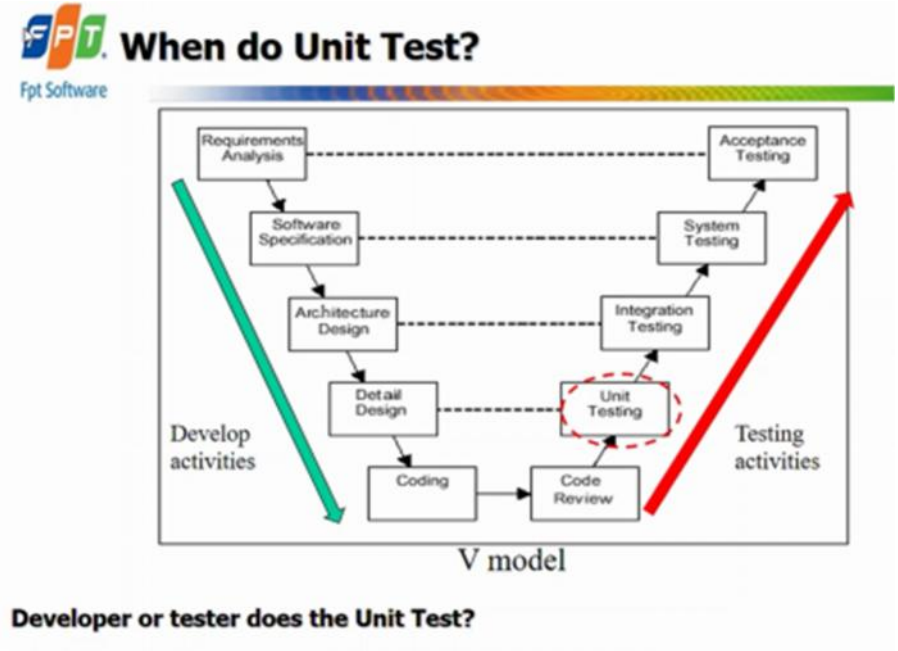
Điều khiển trợ lực lái cho vô lăng thì phát sinh 1 vấn đề là toyota thu hồi 50k xe. Trong việc sản xuất sắp xếp công đoạn không đúng dẫn đến tụ bị nứt dẫn đến thay đổi giá trị điện áp trong việc điều khiển trợ lực đánh lái dẫn đến gây ra lỗi.

Unit test là testing software code thực hiện kiểm tra các đoạn code cho các method, function, đơn vị nhỏ nhất của 1 phần mềm, có thể được sử dụng để.

- "Unit testing" refers to testing software code at the smallest testable unit (method or function) and based on detail design
- Exception testing
  - Range of feasible input
- Functional testing
  - Black Box Testing - conform to specification
  - White Box testing
- Regression testing
  - Conducted after a change
  - To find new fault
- Confirmation testing
  - Test to confirm that the bugs was fixed correctly



V model trong sản xuất phần mềm

Bên tay trái là các hoạt động của dev

Phải là testing. Với mỗi hoạt động của dev thì có 1 hoạt động của test.

Yêu cầu phân tích thu thập và nghiên cứu xem có chức năng gì. Tiếp theo là software specification là phân tích các yêu cầu có thể được thực hiện như yêu cầu về kỹ thuật hay các module cần có

Architecure design thì với các công cụ cần có đấy cần có mối quan hệ và sự phụ thuộc giữa chúng là như thế nào. Mqh và truyền tải link dữ liệu giữa chúng là như thế nào. Với mỗi module cần có detail design các functional feature chính. Với mỗi functional coding như thế nào trong giai đoạn coding.

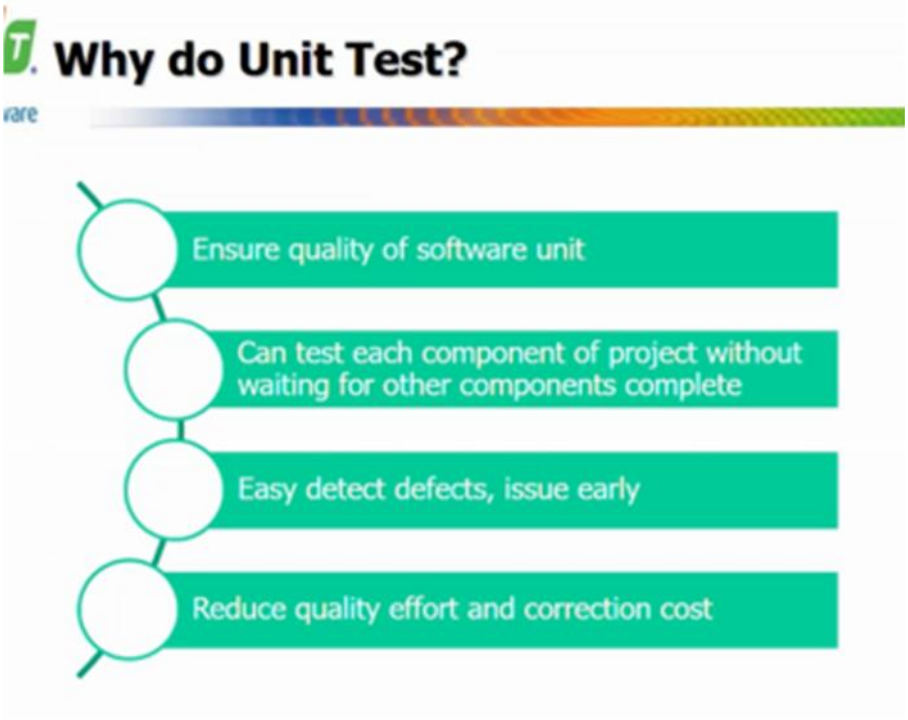
Sau việc code thì cần phải có code review về coding convention, giải thuật có đúng hay không, các vấn đề về complain có đúng ko, các vấn đề khác theo tài liệu

Đầu tiên phải tự review sau đó các sen với review

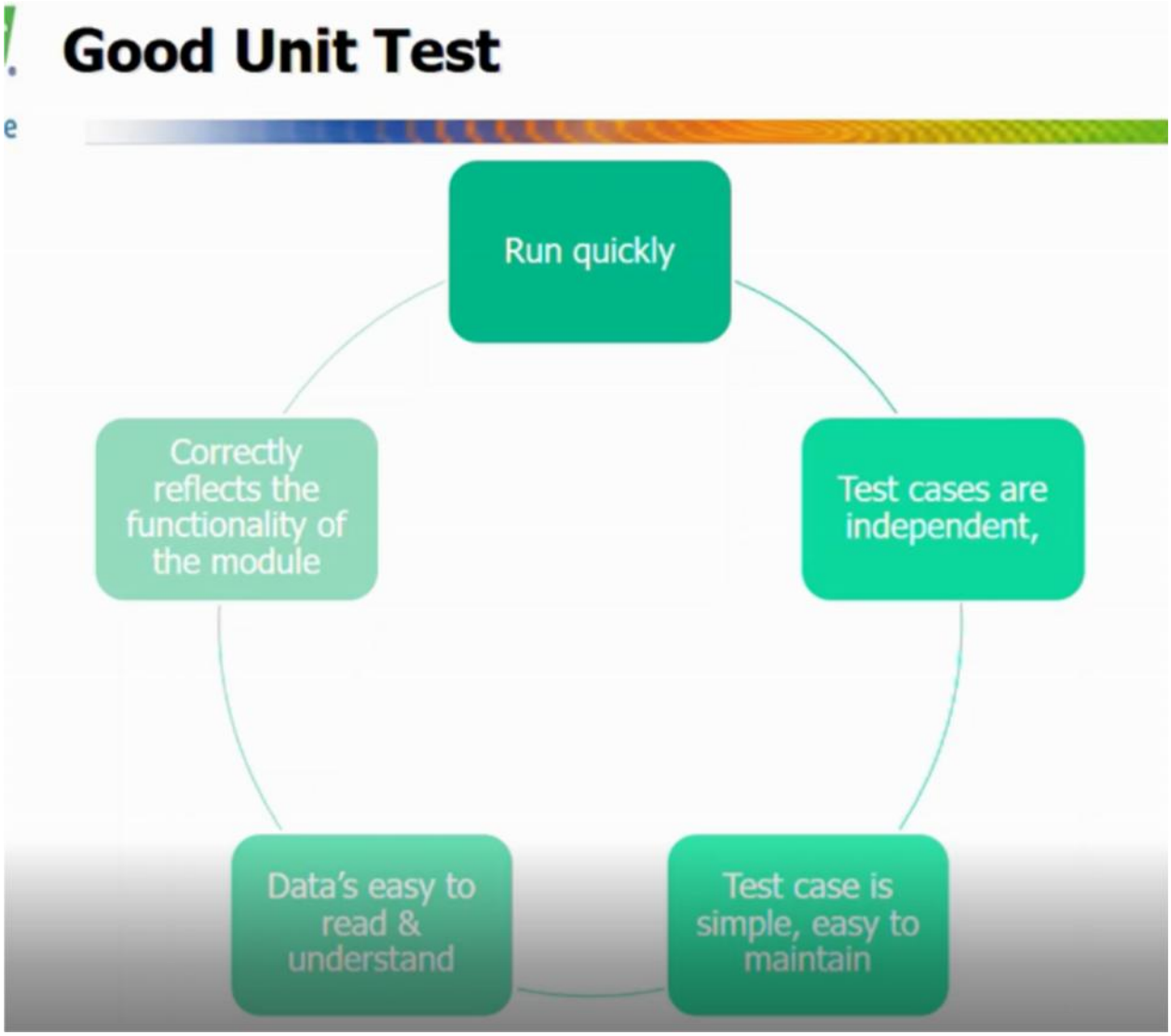
Unit testing là hoạt động của 1 dev là chủ yếu thực hiện các kỹ thuật testing

Intergration testing là test sự tương thích của các module với nhau, gọi là tính hợp lý phù hợp với nhau với các data dạng simulation

Accestance testing là người khác hàng test đưa input thực tế, bài toán thực tế cho mình để test xem có đáp ứng chức năng của cả hệ thống hay không



Để giảm thiểu các vấn đề sau này thì dùng unit test cho các đơn vị nhỏ nhất để đảm bảo quality của đơn vị nhỏ nhất.



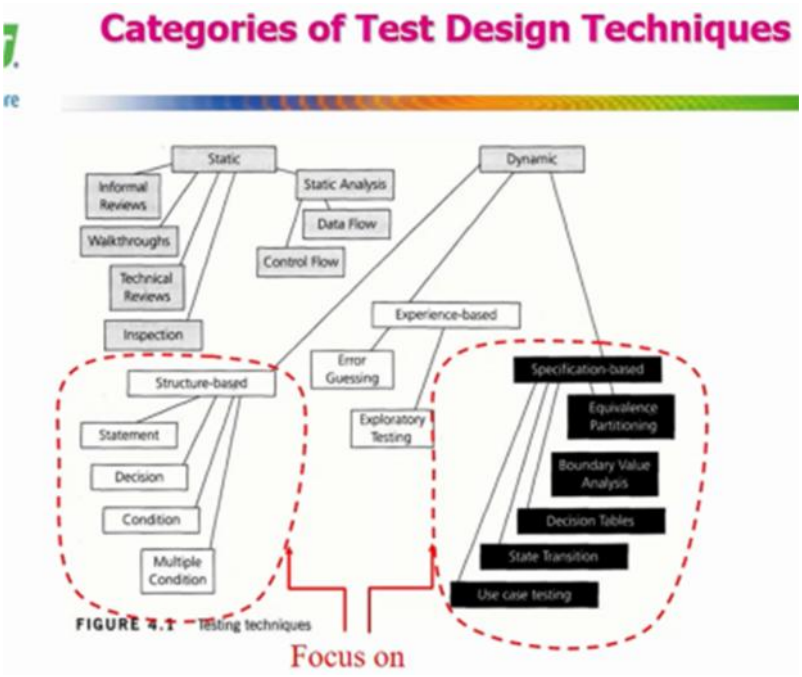
Các unit test cần đảm bảo về tốc độ vì cá đơn vị phần mềm nhỏ nhất dẫn đến là 1 trong các module này có rất nhiều đơn vị phần mềm từng function nhỏ rất nhiều. mỗi 1 unit test trong 1 function nhỏ đó mất thời gian tách ra thì cộng lại cả hệ thống thì tốn rất nhiều effort cho việc chạy có thể ảnh hưởng đến giá trị của sản phẩm

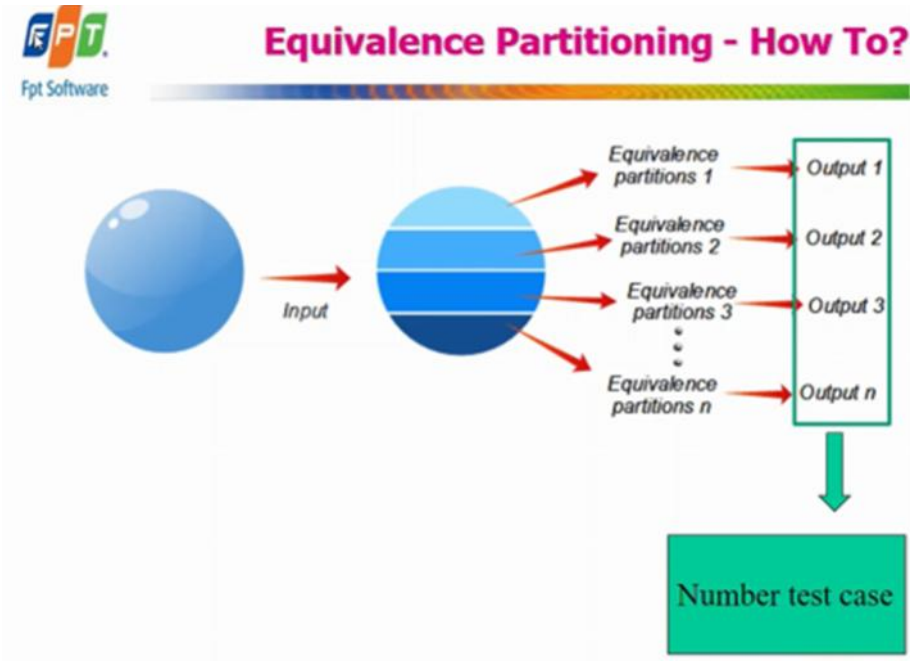
Việc test case chạy độc lập có thể đảm bảo được tính đúng đắn của nó, ko phụ thuộc vào việc chạy trước hay sau của cái này cái kia để đảm báo tính độc lập về kết quả.

Test case phải đơn giản và dễ hiểu để mà trong 1 hệ thống có nhiều người ra vào dự án. Việc test case được viết ra và sau đó có người kế thừa nó, maintance là chuyện rất bình thường. test case càng đơn giản và dễ maintance là tốt nhất.

Tiếp theo là sử dụng data để dễ đọc dễ hiểu, lựa chọn data dễ đọc dễ hiểu thì mọi người có thể dễ dàng hiểu việc tính toán dễ dàng và dễ làm sao đó thực hiện sao đó với con số đó thôi

Quan trọng nhất là tại sao làm việc phải đấu đúng hoạt động của module, để kiểm tra chức năng của 1 đơn vị phần mềm nào đó





Chia các khối cầu màu thành nhiều phân vùng khác nhau tương đương với giá trị input đầu vào.

Với mỗi giá trị tương đương input đầu vào, qua 1 unit test functional nào đó trong 1 output giống nhau thì sẽ có 1 phân vùng tương đồng. trên hình vẽ có n phân vùng tương đồng cho các màu giống nhau

Với mỗi phân vùng này chọn giá trị qua unit test thì sẽ có 1 output từ 1 đến N. với mỗi 1 output này có 1 test case từ 1 đến N

Để thực hiện phương pháp này có 2 bước chính

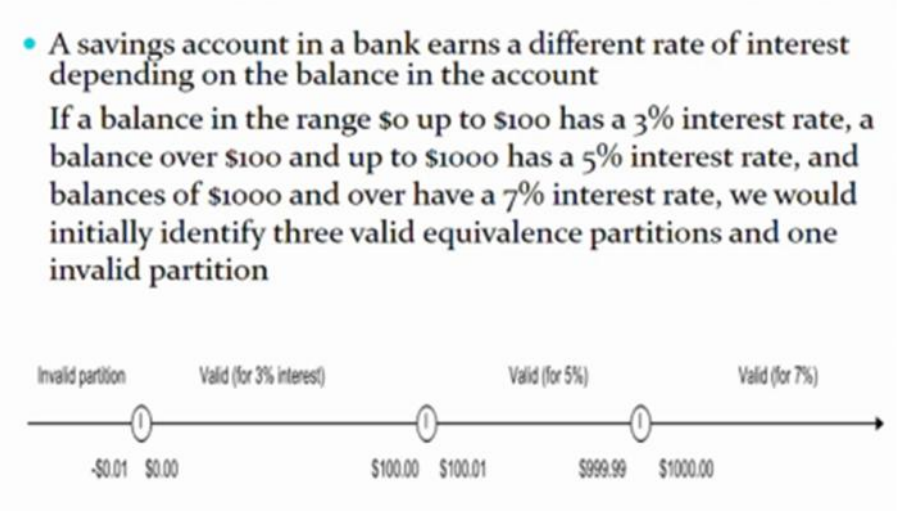
1 là phân biệt và cô đọng được các phân vùng tương đồng như thế nào. Mỗi phân vùng tương đồng đều cần chọn ra 1 giá trị. Với ở điểm keypoint thì ta cần phải thiết lập nó có các phân vùng tương đồng khác nhau dựa vào các driver input đầu vào và các output đầu ra.

Bước thứ 2 là cần có design tổng số lượng test case phụ thuộc vào số lượng phân vùng tương đồng

2 pp chính cho việc khoanh vùng tương đồng thì thứ nhất là sẽ gán giá trị đầu vào, còn đầu ra thì để tạo ra các vùng phân vùng tương đồng là valid class và invalid class.

Với các phân vùng tương đồng này mình sẽ có, tương ứng với 1 phân vùng tương đồng thì mình sẽ tạo 1 test case thì dựa vào đấy mình sẽ tạo 1 design 1 số lượng test case dựa vào phân vùng tương đồng này

Để hiểu rõ hơn thì có 1 ví dụ



Lãi suất tiền gửi ngân hàng

Từ 0 đến 100 đô là 3%

Trên 100 $ - 1000$ là 5%

Trên 1000$ là 7%

Ta có các vùng valid class là 3, 5 và 7 %

Invalid là trường hợp là nhỏ hơn 0 $ do các giá trị ta truyền vào là các giá trị nguyên đặc biệt là bên embedded.

Với mỗi phân vùng khác nhau sẽ chọn ra các giá trị

Ví dụ 3% chọn 10 $

5% chọn 500$

7% thì chọn 200$

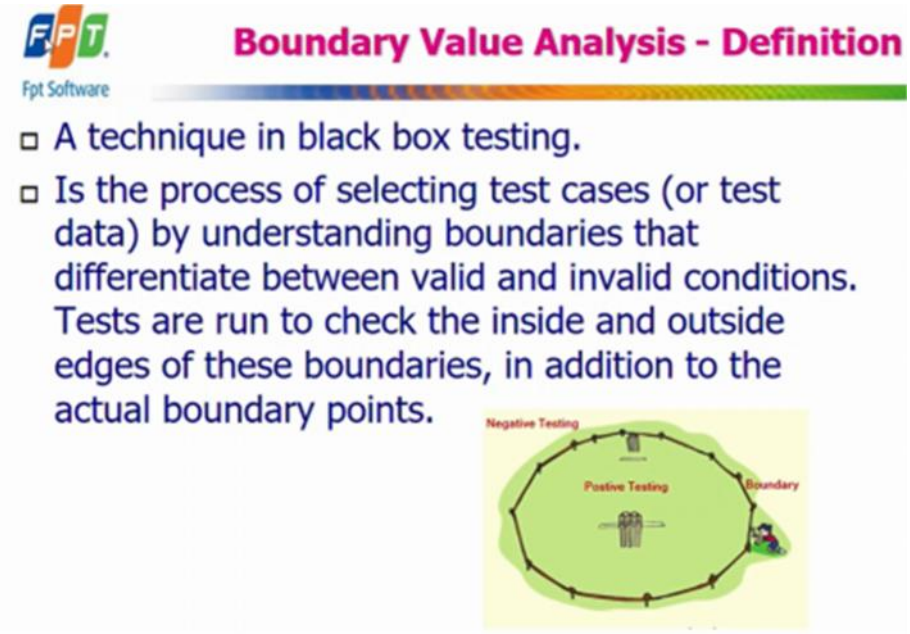
Dưới phân vùng invalid class chọn -5$

Mỗi phân vùng khác nhau cho ra giá trị lãi suất khác nhau

Với 50$ cho ra giá trị output là 3% lãi suất 3% của 50$

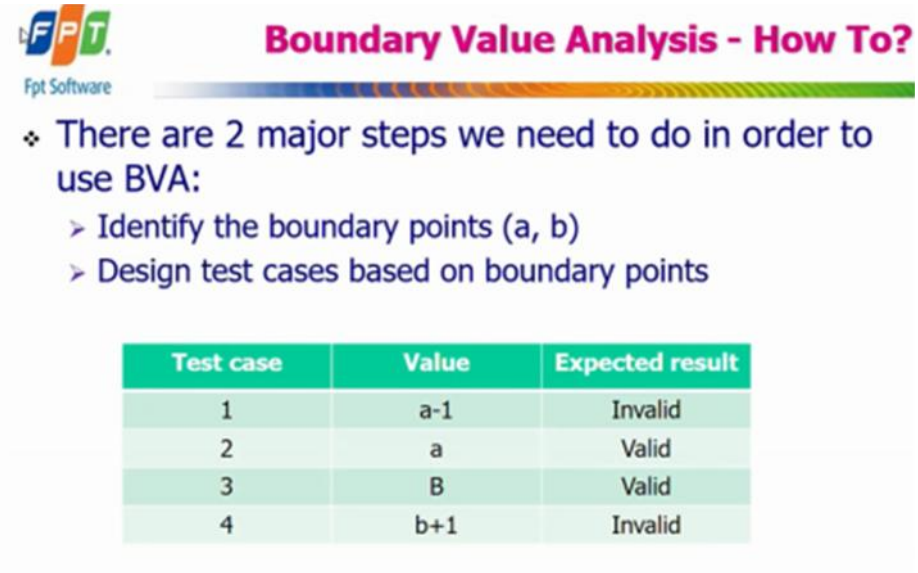
500$ thuộc phân vùng 5% thì output là 5% của 500$

7% của 2000$



Phân tích vùng biên là check cái giá trị biên và cận biên. Trong 1 dải thay đổi của data input thì có thể những giá trị ở giữa ko phát sinh ra lỗi. có những trường hợp cận biên hoặc biên thì mới phát sinh ra lỗi thì mình cần phải makesure dải data để check biên.

Để thực hiện pp này trải qua 2 bước

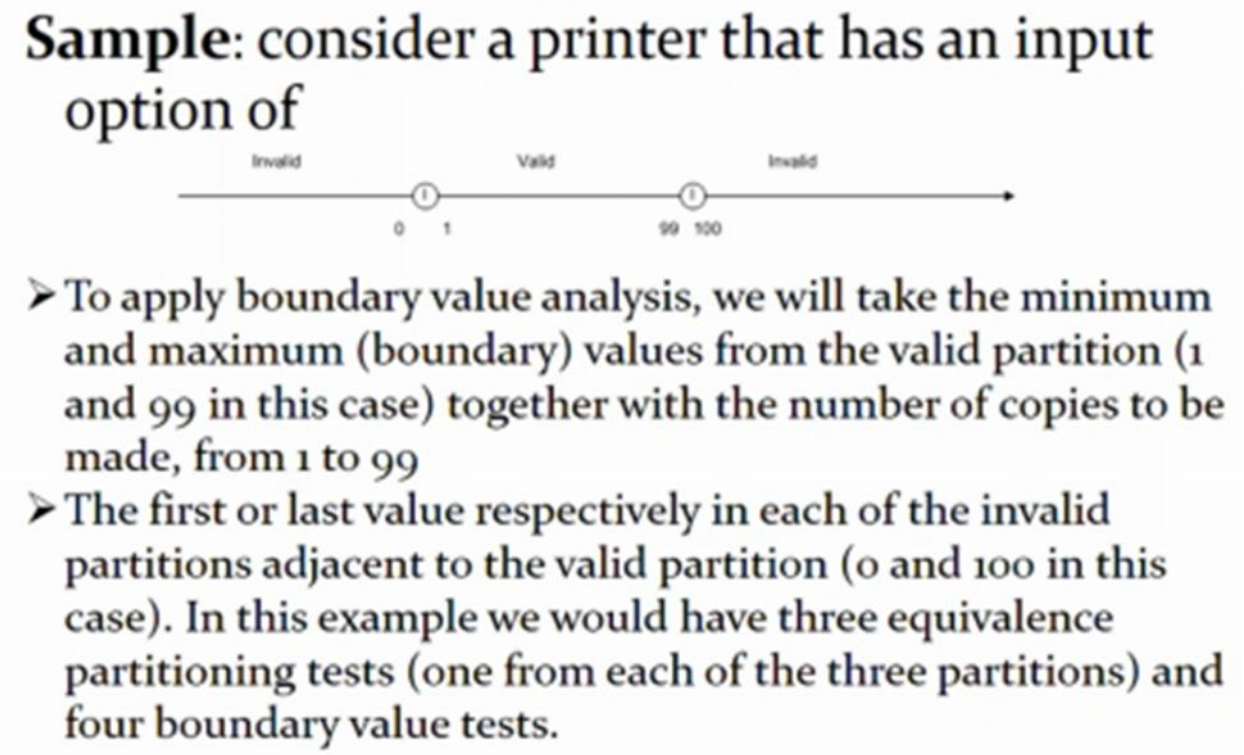


Bước 1 xác định vùng biên

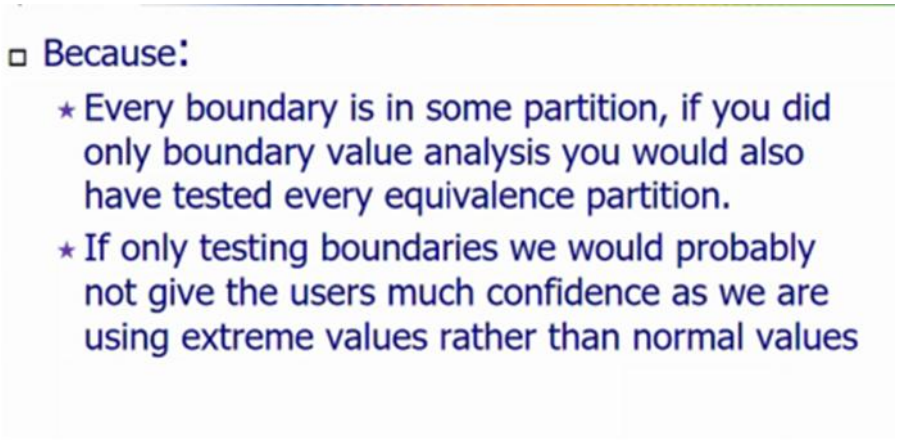
Design test case theo các giá trị vùng biên đó theo công thức bên dưới

Ngoài ra có thể mở rộng là giá trị ở các biên là a – 1 và b + 1 để đảm bảo nó make sure hơn

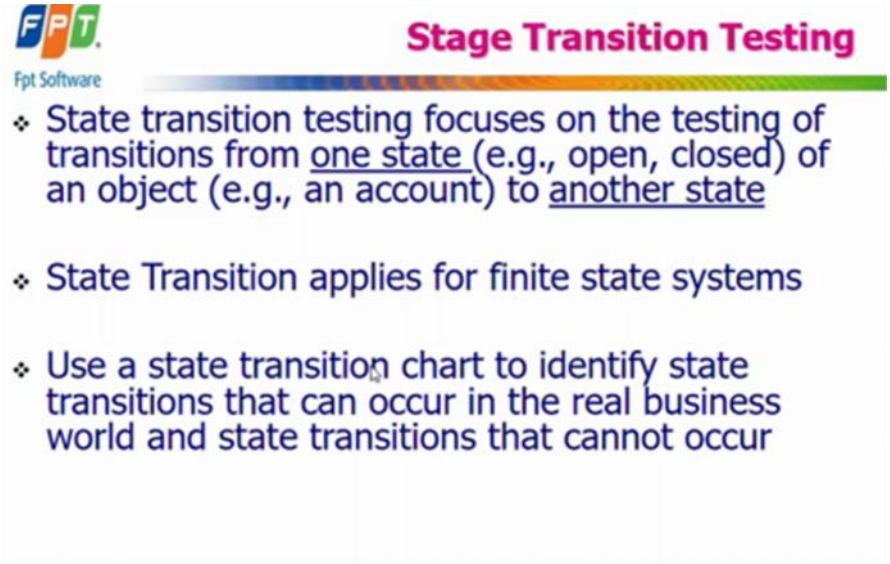
Ví dụ



1 đến 99 là valid còn lại là invalid



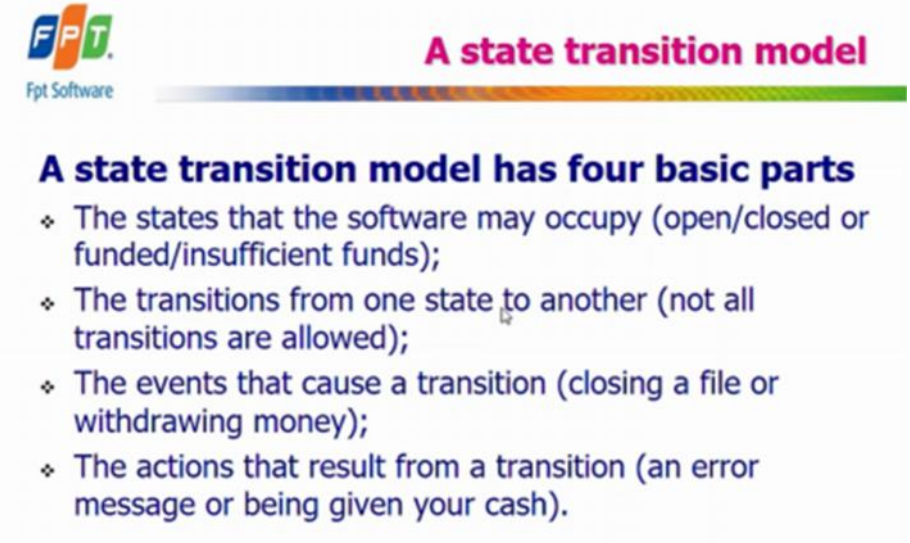
Nếu range rất dài thì chọn phương pháp phân vùng tương đồng cho 1 giá trị đại diện ở giữa chẳng hạn thì pp boundary value thì các giá trị giới hạn ở 2 đầu quay giới hạn ở 2 vùng biên là a b và a – 1 b + 1 thì có thể tiềm ẩn những khả năng bị lách và trường hợp bị lách có thể gây ra lỗi, bỏ sót lỗi. trong thực tế cần phải kết hợp 2 phương pháp này. Chọn các giá trị phân vùng tương đồng và giá trị vùng biên để tạo thành bộ test case hợp lý phù hợp



Pp này là stage transition testing là pp kiểm thử trong đó thay đổi điều kiện đầu vào và thay đổi trạng thái trong các ứng dụng

Pp này apply được cho các trạng thái hữu hạn xác định được

Để làm dc pp này cần có stage transition model

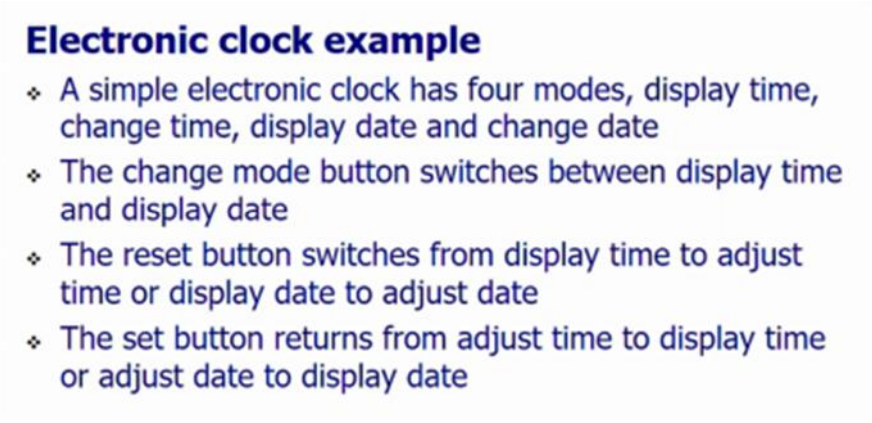


Gồm 4 thành phần

Các trạng thái của chương trình có thể có, ko đủ số dư tài khoản hay là hết hạn hay là các phiên chuyển đổi transition từ trạng thái này sang trạng thái khác

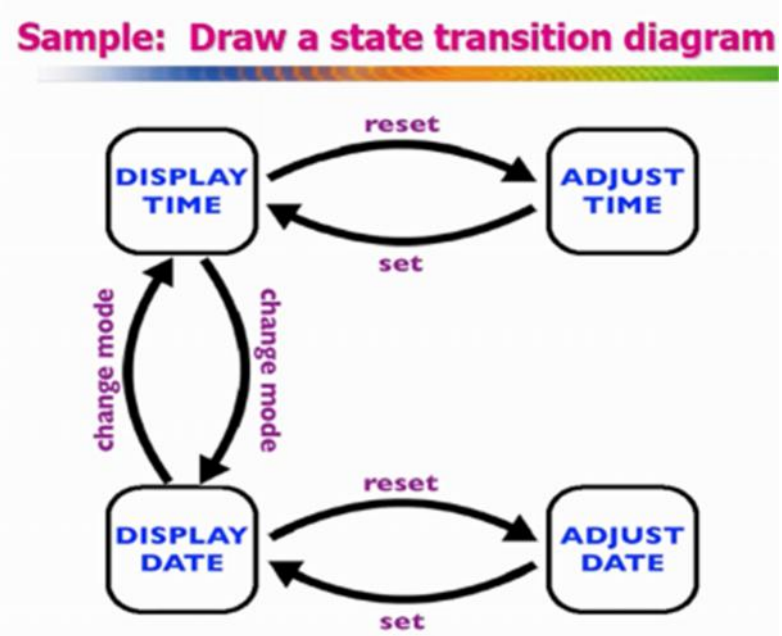
Thứ 3 là các sự kiện, event gây ra bởi phiên chuyển đổi, ví dụ như việc là nhập mã PIN hay là đóng FILE

Thứ 4 là các kết quả



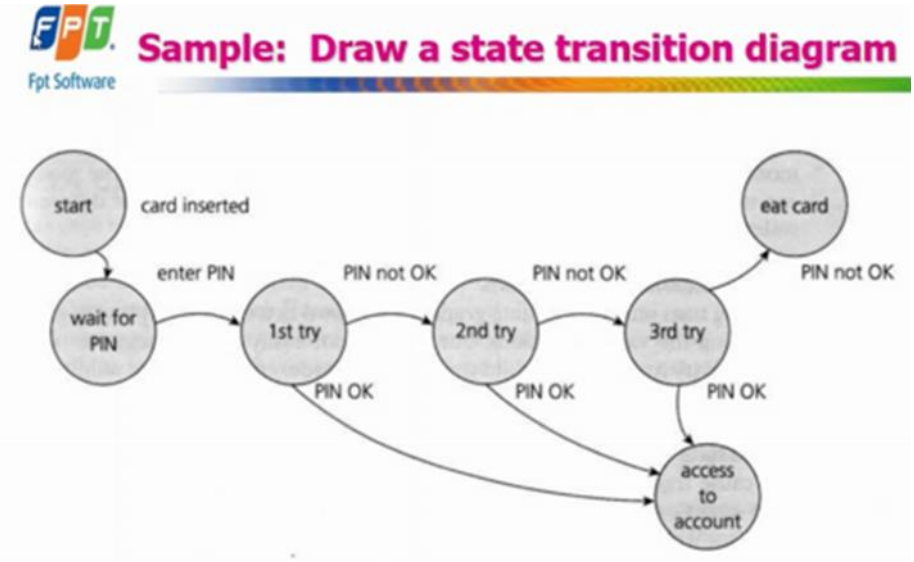
ví dụ về đồng hộ điện tử

Tương ứng với 4 trạng thái là 4 mode



Việc chuyển đổi các trạng thái với nhau có các nút set, reset, change mode. Dựa vào diagram có thể thiết kế ra các test case. Dựa vào đó có thể thấy được **các trường hợp không thể xảy ra**

Với những trường hợp này phát sinh trong test case thì có nghĩa là hệ thống đang bị lỗi



Stage transition của máy ATM: các phiên chuyển đổi, các event, ấn nút, các action

Với mỗi sơ đồ diagram thì mình có thể chuyển đổi các trạng thái dễ dàng. Kiểm tra các trạng thái trước khi chuyển đổi

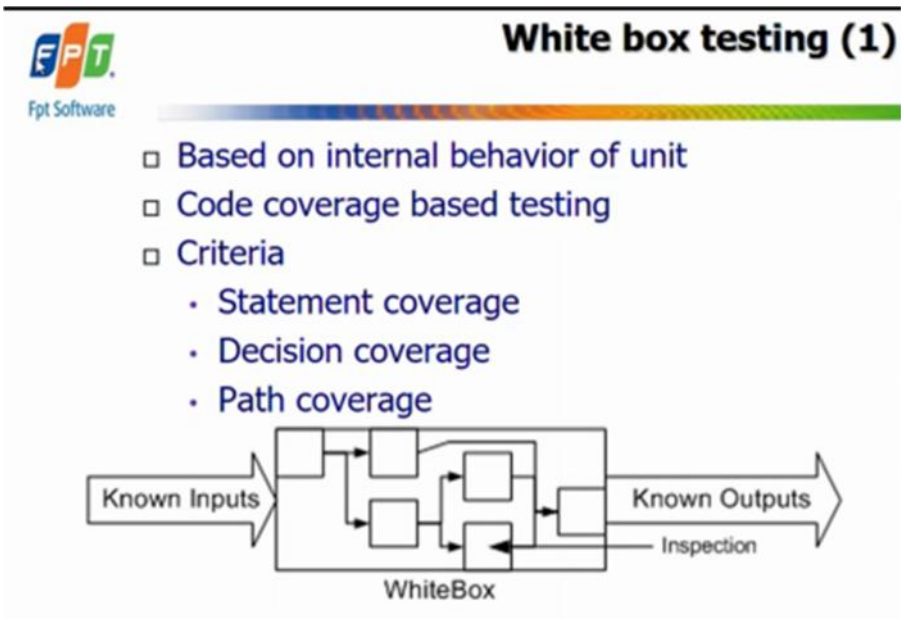
Ví dụ trạng thái 1 thì đăng nhập tới đâu chẳng hạn thì mình kiểm tra trước khi vào 1st thì nó làm cái gì, sau 1st thì nó làm cái gì. Sau 1st thì có các trạng thái là 2nd hoặc access to

account thì nó phụ thuộc vào cái transition hoặc các event tương ứng thì đó là dựa vào stage transition để thực hiện thiết lập số lượng các test case phụ thuộc vào diagram đó.

Phương pháp này dễ dàng cho mọi người nhanh chóng tiếp cận key point của bài toán và làm đúng

Ngoài việc kiểm tra mỗi trạng thái đúng thì còn kiểm tra trạng thái sai để kiểm tra được là việc kiểm tra trạng thái hoàn toàn chính xác ko đúng

Ví dụ như là kiểm tra trạng thái lần thứ nhất mà ko đúng thì ví dụ như là từ 1 lần thứ nhất mà chuyển sang ăn thẻ luôn thì là trường hợp test case như thế là ko thể thực hiện được nếu mà thực hiện được thì sẽ báo lỗi

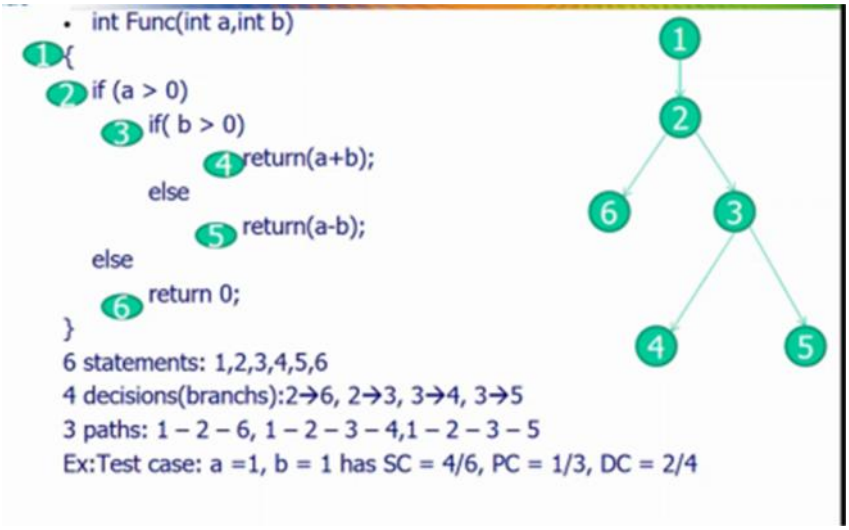


Tại sao gọi là white box

Giá trị sử dụng các block testing thì mn chỉ kiểm tra trạng thái giữa xác nhận trạng thái và functional thông qua các input đầu vào, các expectational output đầu ra nhưng với white box testing thì cần quan tâm tới nội dung bên trong functional nó như thế nào. Với functional có rất nhiều rẽ nhánh khác nhau dẫn đến những trường hợp blackbox testing ko test hết được nhỡ nó vào luồng (stream) xử lý mà mình ko cover được hết thì lỗi issue có thể tiềm ẩn ở luồng nhánh đó

Thì lúc đó mới cần white box testing để tìm ra lỗi bên trong của hàm mục đích là để cover toàn bộ các functional trong nhánh, đã được kiểm tra để tránh lỗi phát sinh tiềm ẩn trong đấy

Đi sâu vào stagement coverage là các dòng, decision coverage là các câu lệnh rẽ nhánh, path coverage là đường đi của các functional đã biết



Với 2 biến đầu vào ab thì tương ứng với các giá trị đầu vào ab thì có các luồng xử lý khác nhau cho các giá trị đầu ra khác nhau thì mình sẽ có 6 stagement

Với a = 1 b = 1 thì mình cover được 4 stagement và sẽ cover được 2 nhánh trên 4 nhánh

Cần cover toàn bộ thì chọn các giá trị khác nhau ví dụ như a = 0, b = 0; a = 0, b = 1

Thì stage mới bao phủ toàn bộ các SC, PC, DC thì khi đó function mới đảm bảo quality tốt hơn

DRIVER là 1 loại modul dùng để kiểm thử, code xong thì dùng 1 driver bên ngoài gọi cái functional đó để test thử, trong trường hợp này

```
Driver: Gọi module để được kiểm thử. (code xong >> dùng 1 driver bên ngoài gọi nó để chạy)
/*======================== FUNCTION NEEDS TO TEST ============================== */
status Clock_Getfreq(const clock_source_t Clk, uint32_t * Fre)
{
    status State;

    if (Fre == NULL)
    {
        State = STATUS_ERRORED;
    }
    else
    {
        switch (Clk)
        {
            case TIMER_CLK:
                /* Do something for calcaulation Frequency value */
                /* Example fre = 1M Hz */
                *Fre = 1000000UL;
                State = STATUS_SUCCESSED;
                break;

            case UART1_CLK:
                /* Do something for calcalation */
                *Fre = 8000000UL;
                State = STATUS_SUCCESSED;
                break;

            case UART2_CLK:
                /* Do something for calcalation */
                *Fre = 4000000UL;
                State = STATUS_SUCCESSED;
                break;
            ...
            ...
            defualt:
                /* Nothing change */
                *Fre = 0UL;
                State = STATUS_UNSUPPORTED;
                break;
        }
    }

    return State;
}
/* ------------------------------------------------------------------- */
```

```
/* ------------------------------------------------------------------ */

/* Implement the driver */
clock_source_t array_Module[4U] = {UART1_CLK, UART2_CLK, TIMER_CLK, CAN_CLK};
status_t array_StateValide[4U] = {STATUS_SUCCESSED, STATUS_SUCCESSED, STATUS_SUCCESSED, STATUS_UNSUPPORTED};
uint32_t array_FreqValidate[4U] = {8000000UL, 4000000UL, 1000000UL, 0UL};
uint32_t array_Freq[4U]={0UL};
/* DRIVER CHECK */
stautus_t Driver_Fucntion()
{
    uint8 i = 0U;
    stautus_t State = STATUS_SUCCESSED;

    for (;i < 4U; i++)
    {
        State = Clock_GetFreq(array_Module[i], &array_Freq[i]);
        if ((State != array_StateValide[i]) || (array_Freq[i] != array_FreqValidate[i]))
        {
            State = STATUS_ERRORED;
            break;
        }
    }


    return State;
}
```

Để test clock frequency

Đây là ví dụ để viết 1 driver gọi 1 functional đang cần kiểm thử thực tế thì việc kiểm thử này được gọi bởi các hàm khác nhau của các functional khác trong việc tính toán các functional frequency