RECETAS PARA EL DISEÑO DE CASOS DE TESTEO

Tanja E.J. Vos, Jorge Sánchez Sánchez

Software Quality, Usability and Certification
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia,
Camino de Vera s/n - 46022 Valencia, Spain
{tanja,jordisan}@iti.upv.es

Resumen. Es difícil clasificar las técnicas de diseño de casos de testeo, de forma que sea sencillo decidir cómo y cuándo seleccionarlas y utilizarlas a la hora de testear un software determinado. A pesar de que esas dificultades han sido señaladas por varios autores, en este artículo describimos algunos resultados de un ambicioso proyecto en ejecución en el ITI (Instituto Tecnológico de Informática). El objetivo de este proyecto es definir una librería o colección de recetas de diseño de casos de test («test-case design recipes») tales que, una vez seleccionada una receta adecuada a la situación concreta, indiquen al testeador cómo realizar los pasos necesarios para generar los casos de test.

Palabras clave: Testeo de software; técnicas para el diseño de casos de testeo; recetas.

1. ALGUNOS CONCEPTOS BÁSICOS

Empezamos definiendo algunos términos comunes en el testeo de software, para clarificar ideas y que pueda servir de introducción a los lectores menos expertos en el tema.

1.1. ¿Qué es el testeo de software?

El testeo de software tiene como objetivo encontrar, sistemáticamente, diferentes tipos de errores en el software utilizando un tiempo y esfuerzo mínimos. Los objetivos y ventajas de hacer testeo son muchos:

- Dar una indicación del nivel de la calidad del software (es decir, si funciona o parece funcionar según lo especificado).
- Mejorar la calidad del software, permitiendo que se reparen los errores que van apareciendo durante el testeo.

- Hacer felices a los usuarios, que encuentran menos errores.
- Ahorrar dinero en el mantenimiento del software (y evitar tratar con usuarios descontentos).
- Ser más competitivo (¡una consecuencia indiscutible de las ventajas mencionadas anteriormente!).

Sin embargo, como Dijkstra ya indicó hace muchos años «testing cannot show the absence of errors -- it can only show that software errors are present» [Dij70]. Por lo tanto tenemos que ser cuidadosos con la definiciones que utilicemos para el testeo de software. Definiciones como "el testeo de software prueba que mi software funciona" o "prueba que cumplimos con los requisitos" ¡no son correctas! Primero porque, como afirmó Dijkstra, *tales demostraciones son imposibles* en la práctica. Segundo, estas definiciones hacen que los testeadores inconscientemente ejecuten *tests con baja probabilidad de encontrar errores*, ya que su tendencia será a mostrar que el software funciona ¡mientras que su verdadero objetivo tiene que ser encontrar errores!

Consecuentemente, hoy en día se ha aceptado que la mejor *definición del testeo de software* es la que ha dado Meyer [Mey04]:

- *Testeo* es el proceso de ejecutar un programa *con la intención de encontrar errores.*
- Un *buen test* (*o caso de test*) es un test que tiene una alta probabilidad de encontrar un error todavía no revelado.
- Un *test exitoso* es un test que ha revelado un error.

¿Por qué "testear" y "testeo" (en vez de "probar" y "prueba")?

Puede resultar extraño el uso en este artículo de los anglicismos "testear" y "testeo-/test" en lugar de "probar" y "prueba". Hemos usado los primeros ya que el término "probar" (y su derivado "prueba") es ambiguo, pudiendo usarse, en español, en diferentes sentidos:

- Hacer examen y experimento de las cualidades de alguien o algo (como en "probar un coche"; en inglés, «to test»).
- Demostrar, hacer patente la certeza de un hecho o la verdad de algo (como en la expresión "probar un teorema"; en inglés, «to prove»). En el ámbito del testeo significaría "demostrar que el software hace lo que tiene que hacer".

Como ya se ha explicado, mediante el testeo podemos hacer patente que hay errores en el software, pero no es posible certificar que no existen (como implicaría la segunda acepción). Para evitar confusiones preferimos utilizar los términos "testear" y "testeo/test", equivalentes al primer significado de "probar": examinar software con la intención de encontrar defectos.

1.1. ¿Qué son (buenos) casos de test?

Hay muchas definiciones sobre qué es un caso de test. Kaner [Kan03] da una lista de definiciones y añade la suya propia. Todas las definiciones, sin embargo, coinciden en que respetan los estándares IEEE [IEEE610.12], ISEB [HM+06] e ISTQB [SLS05]:

> Un *caso de test* es un conjunto de *entradas, precondiciones para la ejecución y salidas esperadas*, desarrolladas con el objetivo de testear el software.

¡No hay mucho misterio detrás de esta definición! Los casos de test pueden tener múltiples formas: ser de alto o bajo nivel, referirse al comportamiento normal del programa o a las excepciones, etc. Estos podrían ser dos casos de test en diferentes situaciones:

- Con un objeto `pila` vacío, la llamada `pila.push('x')` debe retornar `true` y dejar la pila conteniendo un único valor `'x'`.
- En la pantalla inicial del sistema, sin haber realizado ningún intento previo de autenticación, al introducir "user" como usuario y dejar la contraseña en blanco, el sistema devuelve el mensaje "introduzca una contraseña" y se muestra la misma pantalla.

Sin embargo, los casos de test que cumplen con esta definición no necesariamente son *buenos casos de test*, es decir, casos de test con una *alta probabilidad de encontrar errores importantes.*

Escuelas en el diseño de casos de test

Existen 2 escuelas con diferentes teorías sobre cómo se tienen que diseñar buenos casos de test.

- La escuela del «scripted testing».
- La escuela del «exploratory testing» (o «context-driven»).

El «*scripted testing*» se refiere básicamente al diseño de casos de test *utilizando una técnica de especificación de casos de testeo*. Además requiere que el testeador documente sus tests para hacer posible reproducirlos y verificar sus resultados.

«*Exploratory testing*» [Bach02] es un enfoque más informal del testeo de software en que *el aprendizaje, el diseño y la ejecución de los de casos de test son simultáneos*. Mientras testea el software, el testeador aprende, y eso junto con su experiencia y creatividad produce nuevos casos de test buenos. No es obligatorio que el testeador utilice una técnica de testeo específica o que documente sus tests.

En [KBN05] se describen las ventajas y desventajas del «exploratory testing» versus «scripted testing» junto con situaciones en que el «exploratory testing» no se debe utilizar. En este artículo no entraremos en esa discusión, sino que *hablaremos de técnicas que se pueden utilizar en ambos enfoques* (aunque el «exploratory testing» es menos riguroso y es dependiente de los modelos mentales que el testeador tiene del software).

En realidad, el testeo siempre es una *combinación de ambos tipos de testeo*. La tendencia hacia uno u otro depende del contexto y de factores como: disponibilidad de testeadores con experiencia; disponibilidad de una base de testeo; necesidad de poder reproducir los tests por otro testeador o herramienta, etc.

1.2. ¿Qué son técnicas para el diseño de casos de test?

Una *técnica para el diseño de casos de test* es un *método* para crear *casos de test* utilizando *información de referencia* (la base de testeo).

El gran desafío del diseño de casos de testeo es seleccionar casos de test que sean eficaces encontrando errores sin requerir un número excesivo de tests. Por lo tanto, se puede ver el testeo como un *problema de búsqueda*: estamos buscando esas combinaciones de entradas y estados del sistema que tienen alta probabilidad de encontrar errores dentro de un conjunto inmenso de entradas y estados que no podemos testear en su totalidad.

Para enfocar esta búsqueda allí donde pueden existir errores, necesitamos un *modelo* del SBT (Software Bajo Testeo), aunque sea únicamente un modelo mental como en el «exploratory testing».

Clasificaciones de técnicas para el diseño de casos de test

Las técnicas para el diseño de casos de test son diferentes en muchos aspectos:

- Utilizan diferentes *tipos de información y características del SBT* para construir estos modelos.

- Están basadas en diferentes *modelos del SBT* (el modelo del software utilizado para derivar los casos de test).
- Están destinadas a encontrar *diferentes tipos de errores* que son inherentes a la técnica y el modelo en que están basadas.

Consecuentemente, es difícil hacer una clasificación homogénea de todas las técnicas [BM04].

Muchos autores (por ejemplo [PTV01, SLS05, Bur03]) hablan del testeo de software en términos de *caja blanca y caja negra* y clasifican las técnicas en esas dos categorías. El testeo de caja blanca se realiza con conocimiento interno del sistema, mientras que las de caja negra se realizan sin conocimiento previo del sistema ni investigación de su contenido. Poniendo una analogía, si uno tuviera una caja de cartón, el testeo de caja negra serían moverlo para ver cómo suena e intentar adivinar qué hay dentro, mientras que el testeo de caja blanca consistiría en levantar la tapa y mirar dentro.

No obstante, compartiendo la opinión de Boris Beizer [Bei98], pensamos que esta categorización es incorrecta y lleva a confusión sobre cómo y cuándo se pueden aplicar las diferentes técnicas. Esta confusión está confirmada por el hecho de que muchos autores incluyen las mismas técnicas bajo diferentes categorías. Por ejemplo, el testeo de flujo de datos (o flujo de control) muchas veces está categorizado como técnica de caja blanca. Sin embargo, estas técnicas también se pueden aplicar como técnica de caja negra como se demuestra en [Bei95]; depende únicamente de la información que se utiliza para construir el modelo en que está basada la técnica. Por consiguiente, Beizer ha propuesto *otra categorización* [Bei98, Bi99]:

Testeo de comportamiento: el modelo que se utiliza para definir los grupos y/o casos de test se basa en las funcionalidades y/o el comportamiento que el software debe tener (p.ej. utilizando como base de testeo un documento de requisitos, especificaciones, conocimiento del dominio, manuales de usuario, repositorio de defectos, etc.)

Testeo de estructuras: el modelo que se utiliza para definir los grupos y/o casos de testeo se basa en los detalles internos de estructura lógica del programa (p.ej. código fuente, diagramas de flujo de datos/control, etc.)

Esta categorización es mejor porque deja entender que la técnica depende del modelo que podemos crear y no depende de si podemos o no "levantar la tapa de la caja". Sin embargo, tampoco *es de mucha utilidad a la hora de saber cómo y cuándo utilizar y seleccionar entre las diferentes técnicas* a la hora de testear un software determinado.

En el SWEBOK [BM04] se reconoce que es difícil clasificar las técnicas. En un intento de encontrar un compromiso ahí describen una clasificación basado en cómo se genera los casos de testeo (es decir, basado en: intuición del testeador, especificaciones, código, fallos, utilización o las características de la aplicación). Aunque permite clasificar todas las diferentes técnicas, seguimos sin saber cómo y cuándo utilizar y seleccionar diferentes técnicas.

2. DE TÉCNICAS A RECETAS PARA EL DISEÑO DE CASOS DE TESTEO

En la práctica, la elección de las técnicas para el diseño de casos de testeo depende totalmente de *cada situación particular de necesidad de testeo*. Estas situaciones suelen venir determinadas por:

- Disponibilidad de *diferentes tipos de información y características del SBT* (Software Bajo Testeo); por ejemplo: código fuente, un ejecutable, un documento de requisitos, especificaciones, casos de uso, documentos de diseño, manuales usuarios, informes de errores, etc.
- Un *objetivo* de testeo o el *tipo de errores que estamos buscando*; por ejemplo testear un determinado caso de uso, requisito, flujo de trabajo, camino a través del software, etc.

En lugar de definir clasificaciones en función de estos elementos, resultaría más útil ofrecer al testeador un método que le permita *elegir y aplicar las técnicas más adecuadas en cada situación, en función de la información de que disponen y del objetivos*; es decir, el equivalente a un libro de recetas.

2.1. Un símil: las recetas de cocina

Al igual que un testeador, alguien encargado de preparar una comida suele encontrarse ante una situación en la que dispone de unos *elementos de partida* (los ingredientes) y un *objetivo* (la comida preparada).

Nuestro cocinero podría realizar su trabajo valorando los múltiples modos en los que puede combinar y preparar sus ingredientes, sin más apoyo que su intuición y sus conocimientos de las técnicas existentes (cocer, freír, trocear, etc.). Sin embargo, *su trabajo resultará mucho más sencillo si dispone de un libro de recetas que le ayude, a partir de los ingredientes disponibles y de las características que debe tener el resultado* (si es almuerzo o cena, número de comensales, etc.), *a seleccionar una receta que, además, le indica todos los pasos que debe seguir y las técnicas que debe utilizar en cada momento.*

Por supuesto, un cocinero experimentado podrá hacer variaciones o prescindir totalmente de las recetas para conseguir mejores resultados; pero en la práctica las recetas darán un resultado suficientemente bueno en la mayoría de casos.

¿Cómo podemos aplicar el mismo procedimiento al diseño de casos de testeo? Para empezar, identificamos los *pasos que sigue un testeador* en cualquier situación de testeo:

a) Recopilar toda la información necesaria (fuentes, documentos de requisitos, manuales, etc.) y definir el objetivo del testeo.

b) Crear un modelo del software a testear.

c) Aplicar técnicas al modelo para generar casos de test.

En consecuencia, nuestro objetivo -dentro de un ambicioso proyecto en ejecución en el ITI (Instituto Tecnológico de Informática*)- es definir una librería o colección de recetas de diseño de casos de test* («test-case design recipes») tales que, una vez seleccionada una receta adecuada a la situación concreta, indiquen al testeador cómo realizar los pasos indicados anteriormente con el objetivo de generar casos de test.

2.2. Elementos de las recetas para el diseño de casos de testeo

Con el objetivo de ayudar al testeador a seguir los pasos identificados anteriormente, definimos los elementos que tendrán nuestras *recetas* para el diseño de casos de testeo:

1. **Cuándo utilizar esta receta**

   Una descripción que ayude al testeador a seleccionar una *receta* adecuada para la situación en que se encuentra.

2. **Cuál es el modelo del software, y cómo crearlo**

   Indicar la forma concreta que tendrá el modelo del software, y cómo crearlo a partir de la información disponible.

3. **Cómo verificar el modelo creado**

   Para comprobar que el modelo creado en el paso anterior se ha construido de manera correcta.

4. **Qué técnicas y criterios pueden utilizarse**

   Una o varias técnicas que podrán ser utilizadas por el testeador para generar los casos de test.

5. **Cómo generar los casos de test**

   Cómo aplicar las técnicas disponibles al modelo de software definido, para obtener el resultado final: los casos de test.

6. **Herramientas; referencias; ejemplos**

   Otros documentos que sirven de apoyo a la *receta*.

Veamos a continuación un ejemplo de una de esas *recetas*.

3. UNA RECETA PARA EL DISEÑO DE CASOS DE TEST: TESTEO DE DOMINIO MEDIANTE PARTICIONES EQUIVALENTES

En esta sección describimos la *receta* para el testeo de dominio mediante particiones equivalentes. Nuestro objetivo es incluir esta y otras *recetas* (testeo de dominio mediante valores límites, testeo combinatorio, transición de estados, flujo de control, flujo de datos) en nuestra página web (http://squac.iti.upv.es) a medida que transcurra el proyecto.

3.1. Cuándo utilizar esta *receta*

Esta *receta* resulta útil para testear si componentes o funciones individuales *calculan correctamente valores de salida a partir de sus entradas*, siempre que las entradas se puedan dividir en *particiones con características similares*.

Cabe señalar que estas condiciones son muy generales y, por tanto, esta receta puede usarse en multitud de casos; por ejemplo, a nivel de código fuente, para testear subprogramas (funciones, métodos); en el testeo de interfaces de usuario con varios campos, en las que cada campo es una variable de entrada con su propio dominio; etc.

Desarrollemos un ejemplo a medida que definimos la receta para comprender mejor cada uno de sus pasos:

EJEMPLO: A una discoteca pueden acceder, pagando entrada, mayores de edad (18 años o más), siendo necesario que proporcionen su número de DNI o de pasaporte. Si proporcionan un número válido de tarjeta VIP no se comprueba su edad ni pagan entrada.

El acceso la discoteca incluye una función que, a partir de la edad del cliente y del tipo de documento que presenta, calcula si debe permitir su acceso gratis o pagando entrada, o bien denegar su acceso:

```
acceso_disco (edad, núm.documento) = {gratis|pagando|denegar}
```

¿Qué casos de test podemos elegir para comprobar que no existen errores en su implementación?

Se trata de una situación a la que esta receta (testeo de dominio) se adapta perfectamente: comprobar valores de salida en función de las entradas, que además se pueden dividir en particiones con funcionamiento equivalente (mayor edad/menor edad; documento aceptado/no aceptado; etc.).

3.2. El modelo del software: los dominios divididos en particiones equivalentes

El modelo del SBT (Software Bajo Testeo) en esta receta consiste en una colección de particiones equivalentes para cada entrada al sistema; una partición equivalente es un subdominio de una entrada cuyos miembros se procesan de forma equivalente.

Cómo construir el modelo. Los pasos a seguir para obtener el modelo del software son:

i. Identificar todas las funciones (comportamiento, procesamiento, componentes, métodos, etc.) del SBT.

ii. Para cada función, determinar sus entradas y salidas.

iii. Examinar los dominios de cada entrada, es decir, los tipos de valores que estas entradas pueden tener.

iv. Dividir el dominio de cada entrada en particiones equivalentes, que serán subdominios de la entrada cuyos miembros son procesados del mismo modo por el sistema. Cada partición puede ser válida (valores contemplados por la función) o no válida (con valores no contemplados o definidos).

Los siguientes criterios pueden resultar útiles para definir las particiones equivalentes, teniendo en cuenta que deben servir como guía y no como reglas absolutas [Mey04]:

a) Si una condición de entrada especifica una *condición que debe cumplirse* ("debe ser un número de documento aceptado"), identificar una partición válida con los valores que cumplen la condición ({documentos aceptados}) y otra partición no válida con los que no la cumplen ({documentos no aceptados}).

b) Si una condición de entrada especifica un *conjunto* de valores (ej.: "números DNI, pasaporte y tarjetas VIP"), crear una partición válida para cada subconjunto de valores que sean tratados igual (en este caso, dos particiones:{DNIs y pasaportes} y {VIPs}), y una partición no válida con los valores no incluidos en ese conjunto ({documentos no aceptados}).

c) Si una condición de entrada incluye un *rango* de valores (ej.: "la edad es mayor o igual que 0"), crear una partición válida para los valores dentro del rango ($edad \geq 0$) y particiones no válidas para los valores fuera del rango ($edad < 0$).

d) Si se sospecha que los elementos dentro de una partición no serán tratados de modo idéntico, dividir la partición en otras más pequeñas (ej.: si se trata de modo diferente a mayores y menores de edad, dividimos la partición $edad \geq 0$ en dos: $0 \leq edad \leq 17$ y $edad \geq 18$).

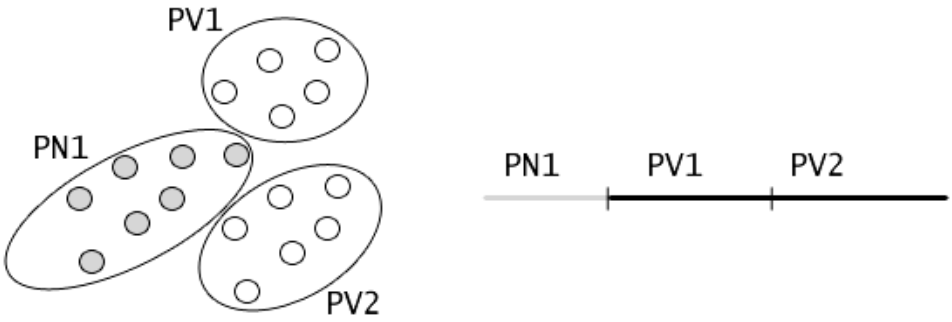

Figura 1. División en particiones, para un dominio discreto y uno continuo

De ese modo obtenemos, para cada entrada al sistema tendremos un conjunto de particiones válidas y otro de particiones no válidas.

En la función del ejemplo, las entradas son `edad` y `documento`; la salida es la de la función `acceso_disco`. Podemos definir estas particiones:

	Dominio	**Particiones válidas**	**Particiones no válidas**
edad (entrada)	números enteros	PV1: 0 ≤ edad ≤ 17 PV2: edad ≥ 18	PN1: edad < 0
documento (entrada)	números de documento	PV1: {dnis y pasaportes} PV2: {vips}	PN1: {documentos no aceptados}

Nótese que no es necesario incluir la partición `{documentos aceptados}` ya que ésta de hecho está cubierta por las particiones PV1 y PV2.

3.3. Verificación del modelo

Para verificar nuestro modelo, habrá que revisar si dentro de cada partición sus elementos son efectivamente equivalentes; es decir, cumplen que:

- ✓ Tienen las mismas características, escenario y/o procesamiento.
- ✓ Si uno de ellos produce un fallo, los otros (probablemente) también.
- ✓ Si uno de ellos no revela ningún fallo, los otros (probablemente) tampoco.

Revisando las particiones del ejemplo:

- Para la entrada `edad`, todos los valores de PV1 (menor de edad) deben ser tratados del mismo modo; lo mismo ocurre con PV2 (mayor de edad). La partición no válida PN1 incluye los valores que corresponden a edades negativas. Nótese que hemos considerado 0 como una edad válida, y que suponemos que no existe una edad máxima.
- Para la entrada `documento`, la partición PV1 incluye todos los tipos de documento aceptados en los que se debe comprobar la edad. La partición PV2 incluye los documentos aceptados en los que no es necesario comprobar la edad. La partición no válida es para entradas que no sean documentos aceptados.

Hemos revisado así que las particiones elegidas agrupan juntas entradas que se deben procesar de modo similar.

3.4. Criterio para elegir casos de test: un representante de cada partición

En primer lugar es necesario seleccionar valores concretos con los que se generarán los casos de test. Esto se realiza *eligiendo un valor representante por cada partición* definida en el paso anterior.

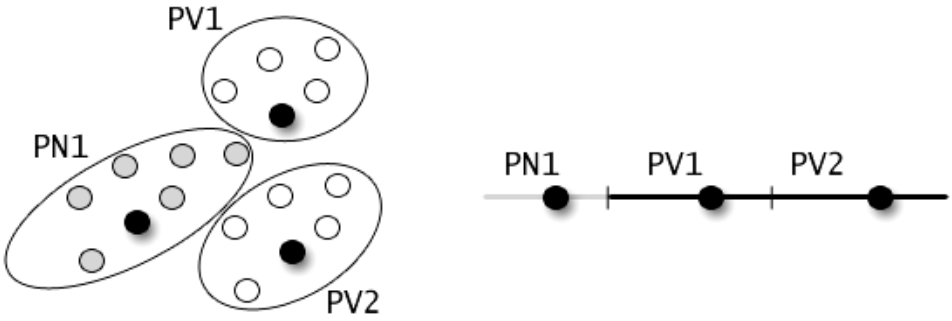

Figura 2. División en particiones, con un representante por partición, para un dominio discreto y uno continuo

El testeador utilizará sus conocimientos del sistema y su experiencia para la elección de los valores concretos. En todo caso, como anteriormente hemos verificado el modelo y sabemos que cumple con las condiciones mencionadas anteriormente, *podemos suponer que cualquier valor de la partición será representativo de las características y funciones del sistema*. Por ejemplo, si -como en la figura- hemos particionado el dominio de una entrada en tres particiones, entonces elegimos simplemente un valor de cada partición (tres valores en total).

En nuestro ejemplo, elegimos un representante cualquiera para cada partición:

	Particiones	**Representante**
edad (entrada)	PV1: 0 ≤ edad ≤ 17	10
	PV2: edad ≥ 18	25
	PN1: edad < 0	-5
documento (entrada)	PV1: {dnis y pasaportes}	PASAPORTE_VÁLIDO
	PV2: {vips}	VIP_VÁLIDO
	PN1: {documentos no acept.}	0

PASAPORTE_VÁLIDO es cualquier número de pasaporte válido que se sabe que es aceptado por el sistema (igualmente podríamos haber elegido un número de DNI); VIP_VÁLIDO es cualquier número de tarjeta VIP que se sabe que es aceptada por el sistema.

3.5. Generar casos de test

En este punto, los casos de test (es decir, los conjuntos de entradas y salidas esperadas que deberán comprobarse) se generan *combinando los valores seleccionados* en el paso anterior para cada entrada. Para combinarlas hay que tener en cuenta estas reglas:

Particiones válidas

Para generar casos de test, cada representante de las particiones válidas se combina con los representantes del resto de particiones de todos los modos posibles. Calculando el número de casos de test generados por particiones válidas tenemos que:

núm. casos de test (particiones válidas) = *multiplicación* del número de particiones válidas de cada entrada.

Recopilando los valores representantes seleccionados anteriormente:

	Valores válidos	**Valores no válidos**
edad (entrada)	10, 25	-5
documento (entrada)	PASAPORTE_VÁLIDO, VIP_VÁLIDO	0

Finalmente, combinando particiones válidas, obtenemos 4 casos de test (es conveniente asignar un identificador único a cada uno de ellos):

	Casos de test			
	C1	**C2**	**C3**	**C4**
edad (entrada)	10	10	25	25
documento (entrada)	PASAPORTE_VÁLIDO	VIP_VÁLIDO	PASAPORTE_VÁLIDO	VIP_VÁLIDO
acceso_disco (salida esperada)	denegar	gratis	pagando	gratis

Particiones no válidas

A pesar de que la función no contemple los valores de estas particiones, entendemos que deben incluirse en los casos de test para *comprobar que el sistema responde de modo robusto* (por ejemplo, devolviendo un mensaje de error en vez de quedar bloqueado) si se produce una entrada no válida; la experiencia muestra que muchos errores reales están motivados por llamadas a funciones con valores que, en teoría, "no podían suceder".

En este caso, consideramos suficiente generar *un caso de test para cada partición no válida, tomando para el resto de entradas cualquier representante válido*. No combinamos particiones no válidas entre sí, ya que los errores producidos proporcionarían pocas pistas sobre la entrada que los causa (los valores no válidos en una entrada podrían enmascarar errores producidos por valores no válidos de otras entradas). De ese modo, el número de casos de test que incluyen particiones no válidas es:

núm. casos de test (particiones no válidas) = *suma* del número de particiones no válidas de cada entrada.

De igual modo utilizando las particiones no válidas del ejemplo, generamos estos 2 casos de test:

	Casos de test	
	C5	**C6**
edad (entrada)	-5	25
documento (entrada)	PASAPORTE_VÁLIDO	0
acceso_disco (salida esperada)	? (no definido)	? (no definido)

Concluyendo con el ejemplo, hemos obtenido de este modo 4 (2x2) casos de test con particiones válidas, y 2 (1+1) con particiones no válidas. Así pues, el testeo de la función usando esta *receta* consistiría en comprobar que:

```
acceso_disco(10, PASAPORTE_VÁLIDO) = denegar    caso de test C1
acceso_disco(10, VIP_VÁLIDO) = gratis           caso de test C2
acceso_disco(25, PASAPORTE_VÁLIDO) = pagando    caso de test C3
acceso_disco(25, VIP_VÁLIDO) = gratis           caso de test C4
```

Y en comprobar que el sistema responde de modo adecuado a estas entradas no válidas (por ejemplo, permitiendo que se siga utilizando sin quedar "colgado"):

```
acceso_disco(-5, PASAPORTE_VÁLIDO)      caso de test C5
acceso_disco(25, 0)                     caso de test C6
```

3.6. Herramientas; referencias; ejemplos

Algunos de los pasos y técnicas utilizados en esta receta han sido adaptados a partir de otros encontrados en diversos libros y artículos bajo diferentes nombres: testeo de dominio [Bei90]; testeo de particiones (categorías) equivalentes [Bur03, Mey04, SLS05]; testeo de combinación de datos [PTV01]; testeo de interfaces de programa [PTV01]; testeo sintáctico [PTV01].

Un ejercicio

Como ejercicio, dejamos al lector este ejemplo para que compruebe que se adapta a esta *receta*, y que pueda seguir sus pasos para la generación de casos de test. Si desea comprobar su solución, puede consultar la página web http://squac.iti.upv.es, o bien ponerse en contacto con nosotros en squac@iti.upv.es.

Imaginemos que tenemos que testear un método `coste_administracion(Money m)` que acepta como parámetro un objeto `Money m` definido así:

```
class Money {
      private int fAmount;
      private String fCurrency;

      public Money(int amount, String currency) {
          fAmount= amount;
          fCurrency= currency;
      }

      public int amount() {
          return fAmount;
      }

      public String currency() {
          return fCurrency;
      }
}
```

El método `coste_administracion` funciona correctamente con estos supuestos:

- `m.amount() ≥ 0`
- `m.currency() in {EUR, USD, GBP, YEN}`

Este método calcula los costes de administración de este modo:

- Si el cliente quiere pagar en USD, GBP o YEN, el tiene que pagar 10% de costes de administración.
- Si el cliente paga en EUR no hay costes de administración.

¿Qué casos de test podríamos utilizar para validar `coste_administracion`?

4. CONCLUSIONES

Además de describir algunos términos básicos referentes al testeo de software, en este artículo hemos introducido y descrito algunos resultados de un ambicioso proyecto en ejecución en el ITI (Instituto Tecnológico de Informática) cuyo objetivo es *definir una librería o colección de recetas de diseño de casos de test* («test-case design recipes») que ayuden a los testeadores a seleccionar y utilizar técnicas adecuadas a una situación de testeo particular.

Hemos identificado, en primer lugar, los aspectos que determinan una situación particular de testeo. A continuación hemos definido los *elementos* de los que constarán nuestras *recetas* y, como ejemplo, hemos descrito la *receta* "testeo de dominio mediante particiones equivalentes", que puede ser ya de utilidad para los testeadores en muchas situaciones, a la espera de que desarrollemos y publiquemos otras recetas siguiendo el mismo esquema.

REFERENCIAS

[Bach02] James Bach. Exploratory Testing Explained. Chapter in The Test Practitioner, 2002. (http://www.satisfice.com/articles/et-article.pdf)

[Bei90] Boris Beizer, Software Testing Techniques, Second Edition, Van Nostrand Reinhold, New York, New York, 1990.

[Bei95] Boris Beizer, Black-Box Testing: Techniques for Functional Testing of Software and Systems, Wiley (May 1995).

[Bei98] Beizer B. The Black Box Vampire or testing out of the box. Presentations of the 15th International Conference and Exposition on Testing Computer Software (ICTCS), Washington, DC, USA, June 1998; 1.7.1–1.7.11.

[Bi99] Robert Binder. Testing Object-Oriented Systems: Models, Patterns, and Tools Addison-Wesley 1999.

[BM04] A. Bertolino y E. Marchetti. Software Testing (Chapter 5). In P. Bourque and R. Dupuis (editors), Guide to the Software Engineering Body of Knowledge SWEBOK, 2004 Version, pages 5-1 – 5-6. IEEE Computer Society, 2004.

[Bur03] Ilene Burnstein Practical Software Testing. Springer. 2003.

[CJ02] Rick D. Craig - Stefan P. Jaskiel. Systematic Software Testing. Artech House, 2002.

[Dij70] Edsger W. Dijkstra. Notes On Structured Programming, 1970, at the end of section 3, On The Reliability of Mechanisms. EWD249 at http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

[DR96] David DeLano and Linda Rising. "Patterns for System Testing", PLoP 1996.

[Fire96] Donald Firesmith "Pattern Language for Testing Object-oriented Software", by, Object Magazine, 1996.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series) 1995.

[HM+06] Brian Hambling (Ed), Peter Morgan, Angelina Samaroo, Geoff Thompson, Peter Williams Software Testing: An ISEB Foundation. ISBN: 978 1 902 505 79 4, 2006.

[IEEE610.12] IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.

[Mey04] Glenford Myers. The Art of Software Testing. John Wiley and Sons, 2004.

[MK96] John McGregor y Anuradha Kare. "Parallel Architecture for Component Testing of Object-Oriented Software", Proceedings of the Ninth International Quality Week, May 96.

[Kan03] Cem Kaner, "What is a good test case?". Software Testing Analysis & Review Conference (STAR) East, Orlando, FL, May 12-16, 2003.

[KBN05] Tim Koomen, Rob Baarda y Tutein Nolthenius, TMap Test Topics, UTN Publishers, Nederland, 2005 ISBN 90-72194-75-6.

[Sie96] Shel Siegel. Object-oriented software testing: a hierarchical approach. New York: John Wiley and Sons, 1996.

[SLS05] A. Spillner, T. Linz y H. Schaefer. Software Testing Foundations. DPUNKT.verlag, 2005. ISBN: 1933952083.

[PTV01] Martin Pol, Ruud Teunissen, Erik van Veenendaal Software Testing: A Guide to the TMap Approach. Addison Wesley Professional, 2001.ISBN: 0201745712.