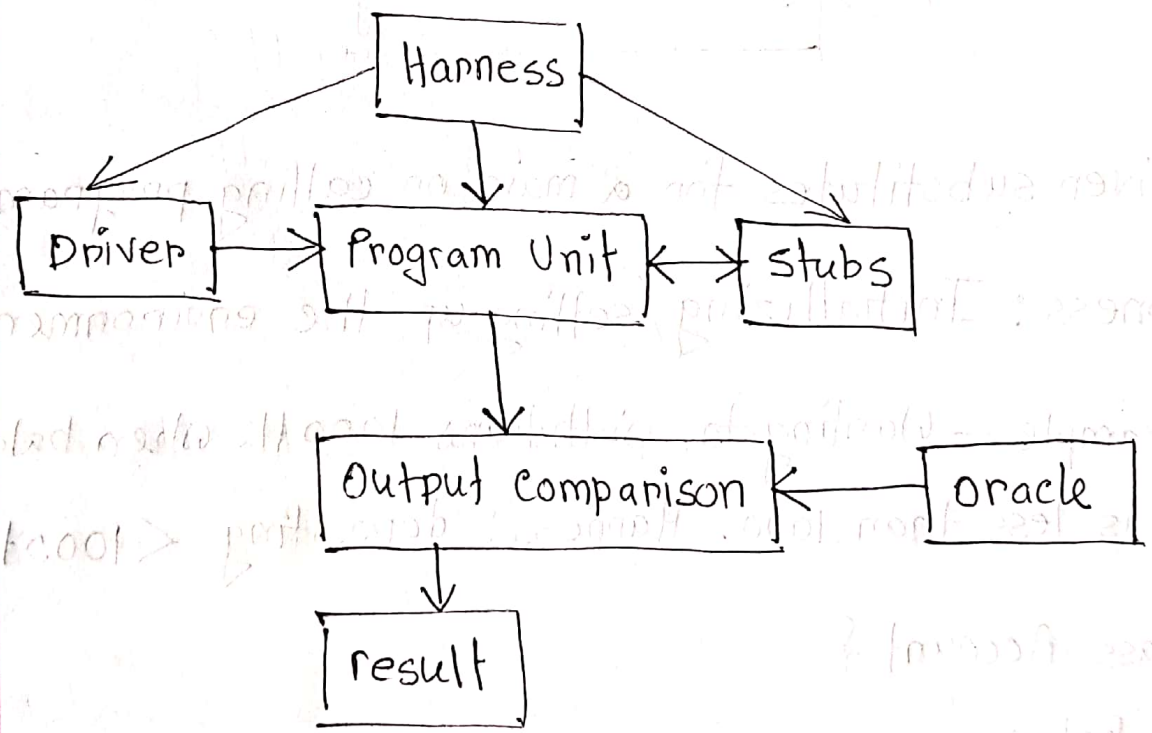
Account acc = new Account ("John", 100);

↳ Harness

withdraw (46.5) → Driver

We can use a mock object in harness for testing purposes.

Test Scaffolding – Flowchart



JUnit:

@Test annotation defines a single test

```
@Test
poublic void test<Feature or method name>_<Testing context>
                                                    (){
  // define Inputs
  try{ // Try to get output.
  } catch (Exception error){
      fail("Why did it fail?");
  }
  // other stuff
}
```

@BeforeEach annotation defines a common test initialization method

```
@BeforeEach
public void setUp() throws Exception {
   ....
}
```

Similarly, @AfterEach annotation

```
@AfterEach
public void tearDown() {...}
```

Also there is, @BeforeAll, @AfterAll

Before/after whole test execution

Some oracles:

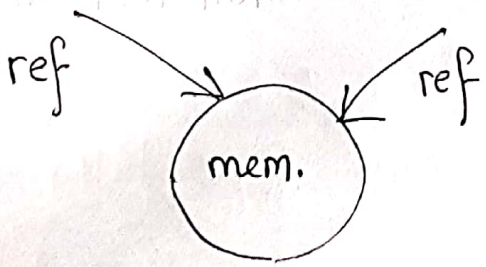
assertEquals(); → relies on .equals() method

assertFalse, assertTrue

↳ checks whether True/False or not.

assertEquals() uses .equals()

assertSame() uses ==



checks if objects are clones

↳ pointing to same memory location

assertNull(), assertNotNull()

Grouped assertions using assertAll()

```
assertAll ("person",
    { () -> assertEquals("John", person.getFirstName()),
      () -> asspt True( person. getAge > 30)) ;
```

↳ multiple assertions!

Software Quality Assurance — lecture 15

22.08.22

contains String (String Pattern)

Java Stream API

string str = "albumen"

str. contains ("a");
str. contains ("b"); → True

assert ~~That~~ (str. contains ("a"));

or assertTrue

AssertThat (str, both (——). and (——))

AssertThat (str, either (——). or (——))

allOf ( ) → all inside must be true

anyOf ( ) → any inside must be true

Unit Testing — Account

```
@Test
public void testWithdraw_normal (){
    // setup
    Account acc = new Account ("Mr Test",
            "1951018-0932", * 48.5);
```

```
// test steps
double withdraw = 16.0; // Input
account.withdraw(withdraw);
...
}
```

assertThrows() Throwable e = assertThrows(—

assertThrows { ———);

() -> {account.withdraw(toWithdraw);});

↳ catch this exception and compare message

assertEquals(e.getMessage(), "———");

```
AssertThrows ( exception,
    () -> {new ArrayList<object>.get(0); }
);
```

checks whether the code block throws the expected exception.

```
AssertThrows(
    IndexOutOfBoundException.class,
    () -> {new ArrayList<object>.get(0);}
);
```

First checks whether the exception class are the same or not.

Second returns the class so it can be saved in a variable.

```
Throwable exception = AssertThrows (___);
@Test
void testWithdraw_InsufficientBal(){
    Account a = new Account("MoI", "123", 50.0);
```

```
Throwable e = AssertThrows(
        InsufficientBalanceException.class,
        ()->{a.withdraw(80.0);}
);
assertEquals("Balance is less than withdrawal amount",
        e.getMessage(),
);
}
```

assertTimeout(——); → used to impose a time limit on an action. A Performance test.

NOT a correctness demo.

Class Object::variable → used to access a static variable of a class.

```
LinkedList list = new ArrayList();
LinkedList fakeList = mock(LinkedList.class)
```

— does not exist
— exist but not tested
— abnormal case

} when we need mock data

Software Quality Assurance — Lecture 17

Build scripts

- ANT
- Maven
- Gradle

} different frameworks

Build lifecycle

Validate → compile → Test → Package → Verify → Install → Deploy

Validate: project is correct and all necessary information available.

Target: collection of tasks you want to run in a single unit.

↳ can depend on other targets

```
<target name="deploy" depends="pacage">
    ....
</target>
```

independent targets are run first.

written in XML.

<property> tag can be used for variables.

```
<property name="sitename" value="hello.com" />
<target name="info">
  <echo> You are in site — ${sitename} </echo>
</target>
```

Output: You are in site — hello.com

(replace with value of property)

For <include> and <exclude>

** → sub-directory <pathelement>

* → partial match <fileset>

starting / doesn't matter

Next Saturday — class test (Unit Testing)

Soft. Quality Assurance — lecture 18

Models and Software Analysis

- Before and while building products, engineers analyze models to address design questions
- software models capture different ways that the software behaves during execution.

Model driven development

① Generate code from model

② Generate test cases from model

Finite State Machines

state space ⟶ set of states

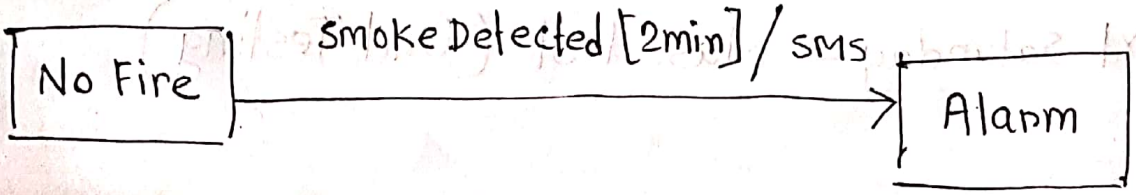
state $\xrightarrow{\text{action}}$ state (transition)

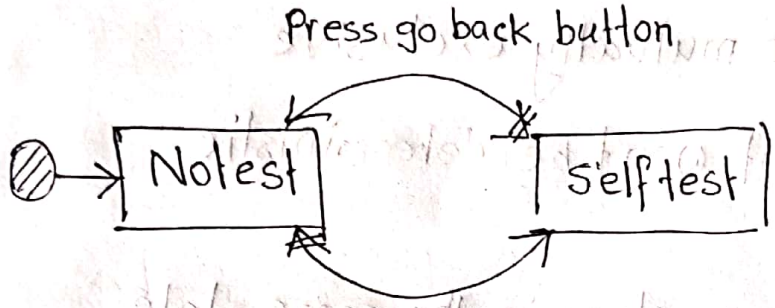
state

transition

labels:

event [guard] / activity





Event: an input that occurs at a defined time

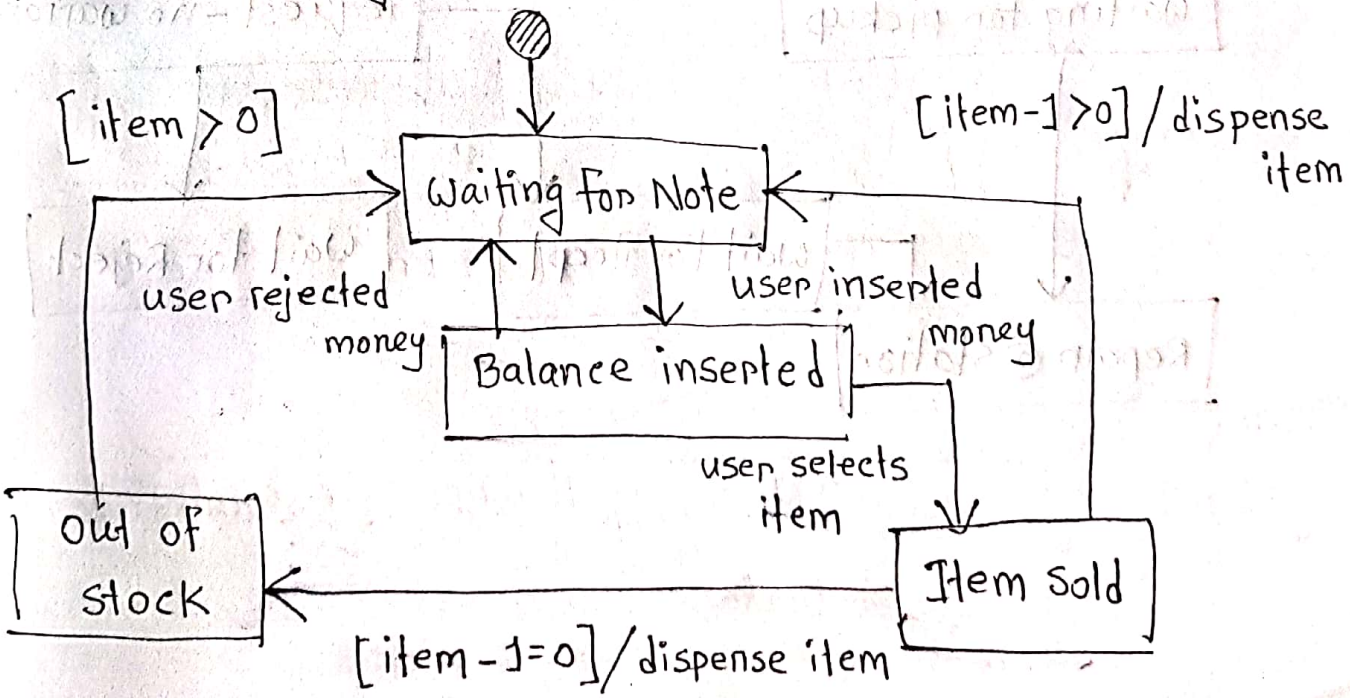
All three are optional.

Missing activity — No output from this transition

Missing Guard — No condition

Missing event — change state immediately

Simple vending machine:



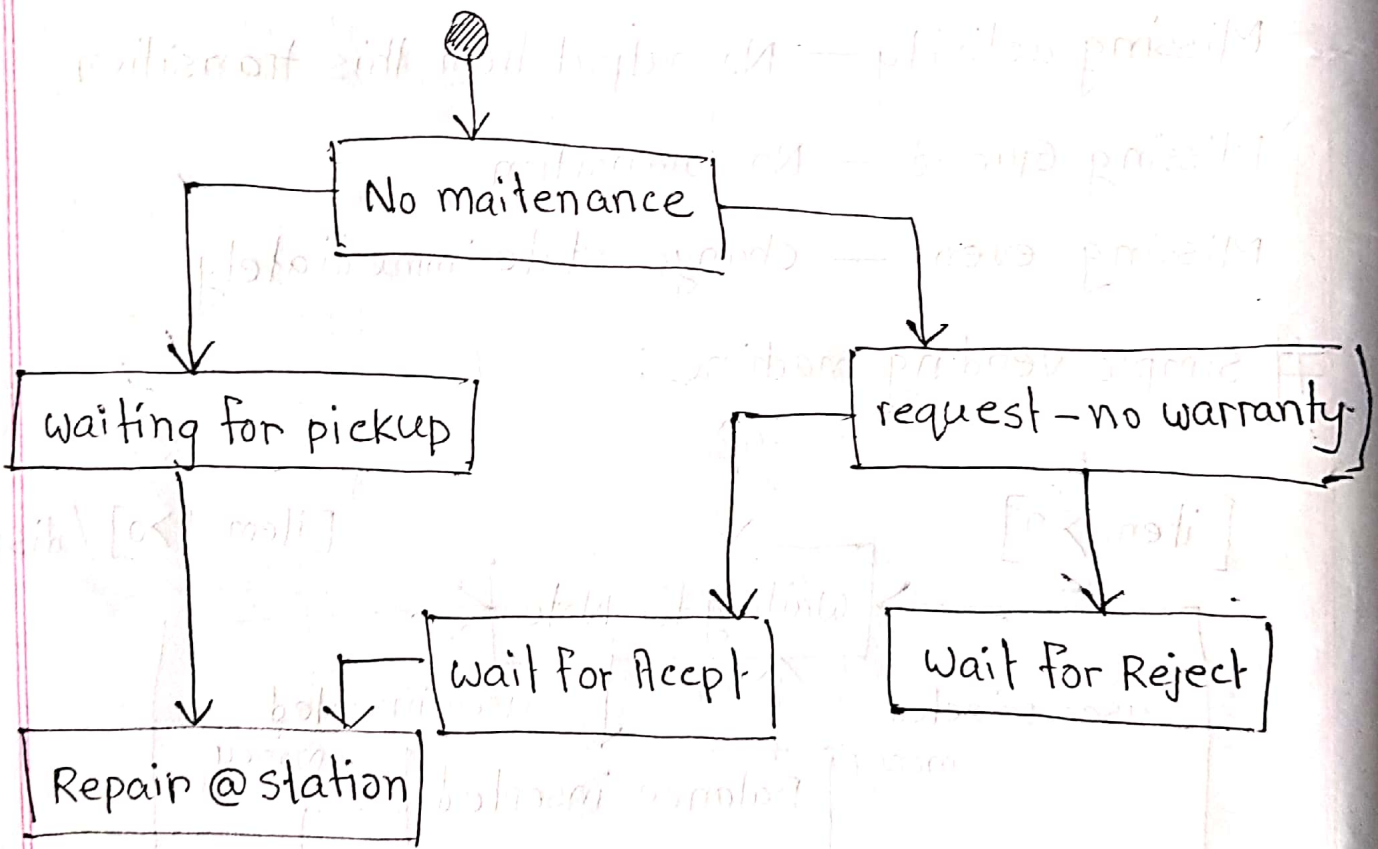
Guards must be mutually exclusive

↳ otherwise it won't be deterministic

Internal Activity : stays in the same state

special activity — entry, exit

not re-triggered



Software Quality Assurance — lecture 19

06.09.22

Slot

Model ⟶ String : "PC-1"

Slot ⟶ String : "LG 22\" monitor"

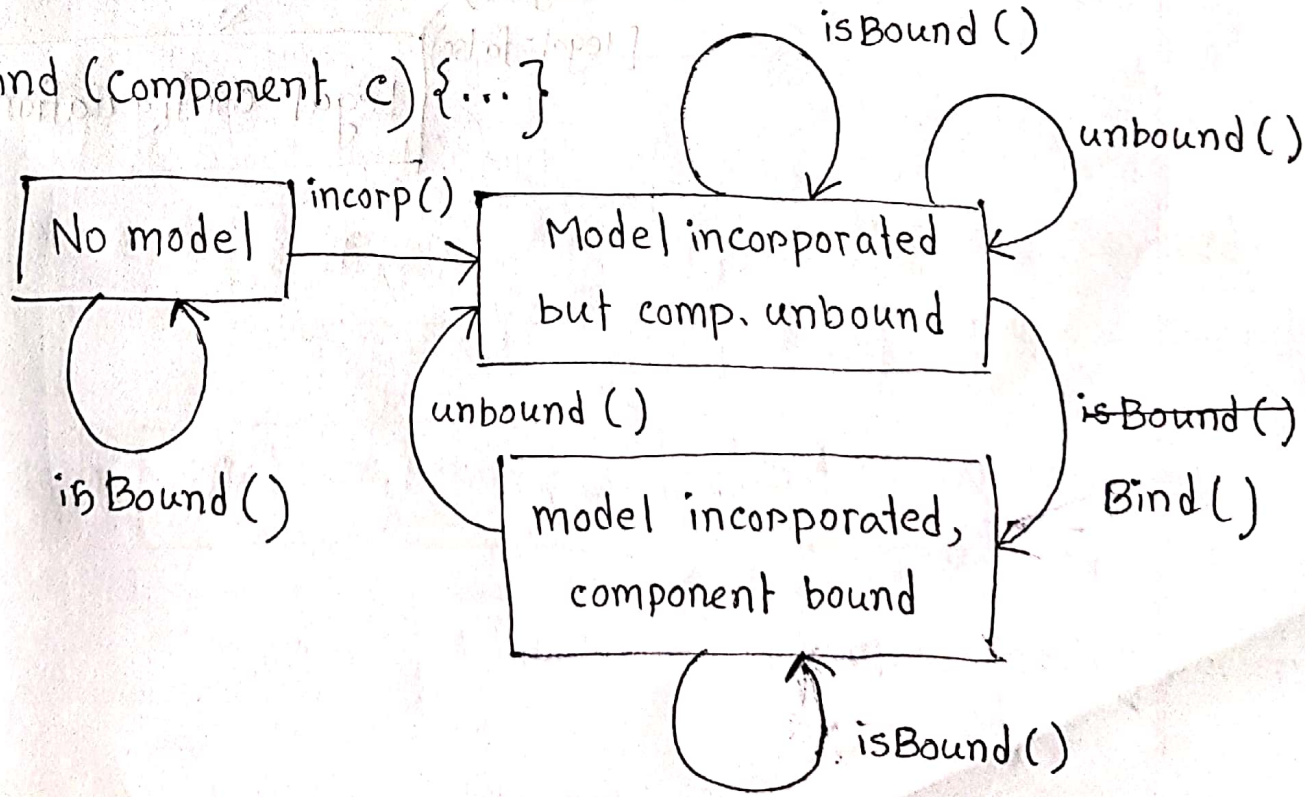
Required ⟶ Boolean : True

incorporate (Model m) ⟶ m: <slot>

⇒ adds to slots list

```
incorporate (Model m) {
    m.slots.add (this);
    this.model = m.modelID;
}
```

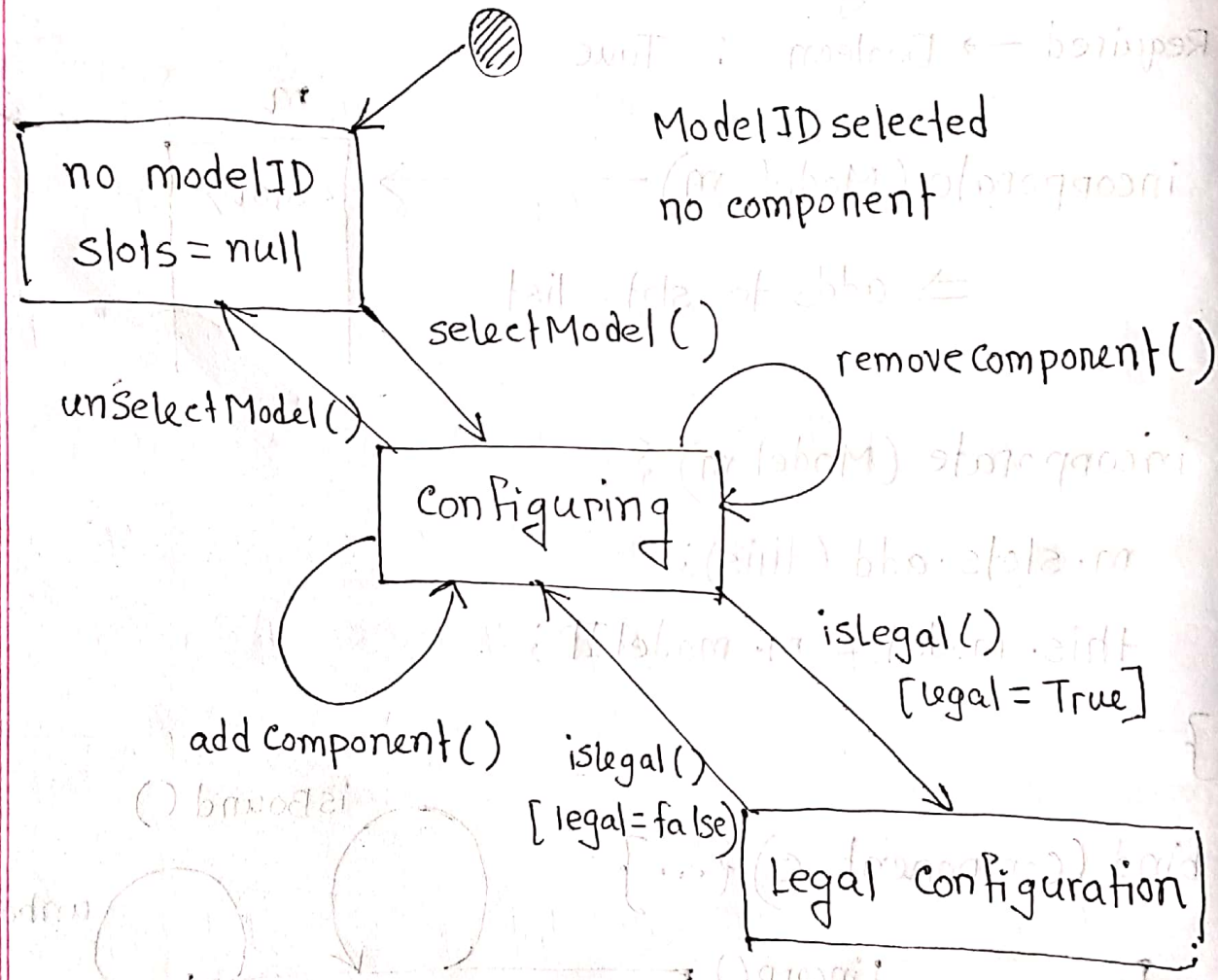
bind (Component c) {...}



Model

String modelID

List <Slot> slots



Software Quality Assurance — lecture 21

10.09.22

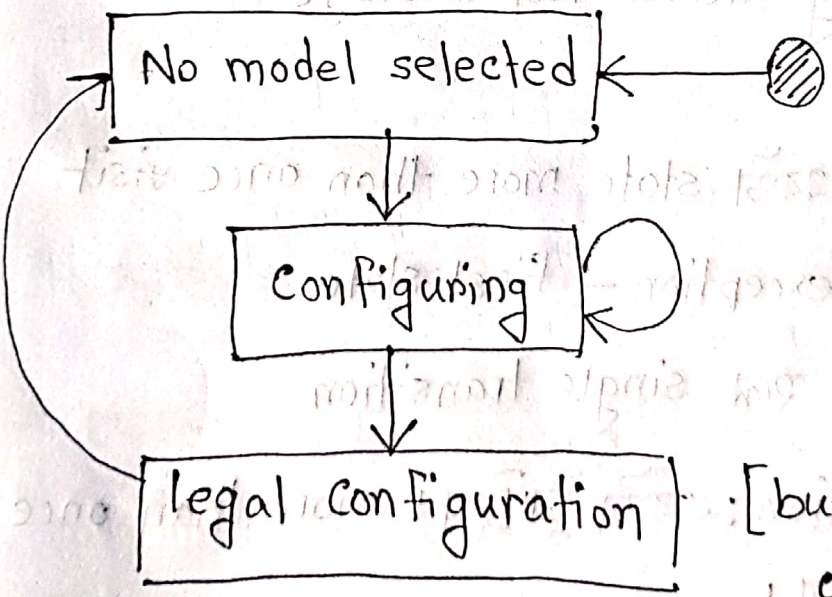
Test case generation — from finite state machine

- Follow the paths
- State coverage
  - * Each state must be covered at least one
  - * num. of covered states / number of states

∵ Easy to understand but low fault-revealing power

selectModel()
add component (s_1, c_1)
isLegal() // legalConfig = true
deselectModel()

} Test Case – I



all states covered.

coverage = $\frac{3}{3} = 1$

[but all events not covered.]

fault-revealing power ↓

- Transition coverage
  * covering each transition at least once
  * ⊌ num. of covered transition / total transition

Goal is overall coverage. Not per test
multiple test cases together should cover.

Path coverage → many possible
loop ⟹ infinite possibility

① single state path coverage
② single transition path coverage
③ Boundary interior loop coverage

single state : একটা state more than once visit করবে না। exception – first state
single state হলে single transition

single transition : একটা ট্রানজিশন more than once visit করবে না।

Boundary Interior Loop → n times একটা loop কে cover করবে।

Final state verification

① Scenario → model
② class → model
③ coverage

} ⊆

If model satisfies the requirement, then program should as well.

Formulating requirements as logical expressions

* propositional logic
* temporal logic

H(x) = Person x is hungry → 0/1, but can change with time

Two properties

- safety property : no bad state (always in good s.)
- liveness property

- Linear Time Logic (LTL)

X (next) ⟶ what will happen next

G (globally) → will always happen

F (finally) → at the end it'll happen

U (until) ⟶

Hunger = F
Burger = T

R (release) ⟶

Hunger = T
Burger = T

→

Hunger = F
Burger = T

(sad && !notRich) → X (sad)

(hungry && haveMoney) → X (orderPizza)

send → F (receive)

winLottery → G (rich)

Soft. Quality Assurance — lecture 22

12.09.22

A (Hunger U Burger) → I guarantee

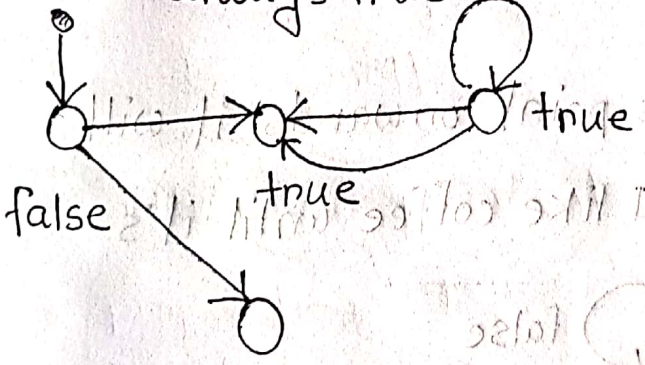
A (Hunger W Burger) → I cannot guarantee.

All A (—)

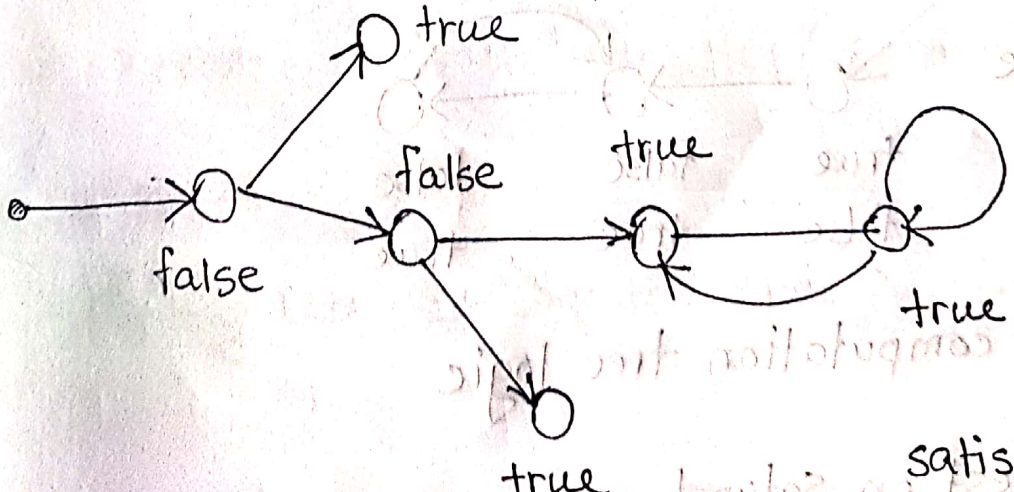
Exists E (—)

AF (EG coffee) all paths will eventually lead to some

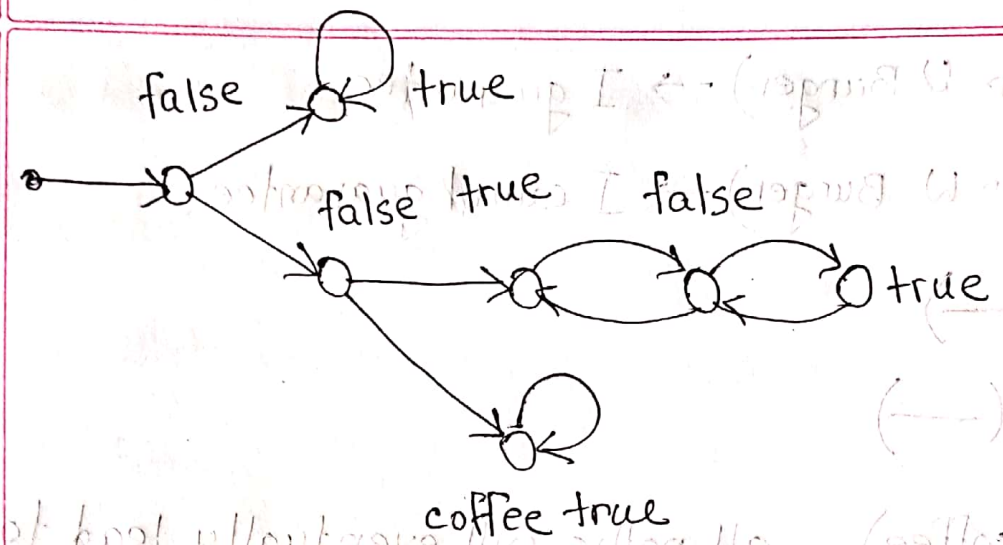
EG coffee : there exists some path where coffee is always true.



coffee = false" true



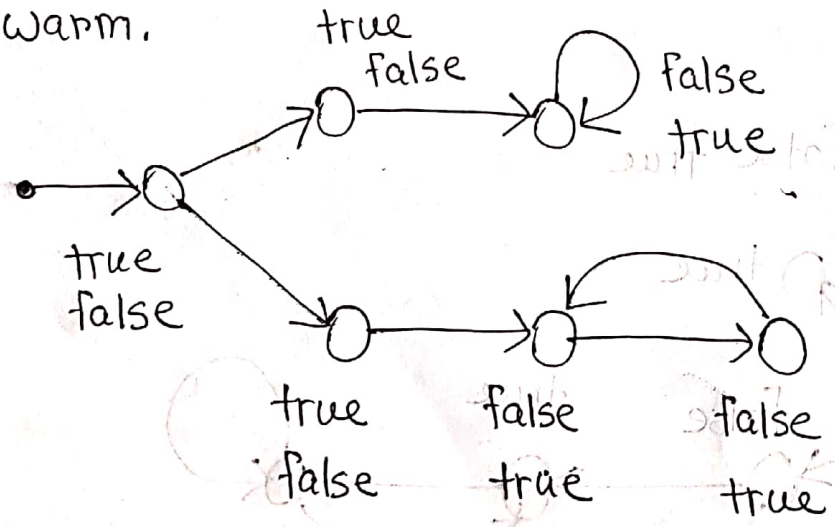
satisfies



AG(coffee U warm)

all paths globally

- For all paths from this point onwards it will always be true that I like coffee until it's warm.



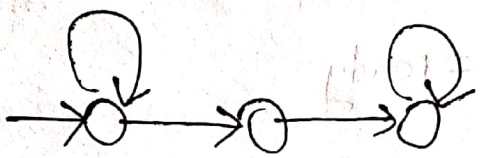
CTL : computation tree logic

Class test on Saturday

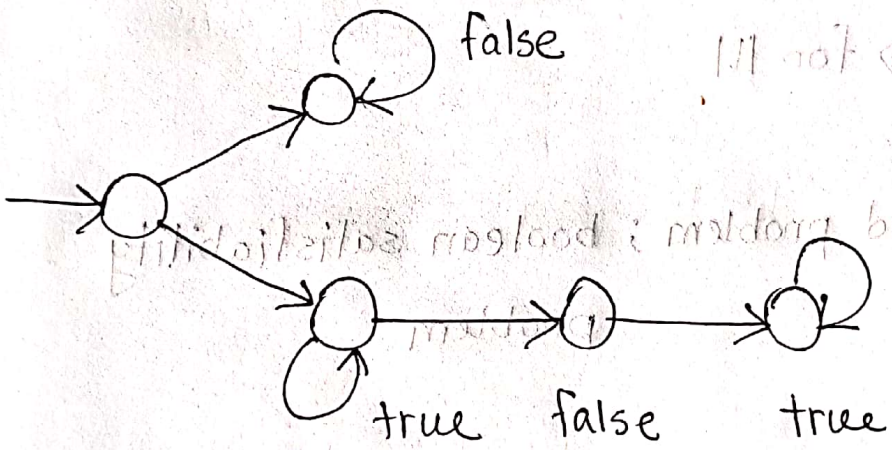
it is always possible (AG) to reach a state (EF) where we can reset.

AG (EF reset)

but G (F reset) also works!



(for this)



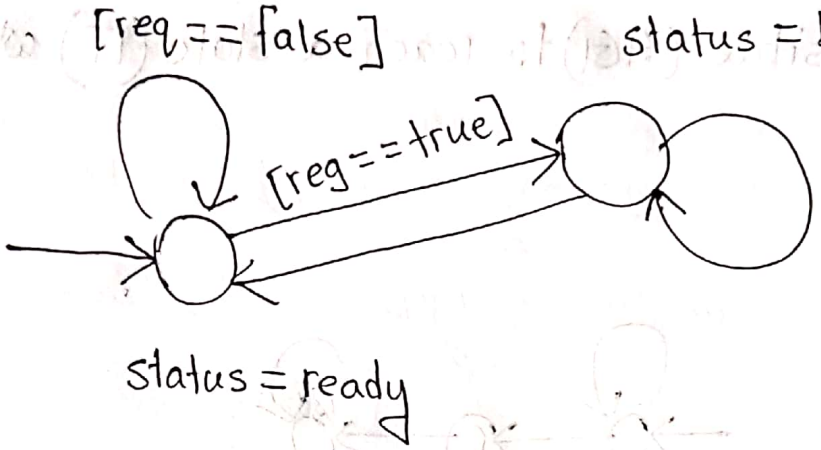
G (F reset) X

AG (EF reset) ✓

G (F reset) equivalent to AG (AF reset)

NuSMV → symbolic model checker

Binary Decision Digrams (BDDs); CTL or LTL



AG (request → AF (status = busy))

spec ⇒ for ctl

LTLSPEC ⇒ for ltl

3sat NP hard problem ; boolean satisfiability problem

Software Quality Assurance + Practice

Linear time logic (LTL)

X (next) → in the next state

G (globally) → in all future states

F (finially) → eventually there will be

U (until) → T, F ⇒ F, T

R (release) → T, F ⇒ T, T. ⇒ F, T

Finite State machines :

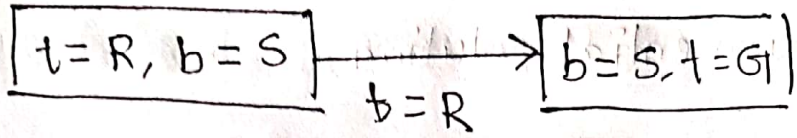
event [guard] / activity

Software Quality Assurance — lecture 23

17.09.22

```
init (traffic_light):= RED;
next (traffic_light) := case
 traffic_light = RED & button = RESET :
     GREEN;
 traffic_light = RED :
     RED;
 traffic_light = GREEN & button = SET :
     {GREEN, YELLOW}
 traffic_light = GREEN :
     GREEN;
 traffic_light = YELLOW :
     {YELLOW, RED};
 TRUE : {RED};
esac;
```

case closing



Safety property : traffic light green হলে pedestrian light walk হবে না।

$AG(\text{traffic-light} = GREER \rightarrow \text{ped-light} \neq WALK)$

liveness property : traffic light red হলে green হবে।

$G(\text{traeffic\_light} = RED \rightarrow F(\text{traffic\_light} = GREEN))$

Proving properties

- search state space for property violations
- violations give us counter examples
- Implications of counter example
  - property is incorrect

Search Based on SAT

- Express properties in conjunctive normal form :

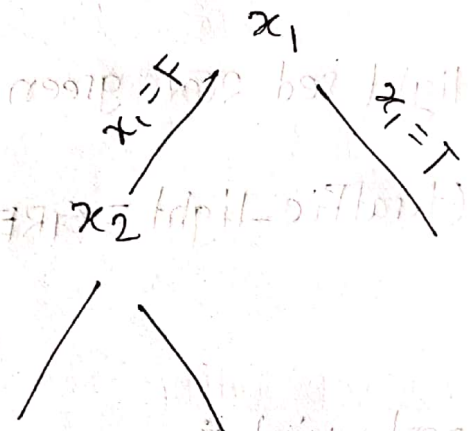
$f = (!x2 \,||\, x5) \,\&\&\, (x1 \,||\, !x3 \,||\, x4) \,\&\&\, (x4 \,||\, !x5) \,\&\&\, (x1 \,||\, x2)$

NP- complete : exponential solution

but smarter search — branch & bound

(boolean satisfiability)

$\varphi = (\neg x_2 \vee x_5)$



DPLL algorithm

clause satisfy করলে সেটা বাদ দিয়ে দিবো।

Software Quality Assurance — lecture 24

20.09.22

JUnit / Unit testing — 12
Buid scripts — 08
Finite State M. — 10
FSM verification — 10

} tentative marks distribution

Class Test ~~02~~ 03

1. Double CtoF(Double C) $-273.16 \leq C < 0$ // case1

$C < -273.16$ // case2

```
@Test
public void testCtoF_normal() {
    double c = -20.5;
    double out = CtoF(c);
    assertEquals(□, out);
}
@Test
public void testCtoF_error() {
    double c = -275.0;
    Throwable e = AssertThrows(ImpossibleValueEx.class
        ()=> {CtoF(c);});
```

```
        assertequals (" e.getMessage(), " Input tempera.
            cannot be below absolule zero." );
}
```

```
CustomObj a = 5;
CustomObj b = 5;
```
} Custom class & string এর ক্ষেত্রে

```
a == b; ⟶ False
a.equals(b); ⟶ True
```

```
assertAll ("person",  ⟶ message (when failed)
    () → assertEquals ("John", person.getFirstName()),
    () → assertEquals (39, person.getAge())
);
```

Mocking :

```
Vector <Student> stdVect = mock (Vector.class);
Student someone = new Student ("MoI", 1234);
```

when (stdVect.get(99)).thenReturn("someone");

when (mockStd.getCourses()).thenReturn("SQA");

not implemented!

class test – 04:

1. a) G1 (cooking = TRUE → F(cooking = False))

b) G1 (cooking = TRUE U (button = start ∧
door = closed ∧
timer > 0))

c) G1 (cooking = True ∧ Button = stop → X (cooking
= False))

Software Quality Assurance — Practice

1. Lecture 07

- Unit testing ☐
- Writing test case (3*) ☐
- JUnit library func and assertion (1*) ☐
- Unit test writing best practice ☐
- Mockito ☐
- Build life cycle ☐
- Build scripts in Ant framework ☐

2. Lecture 08

- State Machines ☐
- Components of state machine, transition ☐
- Designing from scenario (1*) ☐
- Designing from class diagram (2*) ☐
- coverage criteria (state / transition / path) (1*) ☐
- Designing test case with coverage (1*) ☐

3. Lecture 09

- Finite state verification ☐
- Temporal logic ☐
- Expressing requirements in CTL/LTL (2*) ☐
- Safetiness / liveness property (1*) ☐
- NuSMV implementation ☐
- Algorithms to FSV - exhaustive search ☐
- Reducing to boolean SAT problem ☐
- Branch and bound ☐
- DPLL ☐

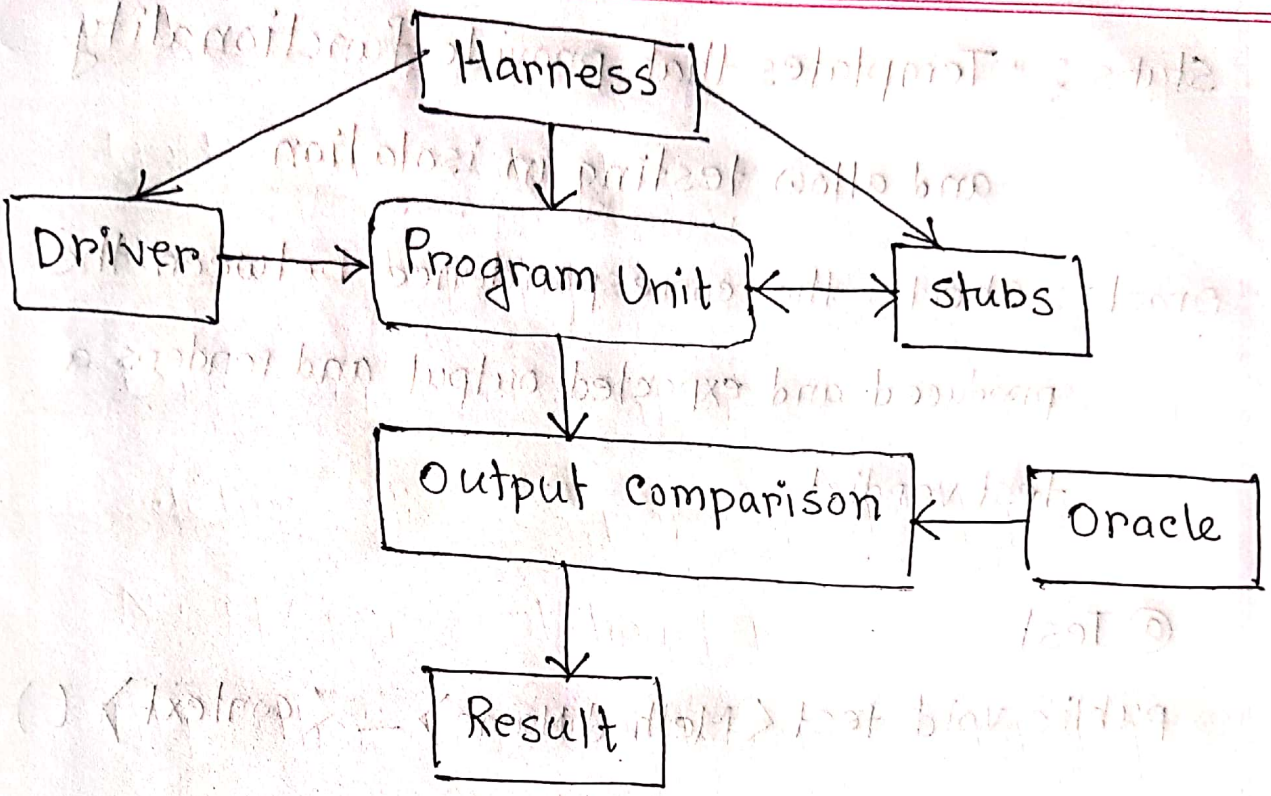
Test Scaffolding

a set of programs written to support test automation

allows:

- testing before all components complete
- testing independent components
- control over testing environment

* A driver substitutes for a main or calling program (Test cases are drivers)
* A harness substitutes for part of the deployment environment
* A stub (or mock object) substitutes for system functionality that has not been tested.
* Support for recording and managing test execution



Test Scaffolding Diagram

Driver :
- Initializes objects
- Initializes parameter variables
- Performs the test
- Performs any necessary cleanup steps

Harness :
- Simulate the execution environment
- can control network conditions, environmental factors, operating systems

P.T.O

Stubs : · Templates that provide functionality and allow testing in isolation

Oracle : Checks the correspondence between the produced and expected output and renders a test verdict.

```
@Test
public void test<Method Name>_<context>(){
    // test code
}
```

@BeforeEach – Shared Initialization ← Registration

@AfterEach – Teardown Method ← log out

@BeforeAll

@AfterAll

assertEquals, assertArrayEquals

assertFalse, assertTrue

assertSame, assertNotSame

Same → ==

Equals → .equals()

assertNull, assertNotNull

Grouping — assertAll

assertThat("albumen", both(containsString("a"))
.and(containsString("b")));

(Arrays.asList("one", "two"), hasItems("one",
"three"));

everyItem(containsString("n"))

"good", allOf(equalTo("good"), startsWith("good"))

not(allOf(...))

anyOf(...)

CombinableMatcher.<Integer> either(equalTo(3)).
or(equalTo(4))

```
Throwable exception = assertThrows (
        IndexOutOfBoundException.class,
    () -> { new ArrayList<Object>.get(0); } );
assertEquals (" ", exception.getMessage)

assertTimeout ( ofMillis (10),
    () -> { order.process (); } );
String str = assertTimeOut ( );
assertEquals (" ", str);
```

Best Practice

- Use assertion instead of print (print always passes)
- if code non-deterministic, test should be deter.
- test negative scenarios and boundary cases in addition to positive

(Boundary case → extreme value)

- Test one unit at a time
- No unnecessary assertions
- Make each test independent of each other
  (use @BeforeEach and @AfterEach to set and clear state)
- Create unit tests to ~~too~~ target exceptions

Mocking

```
LinkedList mList = mock(Linkedlist.class);
when(mList.get(0)).thenReturn("First");
when(mlist.get(anyInt())).thenReturn("element");

Therm mTherm = mock(Therm.class);
when(mTherm.get()).thenReturn(98);
WheatherData wData = new WheatherData();
wData.collect(mTherm);
```

assertEquals (98, wData.temp);

Build systems

Build systems ~~each~~ ease the process of building, running tests, packaging and distributing by automating as much as possible.

Validate → Compile → Test → Package

↓

Deploy ← Install ← Verify

Validate → compile → test → package →

verify → install → deploy

build.xml

```
<?xml version = "1.0" ?>
<project name = "ProjectHW" default = "info">
    <target name = "info">
        <echo> Hellow world ! </echo>
```

```
</target>
</project>
```

```
<target name="deploy" depends="package">...</ >
<property name="sitename" value="___" />
<echo> This is ${sitename} </echo>
```

default — ant.version
ant.file
ant.project.name
ant.project.default-target

```
<property file="build.properties"/>
```

Inside build.properties

sitename = "___"
buildversion = 3.3.2

** file can be in any sub directory
* partial name match

$\varphi = (\neg x2 \vee x5) \wedge (x1 \vee \neg x3 \vee x4) \wedge (x4 \vee \neg x5)$

$\wedge (x1 \vee x2)$

set $x1$ to false

$\varphi = (\neg x_2 \vee x5) \wedge (0 \vee \neg x3 \vee x4) \wedge (x4 \vee \neg x5)$

$\wedge (x\,0 \vee x2)$

set $x2$ to false

$\varphi = (1 \vee x5) \wedge (0 \vee \neg x3 \vee x4) \wedge (x4 \vee \neg x5)$

$\wedge (0 \vee 0) = 0$

Backtrack and set $x2$ to true

$\varphi = (0 \vee x5) \wedge (0 \vee \neg x3 \vee x4) \wedge (x4 \vee \neg x5) \wedge (0 \vee 1)$

DPLL algorithm

$\varphi = (\neg x2 \vee x5) \wedge (x1 \vee \neg x3 \vee x4) \wedge (x4 \vee \neg x5)$

$\wedge (x1 \vee x2)$

1. set x2 to false

$\varphi = (1 \vee x5) \wedge (x1 \vee \neg x3 \vee x4) \wedge (x4 \vee \neg x5)$

$\wedge (x1 \vee 0)$

$= 1 \wedge (x1 \vee \neg x3 \vee x4) \wedge (x4 \vee \neg x5) \wedge (x1)$

set x1 to true

$\varphi = (1 \vee \neg x3 \vee x4) 1 (x4 \vee \neg x5) \wedge 1$

$= (x4 \vee \neg x5)$

set x4 to false

$\varphi = (0 \vee \neg x5)$

$= (\neg x5)$

set x5 to false

$\varphi = (\neg 0)$

$(x1, x2, x4, x5) = (1, 0, 0, 0)$

$x3 = X$

United International University
Department of Computer Science and Engineering

CSE 4495: Software Testing and Quality Assurance
Mid-term Examination : Summer 2022
Total Marks: 30 Time: 1 hour 45 minutes
Any examinee found adopting unfair means will be expelled from the trimester / program as per UIU disciplinary rules.

Answer all the questions. Numbers to the right of the questions denote their marks.

1. (a) What are the properties that must be established in a system to deem it as *'Dependable'*? Describe the relationship between these properties with necessary illustrations [3]

   (b) Consider the following statements- [3]

   i. A system can be correct yet unsafe.
   ii. A system can be safe but not robust.
   iii. A user needs to download a very large file over a slow modem, then he/she should be more concerned about the system *'availability'* rather than the *'Mean time between failures(MTBF)'*

   Now either support or oppose each of these statements and explain the logical reasoning behind your stand.

   (c) Imagine you are the lead developer of *FreeSpace.Inc* game studios. Your company wants to release a new mobile shooting game that will rival popular games like Free Fire, PUBGm etc. To achieve this your system needs to fulfil the following requirements-availability of at least 99.6%, a probability of failure on demand of less than 0.05, and a rate of fault occurrence of less than 4 failures per 36 hour work period. [4]
   After the testing is done you receive the following report from the testing team - "During 10 days of testing the system processed 18972 requests. Some of these requests ended in failure. Three types of failures were observed -

   i. 26 times the system showed an user wrong information about enemy position.
   ii. 27 times the game disconnected the user from a match.
   iii. 32 times the whole system crashed, and servers needed to be restarted. Each restart took 5 minutes on avg."

   Now depending on this report measure the availability, **POFOD** and **ROCOF** of your system. Also decide whether your software is ready for release.

2. (a) Show the V-model of development and elaborate on the different testing stages this model incorporates. [2]

   (b) The delivery route connection check is a high-level function exposed by the API of a International parcel shipping system. It is intended to check the validity of a single connection between two shipments in an itinerary. For example, a package may need to be delivered from Dhaka to New York, but there is a connection through London. Therefore, their itinerary is $Dhaka \rightarrow London(ShipmentA)$ and $London \rightarrow NewYork(ShipmentB)$. This service will ensure that the connection through London is a valid one. For example, if the arrival warehouse of Shipment A differs from the departure Warehouse of Shipment B, the connection is invalid. That is, if we pass in two Shipments, and Shipment A arrives in London, but Shipment B departs from Glasgow, it is not a valid connection. Likewise, if the departure time of Shipment B is too close to the arrival time of Shipment A, the connection is invalid. If Shipment A arrives in London at 8:00, and Shipment B departs at 8:05, there is not sufficient time to complete the unloading process of Shipment A and loading process of Shipment B. [8]

   validConnection(Shipment shipmentA, Shipment shipmentB) returns ValidityCode;

   A **Shipment** is a data structure consisting of:

   - A unique identifying shipping code (string, four characters followed by three numbers).
   - The originating warehouse code (four character string).
   - The scheduled departure time from the originating warehouse (in universal time).
   - The destination warehouse code (three character string).
   - The scheduled arrival time at the destination warehouse (in universal time).

   There is also a **Shipment database**, where each record contains:

   - Four-letter Warehouse code (four character string).
   - Warehouse country (three character string).
   - Warehouse city (three character string).

- Minimum processing times (integer, minimum number of minutes that must be allowed for Shipment connections).

ValidityCode is an integer with value:

- 0 for OK.
- 1 for invalid warehouse code.
- 2 for a connection that is too short.
- 3 for shipments that do not connect (arriving shipment is not stored in the same warehouse as departing shipment).
- 4 for any other errors (malformed input or any other unexpected errors).

Design system test cases using the category-partition method for the validConnection function.

i. Identify choices (aspects that you control and that can vary the outcome) for the two input shipments and the database.

ii. For each choice, identify a set of representative values.

iii. Apply **ERROR**, **SINGLE** and **IF** constraints. Explain the logic behind imposing these constraints

3. (a) Here are some of the configurations that can be controlled in a certain web browser: [6]

Allow Content to Load	Notify About Pop-Ups	Allow Cookies	Warn About Add-Ons
Allow	Yes	Allow	Yes
Restrict	No	Restrict	No
Block		Block	

Figure 1: Configurations for Question 3(a)

What will be the full number of test specifications in this case? Using Combinatorial interaction technique create a covering array covering all pairwise combinations of these browser configurations.

(b) How can one use Category-partition and Combinatorial interaction techniques together? Explain with necessary examples. [2]

(c) What are Unit test cases? Which test cases among Unit tests and system test are executed greater in number? Why? [2]