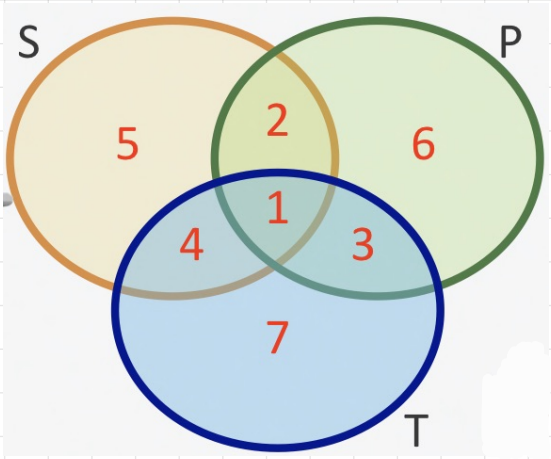
Comportamientos Especificados, programados y probados.



1 = Testeados
2 = Especificados y programados
3 = Programado y testeado
4 = Especificado y testeado
5 = Especificado
6 = Especificado y programado
7 = Testeado

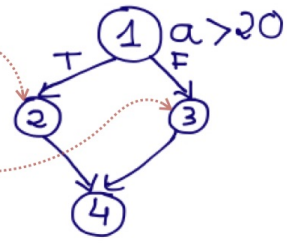
El resultado del proceso de diseño es una Tabla de casos de prueba, cada fila contiene los datos de entrada y resultado esperado de un comportamiento del programa. Las filas se rellenan usando un método de casos de prueba.

Diseño de prueba de caja blanca: Consiste en determinar los valores de entrada de los casos de prueba a partir de la implementación, el resultado esperado se obtiene siempre a partir de la especificación, los comportamientos probados podrán estar o no especificados, podremos detectar comportamientos no especificados, pero no podemos detectar no implementados.

Los métodos de casos de prueba basados en el código: Analizan y obtienen una representación en forma de grafo. Selecciona un conjunto de caminos según sea criterio y obtienen un conjunto de casos de prueba.
Podremos obtener conjuntos diferentes pero siempre serán efectivos y eficientes, estos solo suelen usarse a nivel de unidades de programa. No puede detectar todos los defectos del programa, ya que pueden faltar caminos

CfG: representación gráfica en grafo dirigido del comportamiento de una unidad. Cada nodo representa una o más sentencias secuenciales o una sola condición
- Cada nodo debe de estar etiquetado con un valor único
- Si un nodo contiene una condición anotaremos a su derecha la condición
- Todas las sentencias deben de estar representadas en el grafo

```
If (a > 20) {
    k = " valor correcto"
} else {
    k = " repita entrada"
}
```



Dos tipos de Sentencias:
- Asignacion
- Condicionales

Un conjunto de valores específicos de entrada provoca que se ejecute un camino en específico en el programa

Si ejecutamos todos los caminos independientes, estaremos ejecutando todas las sentencias del programa al menos una vez, además de que se ejecutarán todas las condiciones

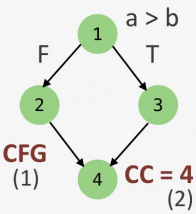
Pasos del método:

1. Construir el grafo
2. Calcular la complejidad ciclomática
3. Obtener los caminos independientes
4. Determinar datos concretos de entrada y su salida esperada.

Complejidad Ciclomática: → CC = número arcos − número nodos + 2

Método que compara dos enteros a y b, y devuelve 20 en caso de que el valor a sea mayor que b, y cero en caso contrario:

```
if (a > b) {
   result = 20;
} else {
   result = 0;
}
```



CFG (1)

CC = 4 - 4 + 2 = 2 (2)

(3) Caminos independientes

C1 = 1-3-4

C2 = 1-2-4

Tabla resultante del diseño de casos de prueba

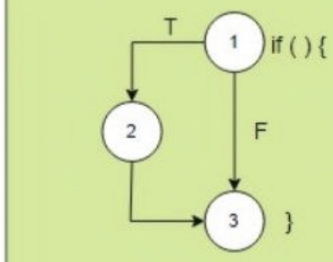
Camino	Datos Entrada		Resultado Esperado
C1	a = 20	b = 10	result = 20
C2	a = 10	b = 20	result = 0

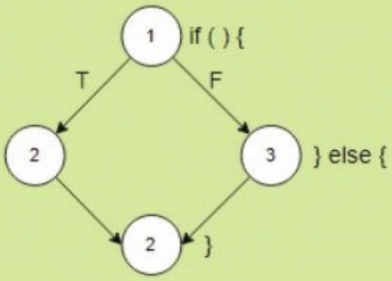
(4) Valores de entrada — Resultado esperado (5)

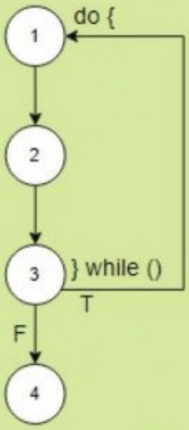
SIEMPRE son valores CONCRETOS!!!

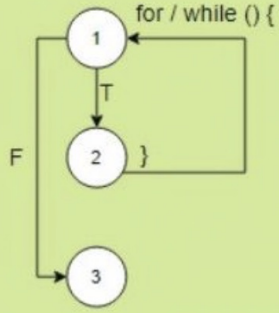
18

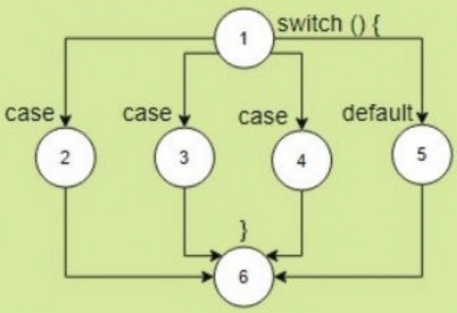
CHULETA

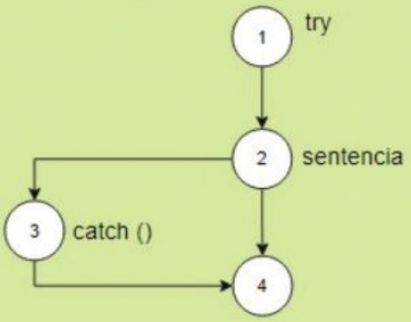








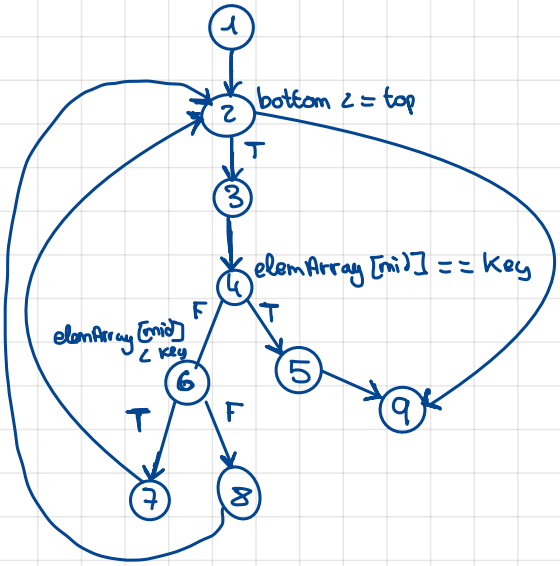




Ejercicios Repaso

```
//Asumimos que la lista de elementos está ordenada de forma ascendente
class BinSearch
public static void search (int key, int [ ] elemArray, Result r) {
  int bottom = 0;  int top = elemArray.length -1;
  int mid;  r.found= false; r.index= -1;
  while (bottom <= top) {
    mid = (top+bottom)/2;
    if (elemArray [mid] == key)  {
      r.index = mid;
      r.found = true;
      return;
    } else {
      if (elemArray [mid] < key)
        bottom = mid + 1;
      else top = mid -1;
    }
  } //while loop
} //search
} //class
```

Especificación del método search():
Dado un vector de enteros ordenados ascendentemente, y dado un entero (key) como entrada, el método search() busca la posición de key en el vector y devuelve el valor found=true si lo encuentra, así como su posición en el vector (dada por índex). Si el valor de key no está en el vector, entonces devuelve el valor found=false



Complejidad Ciclomática = 11 - 9 + 2 = 4

Camino	elemArray	key	Salida Espera	Salida Obtenida
1-2-9	[ ]	4	false	false
1-2-3-4-6-7-2-3-4-5-9	[0,1,2,3]	3	true	true
1-2-3-4-6-8-2-3-4-5-9	[0,1,2,3]	-1	true	true

➪ ➪ Ejercicio 1: *reservaButacas*

Crea la subcarpeta "**reservaButacas**", en donde guardarás tu solución de este ejercicio. Puedes crear uno o varios ficheros. En el caso de que la solución esté dividida en varios ficheros, todos ellos tendrán como nombre "**cine-<sufijo>.<extension>**", siendo <sufijo> la indicación del paso o pasos seguidos: por ejemplo cine-paso1.jpg, cine-pasos2-3.jpg,... o simplemente "**cine.<extension>**" si todo el ejercicio está resuelto en un fichero. <extension> denota el tipo de fichero: xml, jpg, png, ...

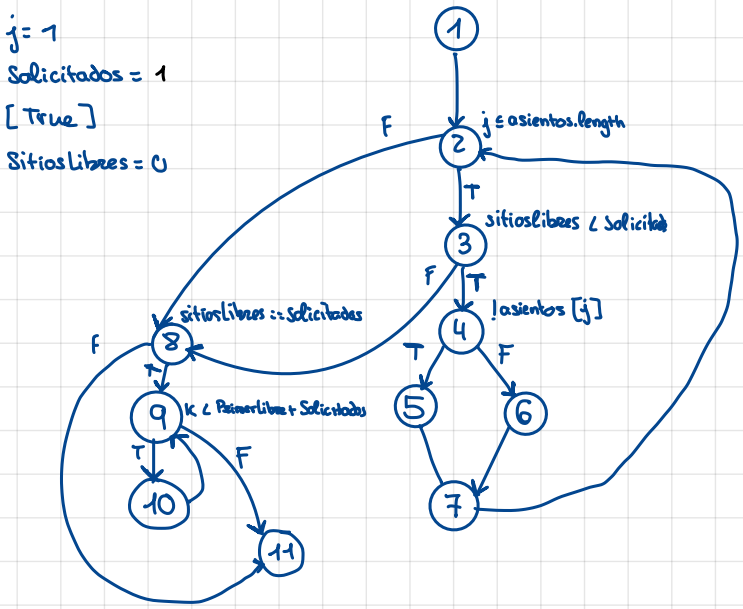
Queremos diseñar los casos de prueba para un método que, a partir de un array de booleanos, que representan una fila de butacas de un cine, y un número de asientos consecutivos a reservar, haga efectiva la reserva, siempre y cuando en la fila haya tantos asientos consecutivos libres como los solicitados, y además devuelva true o false, dependiendo de si ha sido posible hacer la reserva o no.

Cada una de las posiciones del array que representa la fila de butacas, tendrá un valor false, si la butaca correspondiente no está reservada, y true, en caso contrario.

Si el número de asientos consecutivos solicitados excede el número de butacas de la fila, se lanzará una excepción de tipo *ButacasException* con el mensaje "No se puede procesar la solicitud"

A partir de la especificación anterior, hemos implementado el siguiente código:

```
1. public boolean reservaButacas(boolean[] asientos, int solicitados) {
1.   boolean reserva = false; int j=0;
2.   int sitiosLibres =0;  int primerLibre;
3.   while ( (j< asientos.length) && (sitiosLibres < solicitados) ) {
4.      if (!asientos[j]) {
5.         sitiosLibres++;
6.      } else {
7.         sitiosLibres=0;
8.      }
9.      j++;
10.  }
11.  if (sitiosLibres == solicitados) {
12.      primerLibre = (j-solicitados);
13.      reserva = true;
14.      for (int k=primerLibre; k<(primerLibre+solicitados); k++) {
15.          asientos[k] = true;
16.      }
17.   }
18.   return reserva;
19.}
```



Complejidad ciclomática = 15 − 11 + 2 = 6

Camino	asientos []	Solicitados	SolidaEsperada	SolidaObtenida
1-2-3-4-6-7-2 3-4-5-7-2-8 9-10-11	[True, False]	1	True	True
1-2-3-4-5-7 2-3-8-9-10-11	[False, True]	1	True	True
1-2-3-4-6-7-2 8-11	[True]	1	False	False

➪ ➪ Ejercicio 2: *contarCaracteres*

Crea la subcarpeta "**contarCaracteres**", en donde guardarás tu solución de este ejercicio.

Igual que antes, puedes crear uno o varios ficheros. En el caso de que la solución esté dividida en varios ficheros, todos ellos tendrán como nombre "**caracteres-<sufijo>.<extension>**

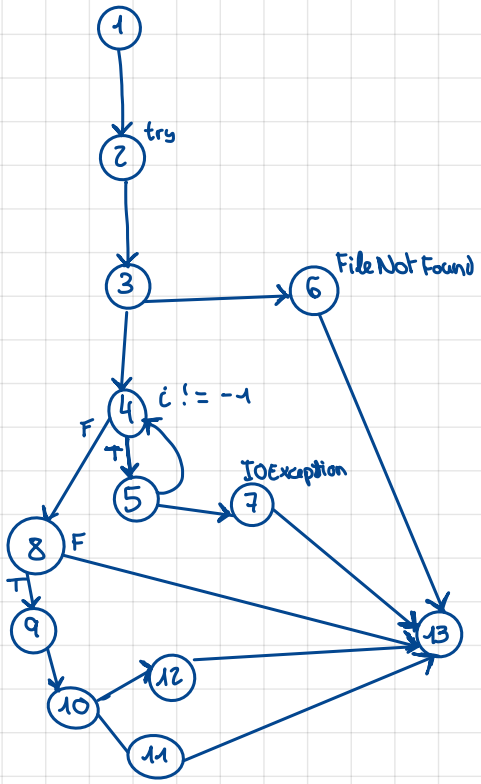
Queremos diseñar los casos de prueba para un método que, dado el nombre de un fichero de texto devuelve el número de caracteres que contiene, a menos que se produzca algún error en el tratamiento del fichero, en cuyo caso el método devolverá una excepción definida por el usuario de tipo *FicheroException* con un mensaje asociado que describe el error.

En concreto, los errores que se pueden detectar en la lectura del fichero y los mensajes de error que se devolverán asociados a la excepción son los siguientes:

Error	mensaje de error
Al abrir el fichero	*nombre_del_fichero* (No existe el archivo o el directorio)
Al realizar una lectura	*nombre_del_fichero* (Error al leer el archivo)
Al cerrar el fichero	*nombre_del_fichero* (Error al cerrar el archivo)

A partir de la especificación anterior, hemos implementado el siguiente código:

```
1.  public int contarCaracteres(String nombreFichero) throws FicheroException {
2.      int contador = 0;
3.      FileReader fichero = null;
4.      try {
5.          fichero = new FileReader(nombreFichero);
6.          int i=0;
7.          while (i != -1) {// comprobación de fin de fichero
8.              i = fichero.read();
9.              contador++;
10.         }
11.     } catch (FileNotFoundException e1) {
12.         throw new FicheroException(nombreFichero +
13.                                    " (No existe el archivo o el directorio)");
14.     } catch (IOException e2) {
15.         throw new FicheroException(nombreFichero +
16.                                    " (Error al leer el archivo)");
17.     }
18.     if (fichero != null) {
19.         try {
20.             fichero.close();
21.         } catch (IOException e) {
22.             throw new FicheroException(nombreFichero +
23.                                        " (Error al cerrar el archivo)");
24.         }
25.     }
26.     return contador;
27. }
```



CC = 17 - 13 + 2 = 6

Camino	Fichero	FileReader	Read	Close	SalidaEsperada	SalidaObtenida
1-2-3-4-5-4-8-9-10-12-13	"CorrectoC1"	...	[a, -1]	...	1	1
1-2-3-6-13	" "	FileNotFound	...	...	"No existe el archivo o el directorio"	"No existe el archivo o el directorio"
1-2-3-4-5-7-13	"Correcto"	...	IoException	...	"Error al leer el archivo"	
1-2-3-4-5-4-8-13	"Correcto"	...	[a, -1]	FicheroException	"Error al cerrar el archivo"	"Error al cerrar el archivo"

Ejercicio 3: método *reservaLibros()*

Crea la subcarpeta "**reservaLibros**", en donde guardarás tu solución de este ejercicio.

Puedes crear uno o varios ficheros. En el caso de que la solución esté dividida en varios ficheros, todos ellos tendrán como nombre "**reserva-<sufijo>.<extension>**

Se proporciona la siguiente **especificación** para el método ***reservaLibros()*:**

Se quiere llevar a cabo la reserva de una serie de libros por parte de un socio de una biblioteca, el método recibe por parámetro el login y password del empleado de la biblioteca (que será el que realice la reserva), un identificador de un socio de la misma (que es la persona que quiere reservar los libros), y una colección de isbns de los libros que quiere reservar. Solamente un empleado de la biblioteca con rol de bibliotecario puede realizar la reserva.

La reserva propiamente dicha (para cada uno de los libros) se hace efectiva en **otro método** (invocado desde realizaReserva), el cual puede lanzar varias excepciones, de forma que devolverá:

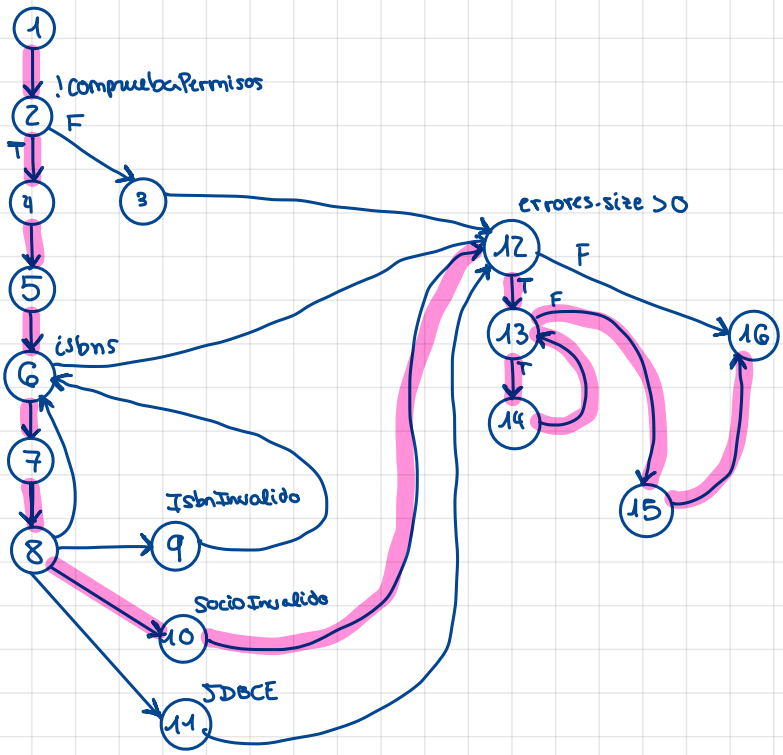
- la excepción `IsbnInvalidoException, con el mensaje "ISBN invalido:<isbn>", si el isbn` del libro que se quiere reservar no existe en la base de datos de la biblioteca (siendo <isbn> el isbn de dicho libro).
- la excepción `SocioInvalidoException, con el mensaje "SOCIO invalido",` si el identificador del socio no existe en la base de datos. En ese caso, no se podrá hacer efectiva la reserva para ninguno de los libros de la lista.
- la excepción `JDBCException, con el mensaje "CONEXION invalida",` si no se puede acceder a la BD.
- si todo va bien y se puede hacer la reserva, el método invocado termina normalmente (no devuelve nada)

El método ***reservaLibros*** termina normalmente (sin devolver nada) si todo va bien y se realiza la reserva de todos los libros de la lista pasada como parámetro. En el caso de que no se pueda hacer efectiva la reserva de algún libro, el método *reservaLibros*() devolverá una excepción de tipo *ReservaException*, con un mensaje formado por todos los mensajes de las excepciones generadas durante el proceso de reserva de cada libro, separados por ";".

Por ejemplo: suponiendo que el login y password del bibliotecario son "biblio", "1234", que el identificador de socio proporcionado existe en la base de datos, que la lista de isbns a reservar es (12345, 23456, 34567), y que el segundo y tercer isbns no están en la base de datos, el método realizaRserva devolverá como resultado una excepción de tipo *ReservaException* con el mensaje: "ISBN invalido: 23456; ISBN invalido: 34567;"

```
1. public void reservaLibros(String login, String password,
2.                           String socio, String [] isbns) throws Exception {
3.   ArrayList<String> errores = new ArrayList<String>();
4.   //El método compruebaPermisos() devuelve cierto si la persona que hace
5.   //la reserva es el bibliotecario y falso en caso contrario
6.   if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
7.     errores.add("ERROR de permisos");
8.   } else {
9.     FactoriaBOs fd = FactoriaBOs.getInstance();
10.    //El método getOperacionBO() devuelve un objeto de tipo IOperacionBO
11.    //a partir del cual podemos hacer efectiva la reserva
12.    IOperacionBO io = fd.getOperacionBO();
13.    try {
14.      for(String isbn: isbns) {
15.        try {
16.          //El método reserva() registra la reserva de un libro (para un socio
17.          //dados el identificador del socio e isbn del libro a reservar
18.          io.reserva(socio, isbn);
19.        } catch (IsbnInvalidoException iie) {
20.          errores.add("ISBN invalido" + ":" +  isbn);
21.        }
22.      }
23.    } catch (SocioInvalidoException sie) {
24.      errores.add("SOCIO invalido");
25.    } catch (JDBCException je) {
26.      errores.add("CONEXION invalida");
27.    }
28.  }
29.  if (errores.size() > 0) {
30.    String mensajeError = "";
31.    for(String error: errores) {
32.      mensajeError += error + "; ";
33.    }
34.    throw new ReservaException(mensajeError);
35.  }
36. }
```

Complejidad ciclomática = 22 - 16 + 2 = 8



Suponemos: login correcto "ppss"
password correcto "ppss"
socio correcto PPSS
isbn validos = {"1111", "2222"}

Camino	login	password	Socio	isbns	BBDD	SalidaEsperada	SalidaObtenida
1-2-3-12-13-14-13-15-16	"otro"	"otro"	PPSS	{"1111"}	True	"ERROR de permisos"	"ERROR de permisos"
1-2-4-5-6-7-8-6-7-8-9-12-13-14-13-15-16	"ppss"	"ppss"	PPSS	{"1111", "3333"}	True	"ISBN invalido: 3333"	"ISBN invalido: 3333"
1-2-4-5-6-7-8-10-12-13-14-13-15-16	"ppss"	"ppss"	ERROR	{"1111"}	True	"Socio Invalido"	"Socio Invalido"
1-2-4-5-6-78-11-12-13-14-13-15-16	"ppss"	"ppss"	Ppss	{"1111"}	False	"Conexión Invalida"	"Conexión Invalido"