



S2. Estrategias y métodos de caja negra


Sitio: [Agencia de Habilidades para el Futuro](#)
Curso: Metodología de Pruebas de Sistemas 2° D
Libro: S2. Estrategias y métodos de caja negra


Imprimido por: Eduardo Moreno
Día: martes, 19 de agosto de 2025, 19:40

Descripción


Inicio


Apertura


Desarrollo


Práctica



Cierre

Tabla de contenidos

1. Introducción a las estrategias de testing

2. Estrategia de caja negra

2.1. Casos de prueba con caja negra

3. Estrategia de caja blanca

3.1. Falla 1: Caminos lógicos grandes

3.2. Falla 2: pruebo todos los caminos, entonces la prueba está completa

4. Historias de Usuarios

5. Modelo de plantilla de Historias de Usuario

6. Concepto de pruebas de usuario

7. Estrategia y métodos de caja negra

8. Método 1: Clases de equivalencia

8.1. Paso 1: Identificación

8.2. Pasos 2 y 3: Definición de casos de prueba

9. Método 2: Análisis de valores límites

9.1. Consejos para analizar

9.2. Ejemplos del método 2

10. Método 3: Grafos de causa - efecto

11. Método 4: Adivinación de defectos

12. La estrategia: combinar métodos



¡Avancemos! A continuación encontrarás la definición y características de cada estrategia.





Estrategia de caja negra

Definición Esta estrategia de *testing* ve al programa como una **caja negra**, donde el comportamiento interno del software y su estructura no es importante. En vez de eso, **se concentra en encontrar las circunstancias en las cuales un software no se comporta según su especificación.**

Con esta estrategia, la información de entrada de los casos de prueba se deriva únicamente de las especificaciones de requerimientos. Es decir, sin tener la ventaja del conocimiento de la estructura interna del programa.

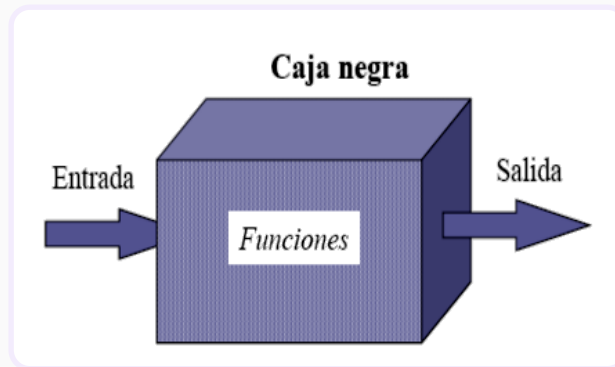


Figura 1: representación gráfica de caja negra



Caja negra: prueba exhaustivas de entradas

Si se quiere encontrar absolutamente todos los errores del programa usando esta estrategia, entonces la respuesta es probar con una cantidad exhaustiva de entradas. Esto quiere decir no sólo usar todas las entradas válidas, sino todas las **entradas posibles**.

Caso de prueba 1: Un valor numérico

En el caso de un campo que debería tomar un valor numérico, deberíamos probarlo con todos los posibles valores desde el 0 hasta el número más grande que permita el lenguaje, pero también con todos los valores posibles inválidos, todas las letras, combinaciones de letras y combinaciones de letras y números permitidas por el lenguaje de programación. Notaremos entonces, que para un solo campo esta es una cantidad casi infinita de casos de prueba. Aún más si pensamos que ningún programa en realidad tiene un solo campo, sino que tiene una combinación de entradas posibles.

Caso de prueba 2: Compilador de C++

Si el caso de prueba 1, suena difícil, imaginá la prueba exhaustiva de programas más grandes y complejos.

Considerando un compilador de C++, no sólo tendríamos que crear casos de prueba representando todos los programas válidos de C++ (un número casi infinito), sino que además tendríamos que considerar todos los programas inválidos de C++ (un número definitivamente infinito) para asegurarnos que el compilador detecte que son inválidos. Esto último es para probar que el compilador no haga algo que no debe hacer, compilar un programa sintácticamente incorrecto; tal como vimos en los principios de la semana 1.

El problema se vuelve aún más grande y complicado para el software transaccional que utiliza bases de datos.

Tomemos de ejemplo de un software de reservas de vuelos de una aerolínea, estas transacciones no sólo dependen de las entradas de un caso en particular, sino también de lo que ocurrió anteriormente. Si ya hay asientos reservados en un vuelo, no podemos permitir que se vuelvan a reservar. Es decir que no sólo debemos probar todas las transacciones válidas e inválidas, sino que tenemos que pensar en todas las posibles secuencias de transacciones.

Podríamos seguir listando situaciones cada vez más complejas, pero el punto de la discusión es que la prueba exhaustiva es imposible. Podemos definir entonces 2 implicaciones:

- 1 No podemos probar un software para garantizar que esté libre de errores (como habíamos visto en la semana 1).
- 2 Una consideración fundamental a la hora de probar, es la económica.

Por lo tanto, no hay lugar para la prueba exhaustiva.

El objetivo debería ser aumentar el rendimiento del *testing* maximizando el número de errores encontrado por un número finito de casos de prueba. Una parte de esto implica mirar el código y asumir algunas cosas respecto al software (en el ejemplo del campo que toma un valor numérico del 1 al 10, asumir que el número 1, 2 y 3 tendrán los mismos resultados.)



Estrategia de caja blanca

Definición

Esta estrategia de *testing* ve el programa como una **caja blanca**, es decir que examina la estructura interna de un programa. Esta estrategia deriva casos de prueba a partir de la lógica del software, desafortunadamente ignorando la especificación de requerimientos.

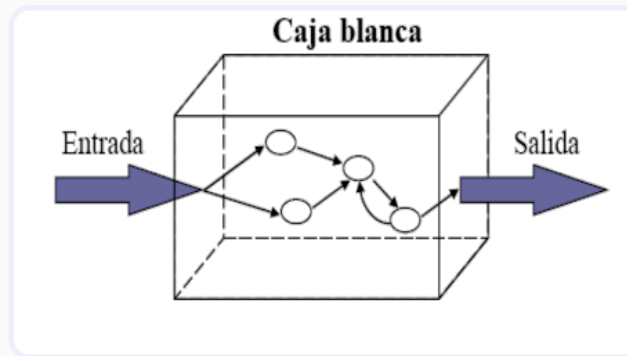


Figura 1: representación gráfica de caja blanca

El objetivo es, entonces, establecer una estrategia análoga a la prueba de entradas que mencionamos en la caja negra.

Queremos causar que cada declaración en el programa se ejecute al menos una vez, pero no es difícil mostrar que esto será extremadamente complejo e inadecuado.

Se suele considerar que si ejecutamos todos los posibles flujos de control del software, entonces ese software se ha probado por completo.



Falla 1

Una de las fallas en la forma de pensar la estrategia de caja blanca es que el número de caminos lógicos de un programa puede ser astronómicamente grande.

Para ilustrar esto, consideremos la figura 1 que se muestra debajo:

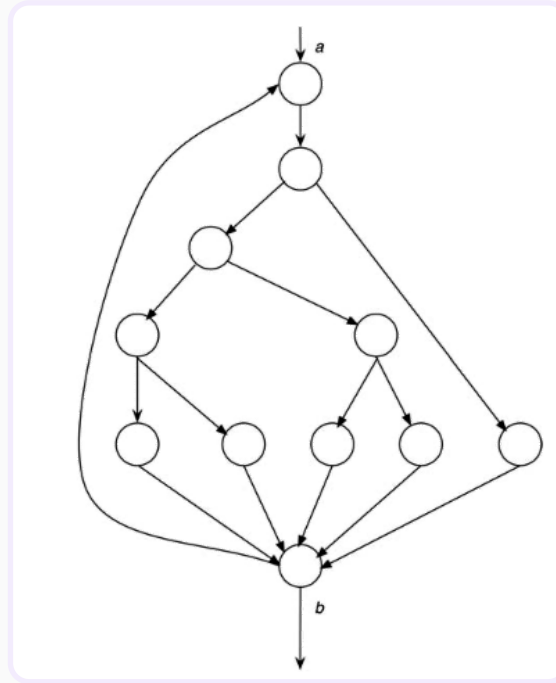


Figura: Diagrama de control de caminos de un programa pequeño

Esta sería una representación de un programa trivial, **es un diagrama de control de flujo del programa.**

- 1 Cada nodo o círculo representa una declaración que se ejecuta de forma secuencial, posiblemente modificándose aún más.
- 2 Cada arco representa la transferencia de control entre las declaraciones. El diagrama representa entonces entre 10 a 20 declaraciones de tipo **FOR** o **WHILE** que puede iterar hasta 20 veces.
- 3 Dentro de cada declaración de tipo *loop* puede haber declaraciones de tipo **IF**.
- 4 Determinar el número único de caminos, entonces, es equivalente a determinar la cantidad de formas únicas de moverse del punto **a** al punto **b** (asumiendo que las decisiones del software son independientes entre sí). Este número puede ser de más de 100 trillones de caminos.

En resumen

Con números tan altos nos es difícil visualizar estos números, así que lo pongamos en una forma más concreta. Si pudiéramos escribir, ejecutar y verificar cada caso de prueba en 5 minutos, nos tomaría aproximadamente 100.000 millones de años probar cada camino. Si pudiéramos escribir, ejecutar y verificar cada caso de prueba en 1 segundo, nos tomaría 3.2 millones de años.

Por supuesto, las decisiones en un software real no son independientes entre sí. Es decir que el número posible de caminos ejecutables sería bastante menor.

Por otro lado, los programas reales suelen ser bastante más complejos que el programa que se muestra en la figura 1. Es decir, la prueba exhaustiva de caminos, al igual que la prueba exhaustiva de entradas, parece cada vez más impráctica, si no imposible.

Analicemos, a continuación, la siguiente falla.



Falla 2

La segunda falla con la forma de pensar “**si pruebo todos los caminos internos del software entonces la prueba esta completa**” es que cada camino de ejecución del software puede probarse, pero el software aun así puede contener defectos.

Existen 3 motivos para esto:

1

Una prueba exhaustiva de caminos no garantiza que el programa cumpla con su especificación.

Consideremos como ejemplo una rutina que ordene legajos de estudiantes por año de ingreso de los más nuevos a los más viejos. Si por error programamos una rutina que los ordene de forma inversa, la prueba exhaustiva de caminos nos aportara poco valor al error a encontrar. Podemos ordenar de forma perfecta a todos los estudiantes, pero ese software tiene un error fundamental: No es el software que se necesita. No cumple con sus especificaciones. La prueba de caminos no encontrará jamás este gran problema.

2

Un programa puede estar erróneo porque falta programar algunos caminos. La prueba exhaustiva de caminos no descubriría este problema.

Si consideramos el software académico que discutíamos en el punto 1, sí deberíamos poder buscar un alumno por legajo y eso se hace bien, podría faltar el camino del error. ¿Qué ocurre cuando no se encuentra un alumno? Este camino podría faltar porque el desarrollador no lo consideró, y una prueba exhaustiva de caminos nunca detectaría la falta de este camino en particular.

3

Una prueba de caminos exhaustiva puede no descubrir errores dados por los datos. Hay muchos ejemplos de este tipo de problemas.

Imaginamos un programa que compara que la diferencia entre 2 números sea menor a un valor determinado, un típico ejemplo de monitoreo. Tendríamos un código en Java como el siguiente:

```
if ((a - b) < c) {  
    print ("El valor está dentro de los parámetros");  
}
```

En este caso, para una cuestión de monitoreo, el código contiene un error. El valor c debería compararse con el valor absoluto de la diferencia en a y b. Ya que, si b es mayor que a, pero aun dentro del rango, el número sea negativo, pero para el negocio estará correcto. Esto dependerá enteramente de los valores de entrada y no será encontrado necesariamente recorriendo todos los caminos del programa

Como conclusión, podemos decir que a pesar de que la prueba exhaustiva de entradas es superior a la prueba exhaustiva de caminos, ninguna es suficiente para asegurar el buen funcionamiento del programa. Para obtener un buen software necesitaremos una forma de combinar ambas estrategias: la de caja negra y la blanca.



¿Qué son las Historias de Usuarios?

Antes de seguir adelante, queremos hacer foco en este concepto clave en la metodología de pruebas de sistemas.

Definición

Una Historia de Usuario es una forma de especificación de requerimientos muy utilizada hoy en día. Es una forma corta y con pocos detalles de implementación, está más centrada en el usuario.

Se escribe de esta forma:

“ Como <ROL> quiero <Funcionalidad> para <Valor de negocio> ”

Esto nos permite saber qué usuario utilizará esa funcionalidad, qué es lo que quiere hacer el usuario y además nos da un valor de negocio para ayudarnos a priorizar las funcionalidades.

En el capítulo 5, podrás conocer en detalle la plantilla de Historias de usuario. Pero, ahora continuemos con otras características:

Las Historias de Usuario además tienen:

- los llamados **criterios de aceptación**, que son criterios que nuestro usuario final utilizará para determinar si aceptará o no una funcionalidad desarrollada.
- Se pueden agregar **notas**, lo que permitirá agregar aun más información a la Historia de Usuario que no necesariamente se expresa en los criterios de aceptación.

Generar estadística de Consumos de agua por período

Como Responsable de Medición quiero generar reportes estadísticos de consumo para analizar el uso del servicio de agua por parte de los clientes.

Nota 1: Los reportes estadísticos pueden ejecutarse por período, por zona y localidad.

Nota 2: En la salida deben mostrarse m³ consumidos por período, zona y localidad.

Nota 3: El período no puede ser nulo.

Nota 4: Los campos Localidad y Zona pueden dejarse Vacíos, se considerarán en el reporte en ese caso TODAS las Zonas o TODAS las Localidades

Nota 5: El reporte puede descargarse en formato PDF, xls oxlsx

Podemos ver que tenemos que desarrollar una funcionalidad para generar estadísticas de consumo del agua de nuestros clientes por un periodo determinado de tiempo. Para poder desarrollar esta funcionalidad necesitaremos hablar con nuestro cliente y pedir más información, pero **de eso se tratan las Historias de Usuario**.



Modelo de plantilla de Historias de Usuario

La frase "Como <ROL> quiero <Funcionalidad> para <Valor de negocio>" es una plantilla utilizada en el contexto de la metodología ágil, particularmente en la metodología *Scrum*.

Esta plantilla se utiliza para definir y estructurar las historias de usuario, que son una forma de expresar requisitos de software de una manera que sea fácil de entender y priorizar.

La plantilla se completa de la siguiente manera:

- **<ROL>**: Aquí se especifica el rol del usuario o persona que necesita una funcionalidad o característica en el software. Puede ser un usuario real, como un cliente o un miembro del equipo, y a menudo se describe de manera específica. Por ejemplo, "Como cliente" o "Como administrador del sistema".
- **<Funcionalidad>**: En esta parte, se describe de manera concisa la funcionalidad o característica que el usuario o rol desea. Esto debe ser claro y específico, y debe ser una descripción breve de lo que se necesita.
- **<Valor de negocio>**: Aquí se indica por qué el usuario o el rol necesita esa funcionalidad. Se relaciona con el beneficio que se obtendrá al implementar esta característica. Puede ser un valor para el negocio, como aumentar la eficiencia, mejorar la experiencia del usuario o cumplir con un requisito legal.

Un ejemplo de historia de usuario utilizando esta plantilla podría ser:

"Como cliente, quiero poder realizar compras en línea (funcionalidad) para facilitar la adquisición de productos y mejorar la conveniencia (valor de negocio)."

Esta plantilla facilita la comunicación y la priorización de requisitos en proyectos ágiles. Ayuda a las personas involucradas en el desarrollo de software a comprender:

- ✓ quién necesita
- ✓ qué necesita,
- ✓ por qué lo necesita y,
- ✓ cómo puede proporcionar valor al negocio ó a los usuarios.



Pruebas de usuario

Las historias de usuario cuentan con pruebas de usuario que no deben confundirse con casos de prueba, ya que estos, entre otras características son documentos que describen de manera detallada cómo se probará una funcionalidad o característica específica del software. El propósito principal de los **casos de prueba** es verificar que una parte específica del software funcione según lo previsto.

Y..., entonces ¿Qué son las pruebas de usuario?

Definición Las **pruebas de usuario** son pruebas rápidas sin especificar del todo que nuestro usuario final o cliente utilizará para determinar si los criterios de aceptación se cumplen, y por tanto se va a aceptar la Historia de Usuario.

Esa lista no es exhaustiva y puede ser bastante corta.

Amplíemos el concepto de pruebas de usuario

- Son una fase más amplia del proceso de aseguramiento de calidad que se centra en la validación de todo el sistema desde la perspectiva del usuario final.
- Las pruebas de usuario **evalúan si el software satisface** las necesidades y expectativas de los usuarios.
- Pueden **incluir casos de prueba**, pero también pueden abarcar escenarios más amplios, como pruebas de aceptación del usuario (UAT), pruebas de usabilidad y pruebas de rendimiento.
- Se centran en la **experiencia del usuario final** y la satisfacción del cliente.
- Se centran en la **validación desde la perspectiva del usuario final**.
- Se evalúa **si el software es fácil de usar**, cumple con las expectativas del usuario y satisface las necesidades del negocio.
- Como las pruebas de aceptación del usuario (UAT), generalmente se realizan en una etapa posterior del proceso de desarrollo, cuando el software está cerca de estar listo para su implementación.

Empresa potabilizadora de agua

Presentamos, a modo de ejemplo, el caso de una Empresa potabilizadora de agua que requiere desarrollar un producto de software que permita gestionar la prestación del servicio de agua potable en las diferentes localidades. Podrán analizar el caso completo en el documento descargable al final de esta página, pero observemos cómo se vería la tabla con las pruebas de usuario, a partir de la Historia de usuario presentada en páginas anteriores:

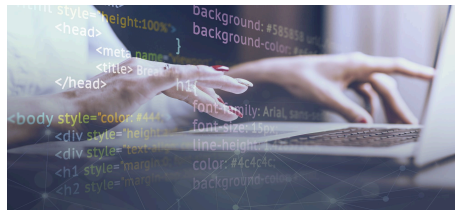
Pruebas de Usuario

- ☐ Probar visualizar reporte para un período existente, para una única zona y localidad (pasa)
- ☐ Probar visualizar reporte para un período existente, para todas las zonas y localidades (pasa)
- ☐ Probar visualizar reporte para un período a futuro (falla)
- ☐ Probar visualizar reporte sin loguearse como Responsable de Medición (falla)
- ☐ Probar visualizar reporte para un período sin consumos (pasa)
- ☐ Probar visualizar reporte para una localidad inexistente (falla)



Te invitamos a repasar la explicación a partir del ejemplo completo de la "Empresa potabilizadora de agua", cuya historia de usuario y pruebas de usuario encontrarás en este archivo: [clic aquí para acceder](#).

Retomaremos este ejemplo más adelante, para analizar los métodos de caja negra.





Partición en clases de equivalencias

Hemos dicho anteriormente que un buen caso de prueba es uno que tiene una alta probabilidad de encontrar un defecto. También hemos establecido que una prueba exhaustiva de entradas es imposible. Es decir, que estamos limitados a un pequeño subconjunto de posibles entradas.

Lo que nosotros queremos es seleccionar el subconjunto correcto de entradas para lograr la mayor probabilidad de encontrar la mayor cantidad de errores.

Una forma de encontrar este subconjunto es notar que un caso de prueba bien confeccionado debe tener 2 propiedades:

- 1 **Reducir** en más de uno el número de casos de prueba que deben ser confeccionados para cumplir el objetivo de una cantidad de pruebas “razonable”.
- 2 **Cubrir** un gran conjunto de otros casos de prueba posibles. Es decir, nos dice algo sobre la presencia o ausencia de errores más allá de las entradas específicas que se usan en el mismo.

Si bien estas propiedades pueden parecer similares, describen 2 consideraciones muy diferentes:

- La primera consideración implica que cada caso de prueba debe **tener en cuenta tantas entradas diferentes como sea posible** para minimizar la cantidad total de casos de prueba.
- La segunda implica que deberíamos **considerar partir el dominio del programa en un número finito de clases de equivalencia**, de forma que podamos asumir (aunque nunca estaremos absolutamente seguros) que una prueba de un valor representativo de una clase es equivalente a probar cualquier otro valor de esa misma clase.

En conclusión

- si la prueba de una clase de equivalencia detecta un error, todos los otros casos de prueba en esa clase de equivalencia deberían tener el mismo error.
- si un caso de prueba no detecta un error para una clase, podríamos esperar que ningún caso de prueba de esa misma clase contenga errores.

Las 2 consideraciones mencionadas anteriormente, se unen para formar un método de prueba de caja negra llamado **partición de equivalencias**:

- La segunda consideración se utiliza para confeccionar **un conjunto de condiciones** interesantes para probar.
- La primera consideración se utiliza para confeccionar **un conjunto mínimo de casos** de prueba cubriendo estas condiciones.

Un ejemplo de una clase de equivalencia

En el programa que valida un número del 1 al 10 de la introducción, sería decir “cualquier número entero mayor o igual a 1 y menor o igual a 10”. Al identificar esta **clase de equivalencia**, lo que queremos decir es que si no se encuentra ningún error con un caso de prueba que está probando un elemento cualquiera de esa condición, por ejemplo el número 2, entonces es poco probable que encontremos un error probando otro elemento del conjunto, como por ejemplo, el número 5.

En otras palabras: Nuestro tiempo de *testing* valdría más invirtiéndolo probando otras clases de equivalencia.

El diseño de casos de prueba utilizando partición de equivalencias tiene dos pasos:

- 1 Identificación de las clases de equivalencia.
- 2 Definición de los casos de prueba.

Es importante notar que, si bien este método es ampliamente superior a probar casos de prueba al azar, aún tiene sus problemas. Uno de los mayores es que no prueba necesariamente los escenarios con mayor rendimiento de errores.



Paso 1: Identificación de las clases de equivalencia

Las clases de equivalencia se identifican tomando las condiciones de entrada (normalmente una sentencia o comentario en la especificación) y partirlo en 2 o más grupos. Se puede utilizar la tabla que se muestra a continuación para lograr esto.

Es importante notar que hay 2 tipos de clases de equivalencias: clases válidas y clases inválidas.

- Las **clases válidas** representan entradas que el programa debería considerar válidas (en el ejemplo del validador, la clase que identificamos “cualquier número entero mayor o igual a 1 y menor o igual a 10”).
- Las **clases inválidas** representan aquellas entradas que causan cualquier otro posible estado de la condición, es decir, valores incorrectos o erróneos (en el ejemplo del validador, podríamos tener la clase “números enteros menores a 1”).

En este método hacemos foco en el principio 5 que vimos en la semana 1, que establece que debemos hacer foco también en las condiciones inválidas o inesperadas, las cuales iremos registrando en una tabla como la siguiente:

Condición Externa	Clases de Equivalencia válidas	Clases de Equivalencia inválidas
Clases de equivalencia de entrada		
Clases de equivalencia de salida		

Dada una salida con condición externa, identificar las clases de equivalencia es mayormente un proceso heurístico. Se pueden seguir los siguientes consejos:

- 1 Si una condición de entrada identifica un **rango de valores**, se identifica una clase válida y 2 clases inválidas. En el ejemplo del campo validador tenemos la válida mencionada antes y las 2 inválidas “números menores a uno” y “números mayores a 10”.
- 2 Si una condición de entrada identifica una **cantidad de valores a ingresar**, entonces identificamos una clase válida y al menos 2 inválidas.

[destacar] Analicemos con un ejemplo. Si evaluamos un juego multijugador que se juega con 2 a 4 jugadores donde cada jugador tiene su nombre de usuario y un contador de partidas jugadas, una clase válida es “cantidad de jugadores igual o mayor a 2 e igual o menor a 4”. Mientras que las clases inválidas son “menos de 2 jugadores” y “más de 4 jugadores”.

- 3 Si una condición de entrada especifica **una serie de valores** y hay alguna razón para creer que el programa los maneja de forma diferente, entonces identificamos una clase válida para cada opción y una clase inválida para cualquier otro valor.

Comprendamos el tercer consejo con el ejemplo.

[destacar] Si consideramos los envíos de Mercado Libre como un ejemplo, podemos identificar “Envío a domicilio” y “Retiro en sucursal” como una clase válida y “Cualquier otro valor” para una clase inválida. Esto se debe a que cuando seleccionamos “Envío a domicilio” debemos ingresar a qué domicilio enviar y Mercado Libre nos muestra una serie de opciones, pero el “Retiro en sucursal” nos muestra una serie de opciones diferentes. Como la selección sólo puede tener uno de esos 2 valores, entonces cualquier otra cosa es una clase inválida.

- 4 Si una condición de entrada **especifica es algo obligatorio**, entonces debe haber una clase inválida para cualquier entrada que cumpla ese criterio y una clase inválida para cualquier entrada que no lo cumpla.

[destacar] En el ejemplo de una contraseña de usuario, podemos tener una clase válida que sea “cadena de caracteres de entre 8 a 16 caracteres con al menos una letra mayúscula y una letra minúscula” y una clase inválida que sea “cadena de caracteres que no cumple con el formato”.

Si tenemos un motivo para creer que el software no manejará de la misma manera distintos elementos de una clase de equivalencia, entonces debemos partir esa clase de equivalencia en otras clases más pequeñas que abarquen menos información.



Paso 2 y 3: Definición de los casos de prueba

El **segundo paso** para el método de 1, es usar las clases de equivalencia para identificar y definir los casos de prueba. El proceso es el siguiente:

- 1 Asignar un número único a cada clase de equivalencia.
- 2 Escribir un caso de prueba cubriendo la mayor cantidad de clases de equivalencia válidas aún no cubiertas posibles. Hacer esto hasta cubrir todas las clases de equivalencia válidas.
- 3 Escribir un caso de prueba cubriendo una clase de equivalencia inválida aún no cubierta, sólo una. Hacer esto hasta cubrir todas las clases de equivalencia inválidas.

El motivo para el **tercer paso**, es que a veces una entrada errónea puede enmascarar otra entrada inválida, dependiendo de cómo se haya escrito el código.

Cuando escribimos los casos de prueba intentamos que sean “agnósticos” a la tecnología. Es decir, que no debería importarnos con qué tecnología han sido implementados ni de qué forma (si una selección usa un *combobox* o un autocomplete). Esto lo hacemos para ver qué es lo que esperamos del software a partir de ciertas entradas, no para ver cómo realmente está implementado.

Para la descripción de los **casos de prueba** debemos especificar:

- **Nombre único:** Nos ayudara a identificar un caso de prueba fácilmente. El nombre debe ser representativo del mismo, por ejemplo: compra de artículos exitosa utilizando tarjeta de crédito sin observaciones.
- **Precondiciones:** Es el contexto que necesitamos para ejecutar una prueba. Generalmente lo podemos derivar de la especificación de requerimientos. Algunas precondiciones comunes son fecha en la que se ejecuta la prueba, artículos pre-cargados con toda la información necesaria para la ejecución (nombre, precio, modelo, etc.), usuario existente y logueado con permisos específicos, seleccionables a usar cargados, etc.
- **Pasos:** Paso por paso a ejecutar durante la prueba. Para programas mas “simples” (como veremos en el ejemplo que les compartimos más abajo) podemos tener un solo paso, pero la idea de un caso de prueba es que cualquiera pueda replicarlo en cualquier momento. Si no tenemos un paso a paso con los datos exactos a ingresar, entonces el caso de prueba no puede replicarse. Este paso a paso incluye el orden en el que el *tester* realiza cada cosa a hacer, así como también la información exacta a usar.
- **Resultados esperados:** ¿Qué es lo que espero del sistema? Se deriva de clases de equivalencia de salida. Puede ser el envío de un e-mail, un mensaje en particular, el resultado de un cálculo.



Te invitamos a leer el documento [S2-Ejemplos de metodología de equivalencias](#), donde encontrarás ejemplos que te ayudarán a comprender este método.



Análisis de valores límites

La evidencia demuestra que los casos de prueba que se centran en examinar situaciones límite obtienen mejores resultados en comparación con aquellos que no lo hacen.

Definición | Las **condiciones límite** son aquellas situaciones que están directamente en, justo por encima o justo por debajo de los límites de las clases de equivalencia de entrada o salida.

El análisis de valores límite difiere de la técnica de partición de equivalencias de 2 maneras:

1 En vez de seleccionar cualquier elemento de una clase de equivalencia y tomarlo como representativo, el análisis de valores límite toma los elementos que están directamente al borde de la clase de equivalencia que se está probando.

2 Esta técnica no sólo se concentra en las condiciones de entrada, sino que también se derivan casos de prueba a partir de las condiciones de salida.

Por ejemplo, si tenemos un software que al finalizar el registro envía un e-mail, entonces "E-mail enviado al usuario registrado dando la bienvenida e informando su nombre de usuario y contraseña" será una clase de equivalencia válida de salida. Mientras que, si el usuario ingresa una contraseña no válida, la clase de equivalencia inválida de salida sería "Mensaje de error que dice 'La contraseña debe ser de entre 8 a 16 caracteres y contener al menos una mayúscula y una minúscula'".

Es muy difícil presentar una "**receta**" para el análisis de valores límite ya que requiere un grado importante de creatividad y de especialización del problema a probar. Por lo tanto, recordando el principio 10 del proceso de *testing*, es importante la creatividad y estado mental, más que de otra cosa.

Sin embargo, se pueden dar algunos **consejos** que ayudarán en la técnica que veremos a continuación.



Consejos para el análisis de valores límites

1

Si una condición de entrada especifica un rango de valores, se debería escribir casos de prueba para los límites del rango. Así como también los valores inválidos que están justo en los límites del rango.

Si tenemos un campo con un dominio válido entre -1.00 y 1.00, entonces deberíamos tener casos de prueba que verifiquen las siguientes situaciones: -1, 1, -1.01 y 1.01.

2

Si una condición de entrada especifica un número de valores, se deberían escribir casos de prueba para la cantidad mínima y máxima de valores, así como también una cantidad justo debajo de la mínima y justo por encima de la máxima.

Por ejemplo, si un software puede importar archivos de Excel de entre 1 a 10000 registros, entonces deberíamos probarlo con un Excel sin registros, con 1 registro, con 10000 registros y con 10001 registros.

3

El consejo 1 también aplica para cada valor de salida posible.

Por ejemplo, si tenemos un software que liquida sueldos y debemos hacer deducciones impositivas, donde el mínimo es \$0.0 y el máximo \$5000, entonces debemos probar condiciones que resulten en una deducción negativa, una deducción de \$0.0, una deducción de \$5000 y una deducción mayor a \$5000.

Es muy importante probar no sólo los límites de las entradas, sino también los límites de las salidas, ya que puede ocurrir que los límites de entrada no siempre tengan como resultado los límites de la salida.

4

El consejo 2 también aplica para cada condición de salida. Si nuestro software tiene un rango de resultados, entonces debemos probar las condiciones para que se den todos los resultados posibles.

Por ejemplo, si tenemos un software que busca artículos médicos y puede resultar en la muestra de entre 2 a 8 artículos, deberíamos probar ocasionar resultados donde muestre 1 artículo, 2 artículos, 8 artículos y 9 artículos.

5

Si las entradas o salidas de un programa es un conjunto específico y pre-definido, entonces se debe prestar especial atención a los primeros y últimos elementos de las listas. Es importante notar que esto no siempre es posible,

si tenemos una selección de genero entre "Femenino", "Masculino" y "No binario" entonces no existe un límite que esté primero o último para aplicar la técnica.

6

Si alguna condición de entrada tiene condiciones particulares, como longitud del campo, peso de un archivo o algo similar, entonces realizar casos de prueba para esas condiciones válidas e inválidas.

Por ejemplo, si nuestro software soporta imágenes de hasta 2 Gb, entonces probar con una imagen de 2 Gb y con una imagen de 2.01 Gb.

7

Finalmente, utilizar la creatividad y la inteligencia para encontrar qué otros límites pueden existir.



Ejemplos del método 2

El ejemplo del campo que valida números del 1 al 10 puede mostrar perfectamente un análisis de valores límites. Para los valores de entrada podemos probar el número 0, el número 1, el número 10 y el número 11. Así tendríamos todos los casos límites cubiertos.

Esta es una técnica extremadamente simple, que es muy fácil de utilizar incorrectamente. Las condiciones límites son muchas veces extremadamente sutiles, por lo que la identificación de las mismas requiere mucha atención y tiempo de pensamiento. Veremos algunas de esas sutilezas en los ejemplos a continuación.

Para la especificación de casos de prueba utilizamos el mismo formato que vimos en la técnica de particiones de equivalencia:

- **Nombre único:** Nos ayudará a identificar un caso de prueba fácilmente. El nombre debe ser representativo del mismo. Ejemplo: “compra de artículos exitosa utilizando tarjeta de crédito sin observaciones”
- **Precondiciones:** Es el contexto que necesitamos para ejecutar una prueba. Generalmente lo podemos derivar de la especificación de requerimientos. Algunas precondiciones comunes son fecha en la que se ejecuta la prueba, artículos pre-cargados con toda la información necesaria para la ejecución (nombre, precio, modelo, etc.), usuario existente y logueado con permisos específicos, seleccionables a usar cargados, etc.
- **Pasos:** Paso por paso a ejecutar durante la prueba. Para programas más “simples” (como veremos en el ejemplo) podemos tener un solo paso, pero la idea de un caso de prueba es que cualquiera pueda replicarlo en cualquier momento. Si no tenemos un paso a paso con los datos exactos a ingresar, entonces el caso de prueba no puede replicarse. Este paso a paso incluye el orden en el que el *tester* realiza cada cosa a hacer, así como también la información exacta a usar.
- **Resultados esperados:** ¿Qué es lo que espero del sistema? Se deriva de clases de equivalencia de salida. Puede ser el envío de un e-mail, un mensaje en particular, el resultado de un cálculo.



Te invitamos a leer el documento [S2-Ejemplos de metodología de análisis de valores límite](#) donde podrás analizar cómo se aplica este método en dos casos concretos: el software MTEST y la Empresa potabilizadora de agua, que ya hemos presentado en el libro anterior.



Método 3: Grafos de causa - efecto

Este método se describirá de manera breve, ya que no se espera que los técnicos superiores en desarrollo de software tengan todas las herramientas necesarias para ejecutarla. Sin mencionar que es poco usada en el mercado laboral, del año 2022, por la complejidad que conlleva utilizarla. Se introducirá para que los estudiantes conozcan de la misma e investiguen por su cuenta por si desean hacerlo.

Los grafos permiten de una forma sistémica seleccionar qué condiciones utilizaremos para ejecutar los casos de prueba. También ayuda a evidenciar cuándo la especificación está incompleta o escrita de una forma ambigua.

El grafo utiliza lógica de circuitos digitales, por lo que no veremos esta técnica ya que se requiere un conocimiento de lógica y álgebra de Boole, así como también conocimientos de la notación electrónica de circuitos.

Esta técnica se basa en **descomponer** el problema a probar en partes extremadamente pequeñas para trabajar con los grafos.

Luego se identifican las causas y efectos de los comportamientos en la especificación.

- Una **causa** es una entrada al sistema, que puede corresponderse con una condición externa o incluso una clase de equivalencia.
- Un **efecto** es una condición de salida o una transformación del sistema.

Las causas y efectos se leen detenidamente y se transforman a lógica de Boole utilizando los grafos, luego se utiliza tablas de verdad para determinar los casos de prueba necesarios para probar el software.



Adivinación de defectos

A veces ocurre que una persona parece naturalmente dotada para el **testing** de software. Sin utilizar ninguna metodología en particular, estas personas suelen tener la habilidad de encontrar errores.

Una explicación para esto es que estas personas están utilizando, de manera inconsciente, una técnica de **diseño de casos de prueba llamada adivinación de defectos**.

Dado un programa en particular, las personas pueden conjeturar ciertos tipos de errores probables y luego diseñar casos de prueba en base a encontrar esos errores. Generalmente estas conjeturas se dan por intuición o, más a menudo, experiencia.

Es difícil dar un procedimiento para la adivinación de defectos, ya que es mayormente intuitiva y un proceso ad-hoc. La idea básica es enumerar todos los posibles errores o situaciones dada a errores, y luego escribir casos de prueba en base a esa lista.

Por ejemplo, la presencia del valor 0 en las entradas de un programa es una situación que se suele prestar para errores. Por lo tanto, escribiríamos casos de prueba donde ciertas entradas en particular reciben el valor 0 y ciertas salidas tienen como resultado el valor 0.

También podemos decir que en los casos donde una entrada o salida presenta una lista (por ejemplo, cantidad de valores en una lista en la cual queremos buscar), una gran cantidad de errores se encuentran cuando la lista esta vacía o contiene sólo un elemento. Otra idea es identificar casos de prueba asociados a cosas que los desarrolladores podrían asumir al leer las especificaciones; especialmente relacionados a factores o condiciones omitidas, ya sea por error o porque el escritor de la especificación las consideraba obvias.

Como no es posible dar un procedimiento para este método, podemos plantear algunas preguntas relacionadas al ejemplo del **MTEST** que no han surgido en la metodología de análisis de valores límites:

- ❓ ¿El programa acepta una respuesta en blanco por parte de un estudiante?
- ❓ ¿Qué pasa si tenemos un registro de tipo "2" (las preguntas) metido entre todos los registros de tipo "3" (los estudiantes)?
- ❓ ¿Qué pasa si 2 estudiantes tienen el mismo legajo?
- ❓ La mediana se calcula diferente según si tenemos una cantidad par o impar de valores, así que se podría probar el programa con ambos casos para ver que el calculo se haga bien en ambos.
- ❓ ¿Qué pasa si el campo "número de preguntas" está vacío?
- ❓ ¿Qué pasa si en el campo "tipo" (donde debería ser 2 o 3) aparece otro número?



La estrategia: combinar métodos

Los métodos de diseño de casos de prueba discutidos esta semana pueden ser combinados en una sola estrategia. La razón para combinarlos debería ser obvia a esta altura...

Cada método contribuye con casos de prueba particulares extremadamente útiles; pero ninguno contribuye con un conjunto de casos de prueba lo suficientemente extensivo.

Una estrategia razonable que combina métodos sería la siguiente:

- 1 Para cualquier evento, utilizar el método de análisis de valores límites. Recordemos que es un análisis de los valores de entrada y salida, específicamente los límites.
- 2 Identificar las clases de equivalencia válidas e inválidas para cualquier entrada y salida, suplementando los casos de prueba que ya se han encontrado.
- 3 Usar la técnica de adivinación de errores para agregar casos de prueba que se deriven del sentido común y no se hayan encontrado aún.
- 4 Examinar la lógica del programa utilizando técnicas de caja blanca, las cuales veremos la semana siguiente.

El uso de esta estrategia no garantizará un programa libre de errores, pero representa un compromiso razonable en cuanto a cantidad y gravedad de defectos. También representa muchísimo trabajo duro, pero nadie dijo que hacer software sería fácil.