

```
int fd = creat("datos.txt", 0666); //Open con las tags de debajo
int fd = open("datos.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
```

O_RDONLY: Solo lectura
O_WRONLY: Solo escritura
O_RDWR: Lectura y escritura

Hay que especificar siempre solo 1 de los 3 primeros y de 0 a 3 de los 3 últimos

O_APPEND: Se escribe siempre al final del fichero
O_CREAT: Si no existe el fichero, lo crea; si existe no tiene efecto
O_TRUNC: Si existe el fichero, lo vacía ("trunca"); si no existe sin efecto

```
int close(int fd);
//Si se pasa un array al read/write no hace falta pasar la dirección, solo el array, pero si no
se usa el carácter &
ssize_t write(int fd, const void buf[.count], size_t count); //Devuelve numero bytes escritos
ssize_t read(int fd, void buf[.count], size_t count);
off_t lseek(int fd, off_t valor, int whence); //Whence ver imagen debajo y devuelve posicion
//del puntero en bytes
```

SEEK_SET: de forma absoluta → nueva posición = valor

SEEK_CUR: relativo a pos. actual → nueva posición = pos. actual + valor

SEEK_END: relativo a final → nueva posición = final del fichero + valor

Estructura STAT: para saber las propiedades de un fichero.

- **st_mode:** el tipo de ficheros y los permisos de acceso
- **st_uid** y **st_gid:** UID y GID del propietario del fichero
- **st_size:** tamaño del fichero en bytes
- **st_blocks:** n.º de bloques de 512 bytes asignados al fichero
- **st_blksize:** tamaño de bloque recomendado para los accesos
- **st_nlink:** n.º de enlaces que referencian a este fichero
- **st_atime:** fecha y hora del último acceso de lectura al fichero
- **st_mtime:** fecha y hora del último acceso de escritura al fichero
- **st_ctime:** fecha y hora de la última modificación de los atributos del fichero

```
int stat(const char *restrict pathname, struct stat *restrict statbuf);
//Si es enlace simbolico devuelve atributos del fichero al que apunta
```

```
int fstat(int fd, struct stat *statbuf);
//Para un fichero Abierto
```

```
int lstat(const char *restrict pathname, struct stat *restrict statbuf);
//Se obtienen atributos del enlace + stat()
```

//Devuelve 0 si ok y -1 si falla

```
int unlink(const char *pathname);
```

```
//Borra el fichero que recibe como parámetro si el numero de enlaces es 0, si no borra ese
//enlace fisico
```

```
int rename(const char *oldpath, const char *newpath); //Renombrar un fichero
```

```
int ftruncate(int fd, off_t length); //Cambiar nombre fichero
```

```
//Para cambiar el descriptor de fichero a otro, dup lo asigna al primero libre (ejemplo de uso,
asignar un fichero a salida estándar)
```

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

<u>Redirección salida con dup</u>	<u>Redirección salida con dup2</u>	<u>Redirección salida sin dup</u>
<pre>d=open(fichero...); close(1); dup(d); close(d);</pre>	<pre>d=open(fichero...); dup2(d,1); close(d);</pre>	<pre>close(1); open(fichero...)</pre>
		no siempre se puede usar

```
int mkdir(const char *pathname, mode_t mode); //El modo son los permisos
```

```
int rmdir(const char *pathname); //Directorio debe estar vacío
```

```
int chdir(const char *path); //Change working directory
```

```
char *getcwd(char *buf, size_t size); //Ruta actual
```

```
DIR *opendir(const char *name); //cd y devuelve un puntero de tipo DIR
```

```
struct dirent *readdir(DIR *dirp);
```

```
//Dirp primera entrada leída en estructura dirent
```

```
struct dirent {
    ino_t      d_ino;        /* inode number */
    off_t      d_off;        /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type;    /* type of file; not supported
                             by all file system types */
    char        d_name[256]; /* filename */
};
```

```
int closedir(DIR *dirp);
```

```
int link(const char *oldpath, const char *newpath);
```

```
//aumenta el Contador de enlaces en el fichero, enlaza físicamente, apunta al mismo inodo que
//oldpath
```

```
int symlink(const char *target, const char *linkpath); //Ruta fichero y ruta enlace
```

```
mount(//todo) //montar nuevo subsistema ficheros
```

Permisos: (**S**) permiso para setear uid/guid si lo tiene el fichero ejecutado y cambia la UID/GUID efectiva del que ejecuta el fichero

Dueño	Grupo	Mundo
rwx	rwx	rwx

//SOLO SE APLICAN MÁSCARAS AL CREAR FICHEROS

```
mode_t umask(mode_t mask); //Numero 0000 -> 0777 quitar esos permisos
```

```
int chown(const char *pathname, uid_t owner, gid_t group); //Cambiar permisos de un fichero
```

```
int chmod(const char *pathname, mode_t mode); //Nuevos permisos sin tener en cuenta mascaras
```

```
//Proyección en memoria
void *mmap(void addr[.Length], size_t Length, int prot, int flags, int fd, off_t offset);
// 1) Poner NULL
// 2) Tamaño de proyección en bytes
// 3) Permisos de acceso (PROT_READ // PROT_WRITE // PROT_EXEC // PROT_NONE) combinados con | se
//     ponde qué permiso se concede
// 4) MAP_SHARED o MAP_PRIVATE
```

- **Comp:** cambios que hace proceso visibles por el resto y actualizan fichero
- **Priv:** cambios que hace proceso no visibles por resto y no actualizan fichero

```
// 5) Descriptor del fichero
// 6) Desplazamiento del fichero a partir del cual se hace la proyección

int munmap(void addr[.Length], size_t Length);
//el primer parametron es el puntero devuelto por mmap y size el tamaño a desproyectar
```

PROCESOS

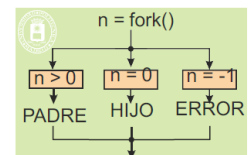
Variables de entorno:

HOME, directorio de trabajo inicial del usuario.
 LOGNAME, nombre del usuario asociado a un proceso.
 PATH, prefijo de directorios para encontrar ejecutables.
 TERM, tipo de terminal.
 TZ, información de la zona horaria

- Los procesos se identifican mediante su PID

`pid_t fork(void);` //Crea un nuevo proceso duplicando el que lo llama, el nuevo proceso es hijo

- Devuelve 0 si es el hijo
- Devuelve el PID del hijo si es el padre
- El hijo y el padre corren en espacios de memoria separados
- En el momento del fork ambos procesos tienen el mismo contenido en memoria y las escrituras en memoria, en mmaps y munmaps no afectan de un proceso al otro
- El hijo es un duplicado salvo por lo siguiente
 - o El hijo tiene su propio PID
 - o No hereda los cerrojos del padre
 - o Las señales pendientes se resetean en el hijo
 - o El hijo no hereda semáforos del padre
 - o El hijo no hereda timers ni alarmas



`exit(int value)` //Cierra el proceso con el valor de retorno de value

- `pid_t getpid(void);`
 - Devuelve el identificador del proceso.
- `pid_t getppid(void);`
 - Devuelve el identificador del proceso padre.
- `uid_t getuid(void);` `gid_t getgid(void);`
 - Devuelven el identificador de usuario real y del grupo real.
- `uid_t geteuid(void);` `gid_t getegid(void);`
 - Devuelven el identificador de usuario efectivo y del grupo efectivo.
- `int setuid(uid_t uid);`
 - Si el proceso es privilegiado (el *identificador de usuario efectivo del proceso que efectúa la llamada es el de root*) se cambian el uid real y el uid efectivo.
 - Si el proceso no es privilegiado solo se cambia el uid real.
- `int seteuid(uid_t euid);`
 - Si el proceso es privilegiado se establece el usuario efectivo.
 - Si el proceso no es privilegiado solamente puede poner como efectivo su real, por ejemplo: `seteuid (getuid);`

- `char *getenv(const char *name);`

- Devuelve el valor de la variable de entorno **name**.
 - » HOME, directorio de trabajo inicial del usuario.
 - » LOGNAME, nombre del usuario asociado a un proceso.
 - » PATH, prefijo de directorios para encontrar ejecutables.
 - » TERM, tipo de terminal.
 - » TZ, información de la zona horaria.

- `int putenv(char *string);`

- Establece el valor de las variables de entorno.

- `int exece(const char *path, char *const argv[], char *const envp[]);`
`int execlp(const char *file, const char *arg, ...);`
`int execlp(const char *file, char *const argv[]);`



Función	pathname	filename	ArgList	argv[]	environ	envp[]
execl	X		X		X	
execlp		X	X		X	
execlde	X		X			X
execv	X			X	X	
execvp		X		X	X	
execve	X			X		X
execvep		X		X		X
Letra		p	l	v		e

- Permite a un proceso pasar a ejecutar otro programa (código). El pid no cambia
- Cambia la imagen de memoria del proceso. El fichero ejecutable se especifica con nombre completo (path) o relativo (file).
- El entorno se mantiene o se cambia mediante envp.
- Mantiene en el BCP todas las informaciones que no son dependientes del programa.
 - Información de identificación, pero podrían cambiar el UID y GID efectivo.
 - Descriptores abiertos.
- Se crea una nueva pila del proceso con el entorno y los parámetros y las variables locales del main.

- Execl → Lista para pasar argumentos
- Execv → se pasa vector
 - o Acabar en p → basado en la variable de entorno path, para no pasar nombre completo
 - o Acabar en e → Coge la variable de entorno, si no coge las de la pila

```
pid_t wait(int *_Nullable wstatus); // Se espera hasta que termina un proceso hijo en status se
//guarda el estado de terminación del hijo
```

```
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options); //Se espera por un PID concreto
```

- En status se guarda si el hijo finalizó correctamente o por una señal y cómo
- Se definen varias macros para leer status
 - o WIFEXITED(status) // valor positivo si el hijo finalizó normalmente
 - o WEXITSTATUS(status) // valor devuelto por el proceso hijo (exit o return) si //finalizó correctamente
 - o WIFSIGNALED(status) //Valor positivo si el proceso finalizó por una señal
 - o WTERMSIG(status) //Número de señal que provoco la finalización

Proceso zombie: si al morir, el padre no recogió el estado de terminación. No se cierra el BCD: () en el terminal

Señales:

FORK

- El hijo hereda el armado de señales del padre.
- El hijo hereda las señales ignoradas.
- El hijo hereda la máscara de señales.
- La alarma se cancela en el hijo.
- Las señales pendientes no son heredadas.

EXEC

- El armado desaparece pasándose a la acción por defecto (ya no existe la función de armado).
- Las señales ignoradas se mantienen.
- La máscara de señales se mantiene.
- La alarma se mantiene.
- Las señales pendientes siguen pendientes.

El SO mantiene uno o varios temporizadores por proceso (en su BCP)

- El proceso activa el temporizador (UNIX: `alarm`).

El SO envía una señal al proceso cuando vence su temporizador (UNIX: `SIGALRM`).

fork: el proceso hijo NO hereda los temporizadores.

exec: Después del exec SI se conservan los temporizadores.

`SIGALRM`, señal de fin de temporización.

`SIGFPE`, operación aritmética errónea.

`SIGILL`, instrucción hardware inválida.

`SIGINT`, señal de atención interactiva (`ctrl + C`).

`SIGKILL`, señal de terminación (no se puede ignorar ni armar) (`kill -9`).

`SIGPIPE`, escritura en un pipe sin lectores.

`SIGQUIT`, señal de terminación interactiva (`ctrl + \`).

`SIGSEGV`, referencia a memoria inválida.

`SIGTERM`, señal de terminación (señal por defecto de `kill`).

`SIGUSR1`, señal definida por la aplicación.

`SIGUSR2`, señal definida por la aplicación.

`SIGCHLD`, indica la terminación del proceso hijo.

`SIGCONT`, continuar si está bloqueado el proceso.

`SIGSTOP`, señal de bloqueo (no se puede armar ni ignorar) (`ctrl + Z`).

```
int sigemptyset (sigset_t *set); //Crea un conjunto de señales vacío
int sigfillset (sigset_t *set); //Crea conjunto con todas las señales del sistema
int sigaddset (sigset_t *set, int signal); //Añade signal al conjunto
int sigdelset (sigset_t *set, int signal); //Borra signal del conjunto
int sigismember (sigset_t *set, int signal); //Determina si signal pertenece al conjunto

int kill (pid_t pid, int sig); //Envía sig al proceso pid

int sigprocmask(int how, sigset_t *set, sigset_t *oldset); //Modifica la máscara de señal activa
-   how
    o   SIG_BLOCK: añade conjunto de señales a las bloqueadas actualmente
    o   SIG_UNLOCK: elimina un conjunto de señales de las que se encuentran bloqueadas
    o   SIG_SETMASK: especifica un conjunto de señales que serán bloqueadas

int sigaction(int sig, struct sigaction *act, struct sigaction *oldact); //Acción de tratamiento
//de sig
-   La estructura sigaction tiene los siguientes elementos
    o   sa_handler:
        ▪   Función de armado → act.sa_handler = nombre_función
            •   void nombre_función (int n){...} //n es el valor de la señal tratada
        ▪   SIG_IGN → Ignorar la señal
        ▪   SIG_DFL → Acción por defecto
    o   sa_mask: señales a ignorar durante la ejecución de la función de armado a parte de sig
```

- `sa_flags: SA_RESTART` → Si la señal se produce estando en llamada bloqueante después del tratamiento sigue en la llamada. Si no se activa, termina la llamada con error.

```
int pause(void); //Bloquea el proceso hasta recibir una señal
unsigned int alarm (unsigned int seconds); //Genera la recepción de SIGALARM pasados seconds
int sleep (unsigned int seconds); //Pausa el proceso seconds
```

THREADS

- Comparten memoria y variables
- Llamadas al sistema bloqueantes
- Menor sobrecarga de ejecución que procesos pesados
- Si hay un error grave en un thread muere el programa
- Se usan variables globales para compartir datos

```
int pthread_attr_init(pthread_attr_t *attr) //Inicializa estructura atributo para crear threads
int pthread_attr_destroy(pthread_attr_t *attr) //Libera la estructura atributo
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)
//Define el heap para el thread creado mediante la estructura atributo

int pthread_attr_setdetachstate (pthread_attr_t *attr, intdetachstate)
- PTHREAD_CREATE_JOINABLE: thread no desaparece hasta que otro espere por su finalización
- PTHREAD_CREATE_DETACHED: thread independiente, libera sus recursos al terminar

int pthread_attr_getdetachstate (pthread_attr_t *attr, int *detachstate)
//Leer la función anterior

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*func)(void *),void
*arg)
//Crear el thread con la estructura atributo, en func la función a ejecutar y sus argumentos
//En la estructura pthread_t va el identificador del thread → se
void *func(void *arg){} → pthread_create(&thread,&attr,&func,&arg);

pthread_t pthread_self() //Devuelve el identificador del thread que la llama

int pthread_join(pthread_t thid, void **value)
//Para la ejecución de un thread hasta que termine el thread con identificador thid. Devuelve el
//estado de terminación

int pthread_exit(void *value) //Exit pero para threads
```

COMUNICACIÓN

1) Modelo productor-consumidor

- Un proceso genera ciertos datos que son consumidos por otro proceso
 - Hace falta un mecanismo de comunicación
 - Se debe sincronizar
 - Bloquear productor cuando mecanismo lleno
 - Bloquear consumidor cuando mecanismo vacío
 - Se soluciona mediante el uso de buffers circulares
 - Contador de huecos libres
 - Puntero al primer hueco libre
 - Puntero al dato más antiguo
 - Una tubería ya resuelve este problema

2) Modelo lectores-escritores

- Existe un recurso que va a ser compartido por varios procesos
 - Lectores → Acceden sin modificar
 - Escritores → Acceden para modificar
- Restricciones
 - Solo puede haber 1 escritor con acceso al mismo tiempo (sin lectores)
 - Múltiples lectores

3) Tuberías

Primero se debe crear la tubería y luego para leer y escribir se emplean los comandos write y read

- Como las lecturas y escrituras son atómicas y la lectura bloquea un proceso si esta vacío ya soluciona la se

```
int pipe(int pipefd[2]); // pipefd[0] → read end //pipefd[1] → write end
```

- Si se lee de la tubería
 - o Menos datos de los pedidos, devuelve esos datos sin bloquearse
 - o Si se vacía se bloquea, si no quedan descriptores de escritura abiertos devuelve 0
- Si se escribe en la tubería
 - o Si no caben datos se bloquea hasta que quepan
 - o Si no quedan descriptores abiertos devuelve (-1) y la señal SIGPIPE

```
int mkfifo(const char *pathname, mode_t mode); //Tuberia con nombre, en em mode los permisos
```

- Se abre con el servicio open convencional
 - o O_RDONLY
 - o O_WRONLY
 - o O_RDWR
- Se lee y escribe con read y write

4) Semáforos

- Usa un valor entero con valor inicial no negativo y que implementa las siguientes funciones sobre un testigo
 - o wait(s) {s = s-1; if(s<0) {Bloquear el proceso}}
 - o signal(s) {s = s+1; if(s<=0) {Desbloquear a un proceso del wait}}
 - o El numero de procesos bloqueados es -s si s < 0
- La sección critica va entre el wait y el signal
- Productor consumidor: Huecos inicializa a 8 y elementos a 0

```
/* tamaño del buffer */  
#define TAMAÑO_DEL_BUFFER 8
```

```
Productor() {  
    int posicion = 0;  
    for(;;) {  
        Producir un dato;  
        wait(huecos);  
        /* se inserta en el buffer */  
        buffer[posicion] = dato;  
        posicion = (posicion + 1) % TAMAÑO_DEL_BUFFER;  
        signal(elementos);  
    }  
}  
  
Consumidor(){  
    int posicion = 0;  
    for(;;) {  
        wait(elementos);  
        /* se extrae del buffer */  
        dato = buffer[posicion];  
        posicion = (posicion + 1) % TAMAÑO_DEL_BUFFER;  
        signal(huecos);  
        Consumir el dato extraído;  
    }  
}
```


- Lectores escritores
 - o Como el escritor es exclusivo solo debe esperar al recurso
 - o El semáforo con los lectores esta implementado para proteger la sección crítica de incrementar/decrementar el número de lectores
 - Ambos inicializados a 1

```

Lector() {
    wait(sem_lectores);
    n_lectores = n_lectores + 1;
    if (n_lectores == 1)
        wait(sem_recurso);
    signal(sem_lectores);

    < consultar el recurso compartido >

    wait(sem_lectores);
    n_lectores = n_lectores - 1;
    if (n_lectores == 0)
        signal(sem_recurso);
    signal(sem_lectores);
}

Escritor(){
    wait(sem_recurso);
    /* se puede modificar el recurso */
    signal(sem_recurso);
}

```

- Sin nombre

```

int sem_init(sem_t *sem, int pshared, unsigned int value);
//Crea el semáforo que se recoge con variable sem_t
//shared
- 0 → solo en threads creados dentro del proceso que inicia el semáforo
- Distinto de 0 → Se hereda en fork
//Val → Valor inicial

int sem_destroy(sem_t *sem); //Eliminar semaforo
int sem_wait(sem_t *sem); //Wait
int sem_post(sem_t *sem); //Signal

```

- Con nombre

```

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
//Crea el semáforo que se recoge con variable sem_t y con el nombre deseado
//Flag = 0 → No se requieren los 2 últimos parámetros
//Flag = O_CREAT → Se requieren los 2 últimos parámetros
//Mode → permisos
//Val → Valor inicial

sem_t *sem_open(const char *name, int oflag);
//Abre el semáforo que se recoge con variable sem_t y con el nombre deseado
//Flag = 0

int sem_close(sem_t *sem); //Cerrar el semáforo en el proceso

int sem_unlink(const char *name);
//Borrar el semáforo con nombre si todos los procesos lo han cerrado

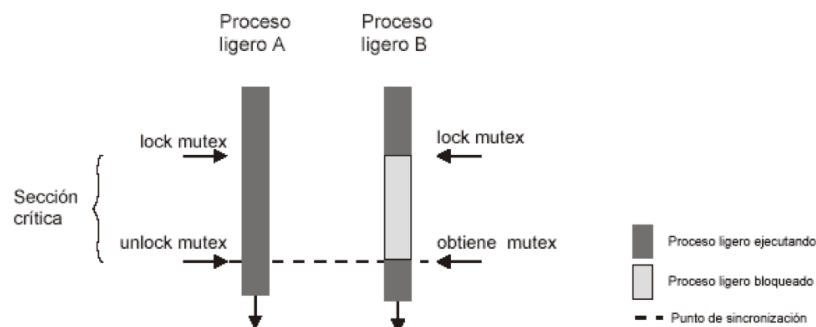
```


5) Mutex

- Se usan para threads
 - o Lock: intenta bloquear mutex → si ya esta bloqueado el proceso se bloquea
 - o Unlock: desbloquea el mutex → Se desbloquea un proceso bloqueado en el lock. La debe ejecutar el thread que lockeo el mutex

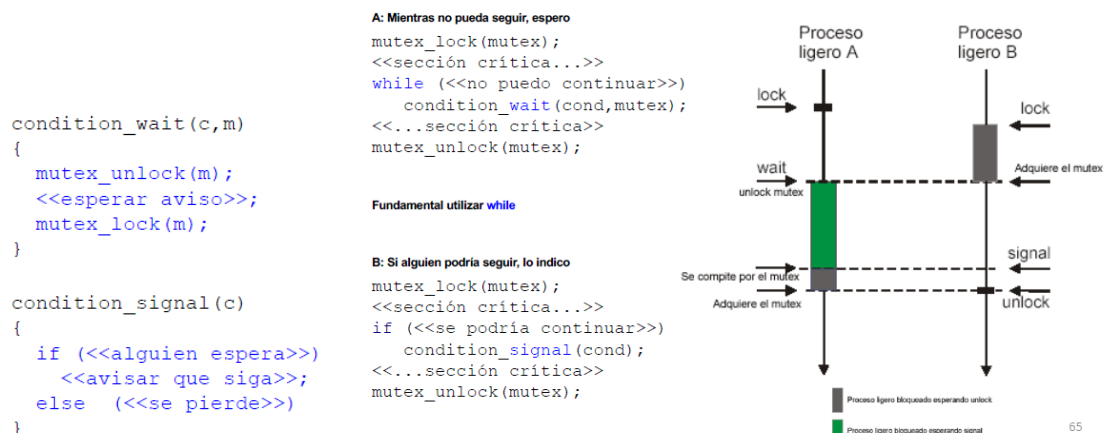
```
mutex_lock(m)
{
    if (<<no hay llave>>)
        <<esperar llave>>;
    <<abrir, entrar, cerrar y llevármela >
}

mutex_unlock(m)
{
    if (<<alguien espera>>)
        <<entregar llave>>;
    else
        <<devolver llave a cerradura >>
}
```



```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *attr);
//Crea el mutex. → Attr a NULL
int pthread_mutex_destroy(pthread_mutex_t *mutex); //Borrar el mutex
int pthread_mutex_lock(pthread_mutex_t *mutex); //LOCK
int pthread_mutex_unlock(pthread_mutex_t *mutex); //UNLOCK
```

- Se usan variables condicionales en caso de que un thread no pudiese continuar con su ejecución por no cumplir una condición
 - o Entre lock y unlock
 - c_wait → bloquea el proceso y lo expulsa del mutex permitiendo que otro proceso lo adquiriera
 - c_signal → desbloquea uno o varios procesos suspendidos en la variable condicional



```
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
//Crea la variable condicional → Attr a NULL
```

```
int pthread_cond_destroy(pthread_cond_t *cond); //Borrar variable condicional
```

```
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
//Suspende el thread hasta que se ejecute c_signal sobre la variable condicional
//Se libera el mutex hasta que se despierte por la llegada de c_signal
```

```
int pthread_cond_signal(pthread_cond_t *cond); //Señal que desbloquea el proceso en wait
```

- Productor consumidor

<pre> Productor() { int pos=0; for(num_datos) { //producir un dato lock(mutex); while(num_elem==TAM_BUFFER) c_wait(lleno,mutex); buffer[pos]=dato; pos=(pos+1)%TAM_BUFFER; num_elem++; if(num_elem==1) c_signal(vacio); unlock(mutex); } } </pre>	<pre> Consumidor() { int pos=0; for(num_datos) { lock(mutex); while(num_elem==0) c_wait(vacio,mutex); dato=buffer[pos]; pos=(pos+1)%TAM_BUFFER; num_elem--; if(num_elem==TAM_BUFFER-1) c_signal(lleno); unlock(mutex); } } </pre>
---	---

- Lectores - Escritores

```
pthread_mutex_t mutex; /* Control de acceso */
pthread_cond_t a_leer, a_escribir; /* Condiciones de espera */
int leyendo, escribiendo; /* Variables de control: Estado del acceso */
```

```

void Lector(void) {
    pthread_mutex_lock(&mutex);
    while(escribiendo != 0) /*condición espera */
        pthread_cond_wait(&a_leer, &mutex);
    leyendo++;
    pthread_mutex_unlock(&mutex);
}

```

```

void Escritor(void) {
    pthread_mutex_lock(&mutex);
    while(leyendo != 0 || escribiendo != 0) /*condición espera */
        pthread_cond_wait(&a_escribir, &mutex);
    escribiendo++;
    pthread_mutex_unlock(&mutex);
}

```

<<Lecturas simultáneas del recurso compartido>>

<<Acceso en exclusiva al recurso compartido>>

```

pthread_mutex_lock(&mutex);
leyendo--;
if (leyendo == 0)
    pthread_cond_signal(&a_escribir);
pthread_mutex_unlock(&mutex);
pthread_exit(0);
}

```

```

pthread_mutex_lock(&mutex);
escribiendo--;
pthread_cond_signal(&a_escribir);
pthread_cond_broadcast(&a_leer);
pthread_mutex_unlock(&mutex);
pthread_exit(0);
}

```

6) Cerrojos

- Para proteger ficheros que se accede desde distintos procesos
 - o Compartido → No se puede solapar con un exclusivo → Solo read
 - o Exclusivo → No se puede solapar con ninguno → Read/Write

```
int fcntl(int fd, int cmd, struct flock flockptr); //Crea el cerrojo sobre el archivo en fd
```

//CMD

- F_GETLK → Obtener si existe un cerrojo
- F_SETLK → Cerrojo no bloqueante → Si no lo consigue devuelve -1
- F_SETLKW → Cerrojo bloqueante → Si no lo consigue se bloquea

//Estructura flock

- l_type
 - o F_RDLCK → Compartido
 - o F_WRLCK → Exclusivo
 - o F_UNLCK → Elimina cerrojo
- l_whence
 - o SEEK_SET
 - o SEEK_CUR
 - o SEEK_END
- l_start → Desfase con respecto a whence
- l_len → Tamaño → Si es 0 hasta EOF
- l_pid → PID del primer elemento con lock → Solo F_GETLK

- Los cerrojos no se heredan y se pierden si el proceso cierra un descriptor cualquiera del fichero
- Lectores - Escritores

Lector	Escritor
<pre>int main(void) { int fd, val, cnt; struct flock fl; fl.l_whence = SEEK_SET; fl.l_start = 0; fl.l_len = 0; fl.l_pid = getpid(); fd = open("BD", O_RDONLY); for (cnt = 0; cnt < 10; cnt++) { fl.l_type = F_RDLCK; fcntl(fd, F_SETLKW, &fl); lseek(fd, 0, SEEK_SET); read(fd, &val, sizeof(int)); printf("%d\n", val); /*Lecturas simultáneas del recurso compartido*/ fl.l_type = F_UNLCK; fcntl(fd, F_SETLK, &fl); } return 0; }</pre>	<pre>int main(void) { int fd, val, cnt; struct flock fl; fl.l_whence = SEEK_SET; fl.l_start = 0; fl.l_len = 0; fl.l_pid = getpid(); fd = open("BD", O_RDWR); for (cnt = 0; cnt < 10; cnt++) { fl.l_type = F_WRLCK; fcntl(fd, F_SETLKW, &fl); lseek(fd, 0, SEEK_SET); read(fd, &val, sizeof(int)); val++; /*Acceso exclusivo*/ lseek(fd, 0, SEEK_SET); write(fd, &val, sizeof(int)); fl.l_type = F_UNLCK; fcntl(fd, F_SETLK, &fl); } return 0; }</pre>

7) Sockets

- Arquitectura cliente servidor
- Los enteros cuando se mandan deben convertirse de formato maquina a formato red y viceversa
 - o Host to network: htonl/htons → Short y long
 - o Network to host: ntohl/ntohs → Short y long

`int socket(int dominio, int tipo, int protocolo);` //Devuelve el sd para acceder al socket

- Dominio → AF_INET : Familia de direcciones usadas
- Tipo
 - o SOCK_STREAM → Orientado a flujo de datos
 - Con conexión
 - Asegura entrega y orden
 - Conexión continua
 - o SOCK_DGRAM → Orientado a mensajes
 - Sin conexión
 - NO fiable
 - Se envia mensaje y se cierra conexión
- Protocolo
 - o IPPROTO_TCP → STREAM
 - o IPPROTO_UDP → Datagrama

Las direcciones dependen del dominio pero los servicios no

- Estructura genérica de dirección

```
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[14];
};
```

- Estructura específica AF_INET → Debe inicializarse a 0 con bzero

```
struct sockaddr_in {
    short sin_family;           // e.g. AF_INET.
    unsigned short sin_port;    // e.g. htons(3490).
    struct in_addr sin_addr;    // la estructura struct in_addr, se declara más abajo.
    char sin_zero[8];           // no se usa, poner a cero con función bzero().
};
```

- En sin_port → puerto (1024 reservados)
- En sin_addr → dirección del host 4 octetos separados por puntos de 0 a 255

```

int inet_aton (char * str, struct in_addr * dir);

//Para pasar a formato red se recoge en dir para sockaddr y la dirección en formato
//red es devuelta

char* inet_ntoa (struct in_addr dir);

//Para pasar a formato decimal-punto

- Para convertir a binario desde dominio-punto

struct hostent * gethostbyname (char * str);
// La estructura devuelta contiene la dirección en formato red.

struct hostent {
    char h_name;           // nombre de la máquina
    char **h_aliases;      // lista de alias
    int h_addrtype;        // tipo de dirección
    int h_length;          // longitud de las direcciones
    char **h_addr_list;    // lista de todas las direcciones
};

```

Para asignar una dirección (si no lo hace el sistema) → Importante hacer en el servidor

```

int bind(int sd, struct sockaddr* dir, int long)

//Importante tener sockaddr a cero antes de usarse → luego inicializar elementos y luego
//pasarlo como parámetro. En long se pone sizeof(sockaddr_in) / variable de este tipo

struct sockaddr_in s_ain;

bzero( (char*) &s_ain, sizeof (s_ain) );

int listen (int sd, int backlog) → Sentido para TCP

//En backlog el numero de peticiones máximo pendientes que se pueden encolar

int close(int socket) //Cierra el socket

```

- Servicios basados en conexión TCP

- Servidor

```

int accept(int sd, struct sockaddr* dir, int* long)

//sd → Descriptor socket servidor

//dir → dirección del cliente

//long el sizeof() del struct

```

- Cliente

```

int connect(int sd, struct sockaddr* dir, int long)

//sd → Descriptor socket cliente

//dir → dirección del servidor

//long el sizeof() del struct

```

- Enviar

```

int send(int socket, char* mensaje, int longitud, int flags); //Flags a 0

int write(int socket, char* mensaje, int longitud);

//El int de socket es el propio del cliente o servidor → Comando Socket()

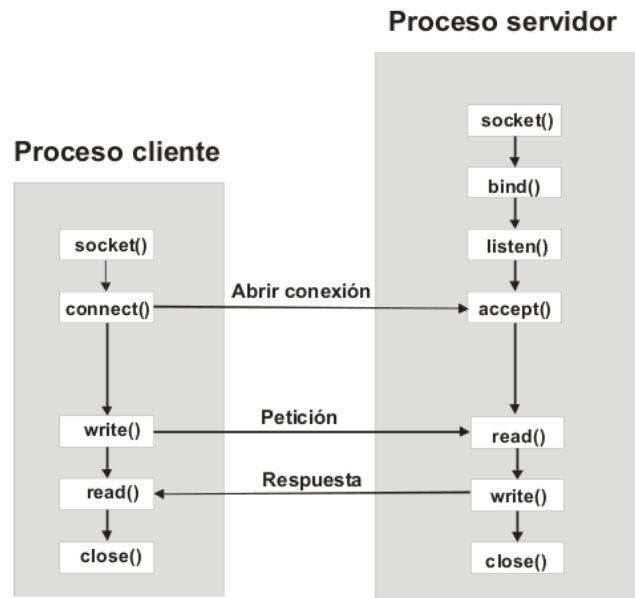
```

- Recibir

```
int recv(int socket, char* mensaje, int longitud, int flags); //Flags a 0
```

```
int read(int socket, char* mensaje, int longitud);
```

//El int de socket es el propio del cliente o servidor → Comando Socket()



- Servicios basados no conectados UDP

- Enviar

```
int sendto(int socket, char* mensaje, int long, int flags, struct sockaddr* dir, int long)
```

//El int de socket es el propio del cliente o servidor → Comando Socket()

/*dir la dirección del socket a enviar → Parametro a escribir

//Flags a 0

- Recibir

```
int recvfrom(int socket, char* mensaje, int long, int flags, struct sockaddr*dir, int * long)
```

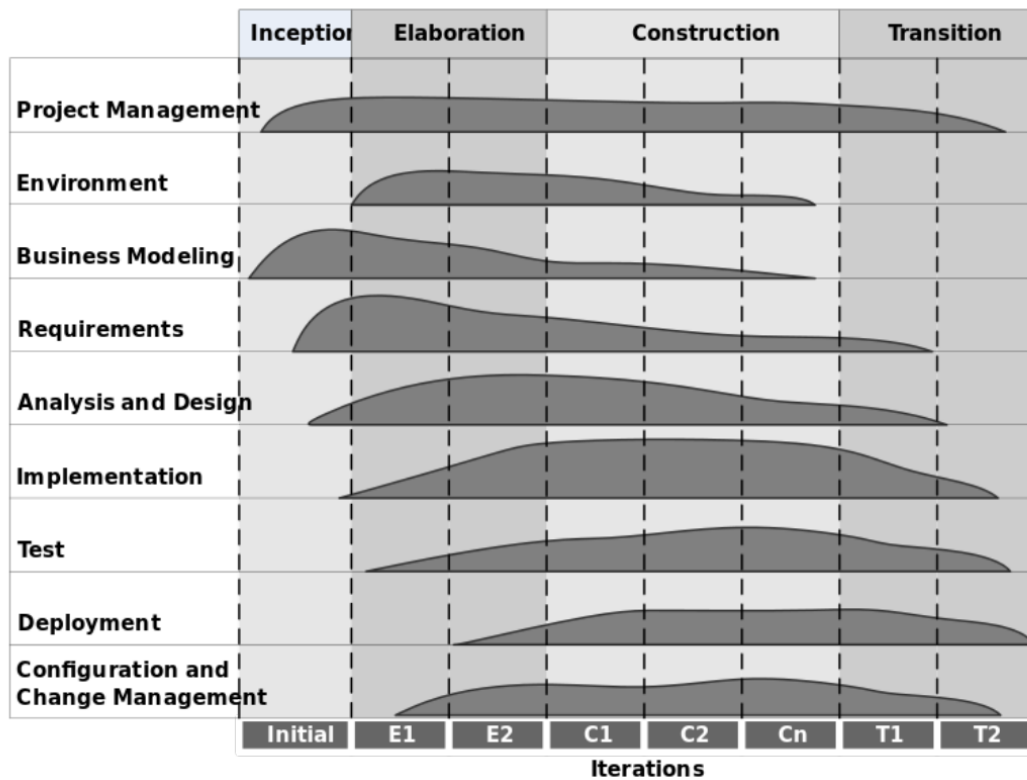
//El int de socket es el propio del cliente o servidor → Comando Socket()

/*dir la dirección del socket que ha enviado el mensaje → Parametro a leer

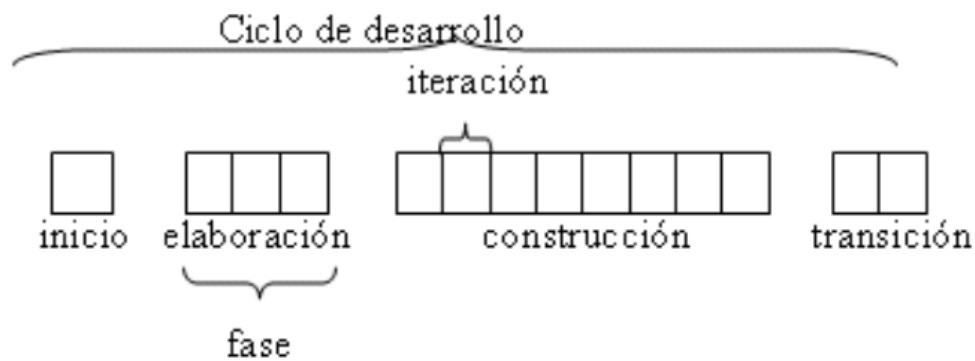
//Flags a 0

Formulario objetos:

- No olvidar destruir los objetos
 - o En el vector: delete vector[i] y luego vector.clear()
 - o Si no borrar elemento por elemento
- Tema 1:



Disciplina	Artefacto	Inicio	Elaboración	Construcción	Transición
Requisitos	Modelo de Casos de Uso	c	r		
	Visión	c	r		
	Especificaciones Complementarias	c	r		
	Glosario	c	r		
Modelado del Negocio	Modelo del dominio		c		
Diseño	Modelo de Diseño		c	r	
	Documento de Arquitectura SW		c		
	Modelo de Datos		c	r	
Implementación	Modelo de implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		



Artefactos	Comentario
Visión y análisis del negocio	Describe los objetivos y las restricciones de alto nivel, el análisis del negocio y proporciona un informe para la toma de decisiones.
Modelo de Casos de Uso	Cuenta los requisitos funcionales y en menor medida los no funcionales.
Especificación Complementaria	Documenta los requisitos relacionados con la calidad del SW.
Glosario	Terminología clave del dominio
Lista de Registros & Plan de Gestión del Riesgo darles respuestas.	Detalla los riesgos del negocio, técnicos, recursos, planificación y las ideas para mitigarlos.
Prototipos y pruebas de conceptos	Para clarificar la visión y validar las ideas técnicas.
Plan de Iteración	Describe qué hacer en la primer iteración de la elaboración.
Fase Plan de & Plan de Desarrollo de Software	Estimación de poca precisión de la duración y esfuerzo de la fase de elaboración. Herramientas, personas, formación y otros recursos.
Marco de Desarrollo	Una descripción de los pasos del UP y los artefactos adaptados para este proyecto. El UP siempre se debe adaptar al proyecto.

► **Modelo FURPS+:**

- **Funcional (*Functional*):** características funcionales de la aplicación, esto es, las utilidades que le dan al usuario.
- **Facilidad de uso (*Usability*):** factores humanos relacionados con su manejo, ayuda y documentación.
- **Fiabilidad (*Reliability*):** Capacidad de recuperación ante los fallos.
- **Rendimiento (*Performance*):** Tiempos de respuesta, precisión, eficacia.
- **Soporte (*Supportability*):** Adaptabilidad, facilidad de mantenimiento, internacionalización, configurabilidad.
- **+ (Requisitos adicionales):** Implementación, Interfaz, Operaciones, Empaquetamiento, Legales.

Esquema de casos de uso:

► Ejemplo de esquema a dos niveles:

1. El sistema debe de procesar las imágenes procedentes del microscopio
2. El sistema debe de clasificar los espermatozoides
3. El sistema debe producir un informe de resultados

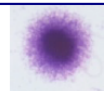

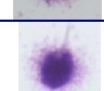
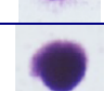
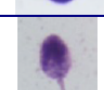


Modelo de casos de uso:

1. El sistema debe de procesar las imágenes procedentes del microscopio
 1. Adquirir las imágenes de ficheros de tipo JPG, BMP, TIF, ...
 2. Segmentar cada espermatozoide
 3. Diferenciar en cada espermatozoide el núcleo y el halo.
 4. Eliminar manchas y elementos espurios que hubiese en la imagen
 5. Sólo se tomarán espermatozoides que estén completo su halo y su núcleo.
 6. No se considerarán aquellos espermatozoides que estén juntos.
2. El sistema debe de clasificar los espermatozoides
 1. Clasificarlos en un rango numérico de 0 a 1.
 2. Correspondencia entre el rango numérico con las cinco categorías morfológicas de los espermatozoides.
3. El sistema debe producir un informe de resultados
 1. Generar histograma continuo de las frecuencias de los espermatozoides del 0 al 1.
 2. Tanto por ciento de espermatozoide del total en cada categoría. Se asociarán intervalos del rango del 0 al 1 con las categorías morfológicas de HG, HM, HP, SH y D.
 3. Datos auxiliares: Fecha y hora, número de espermatozoides total del análisis, datos del cliente (Nombre, dirección,...).
 4. Informe imprimible.



Glosario

Término	Definición e Información	Alias	Otros
Cromatina	Material nuclear (ADN + proteínas) que se tiñe de manera diferencial con ciertos colorantes (en nuestro caso: TODO LO QUE SE COLOREA, excepto la pieza intermedia y la cola)		
Tinción	Colorante que retiene la cromatina del espermatozoide. Una mayor densidad de color se traduce en una mayor masa de cromatina.		
Nucleoide	Núcleo celular parcialmente desproteínizado. Se compone de un "core" central y de un halo periférico.		
Halo	Superficie de la cromatina que se dispersa y que emana del core. Los halos tienen forma de corona circular o elíptica.		
Core:	Cuerpo central del nucleoide, que corresponde a la silueta del núcleo del espermatozoide. Los cores tienen forma circular o elíptica.	Núcleo	
Espermatozoide con Halo grande	Espermatozoide cuyo halo es igual o mayor que el diámetro menor del core.	HG	
Espermatozoide con Halo mediano	El grosor del halo está comprendido entre: mayor que 1/3 del diámetro menor del core y menor que el diámetro menor del core	HM	
Espermatozoide con Halo pequeño	El grosor del halo es igual o menor que 1/3 del diámetro menor del core	HP	
Espermatozoide Sin Halo	Espermatozoide que carece de halo de dispersión de la cromatina	SH	
Degradada(D):	Aquellos que sin mostrar halo, presentan la cabeza fragmentada en gránulos o muestran una tinción muy débil.	D	

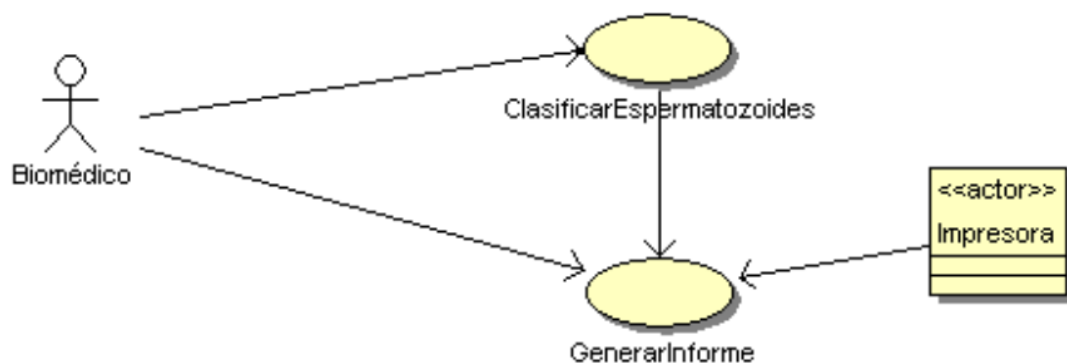
Guia EBP (Elementary Business Processes):

Evento	Actor	Objetivo
Mover Hombre	Jugador	Evitar las Esferas y lanzar Disparos
Iniciar saque	Jugador	Lanzar inicialmente las Esferas de forma aleatoria
Interacción entre Esfera-Disparo	Maquina	Se convierte en dos Esferas o desaparece si es muy pequeña
Interacción entre Esfera-Pared	Maquina	La Esfera cambia la dirección de la velocidad

Informe casos de uso:

Actores:	Jugador
Descripción:	El jugador inicia una partida. El jugador mueve un muñeco y dispara para destruir las esferas. Las Esferas rebotan contra las paredes y las plataformas. Si un disparo da a una Esfera, ésta se destruye. Si una Esfera toca al muñeco la partida se acaba.
Precondiciones:	El jugador conoce las reglas y sabe cuáles son las teclas para mover la raqueta.
Poscondiciones:	El jugador ha terminado de jugar cuando alguna esfera le toca o todas las esferas han sido destruidas.
Curso normal:	1.0. El jugador inicializa el juego. 2.0. Las Esferas rebotan contra las paredes y las plataformas y las Esferas son atraídas por la gravedad. 3.0. El jugador mueve al hombre y lanza disparos hacia arriba. 4.0. El disparo da a una Esfera y se divide en dos. 5.0. Una esfera toca al hombre y se acaba la partida.
Curso alternativo:	4.1 El disparo no da a ninguna Esfera y desaparece del escenario. 4.2 La esfera es pequeña y desaparece del escenario 5.1 Una esfera llega al suelo, no da al hombre y desaparece.
Excepciones:	2.0.E.1 Hay algún problema en el choque entre Esfera-Pared. 4.0.E.1. Hay algún problema en la división de las Esferas.
Inclusiones:	
Prioridad:	Máxima. Núcleo del sistema.
Frecuencia de uso:	Podría ser casi continuo.
Reglas de negocio:	
Requerimientos especiales:	
Suposiciones de partida:	
Notas y documentos:	

Diagrama de casos de uso

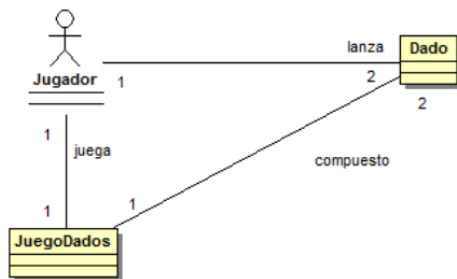


Fase de elaboración artefactos:

Artefacto	Comentario
Modelo del dominio	Es una visualización de los conceptos del dominio; es similar al modelo de información estático de las entidades del dominio.
Modelo de Diseño	Es el conjunto de diagramas que describen el diseño lógico. Comprenden los diagramas de clases de diseño, diagramas de interacción, diagramas de paquete, etc.
Documento de la Arquitectura Software	Una ayuda de aprendizaje que resume las cuestiones claves de la arquitectura y de cómo se resuelven en el diseño. Es un resumen de las ideas destacadas del diseño y su motivación en el sistema.
Modelo de Datos	Incluye esquema de bases de datos y las estrategias de transformación entre representaciones de objetos y no objetuales.
Modelo de Pruebas	Una descripción de lo que se probará y de cómo se hará.
Modelo de Implementación	Se corresponde con la implementación real (el código, los ejecutables, los manuales, las bases de datos, etc.)
Guiones de Casos de Uso, prototipos GUI	Descripción de la interfaz de usuarios, caminos de navegabilidad, modelos de facilidad de uso, etc.

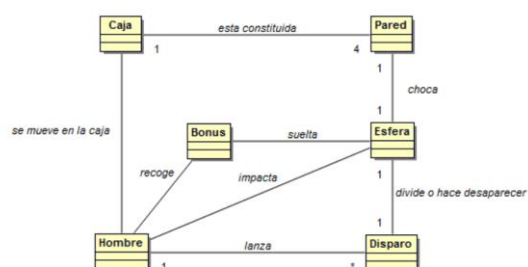
1) Modelado del dominio

Modelo Dominio = clases conceptuales + asociaciones + atributos



Pasos para crear un modelo del dominio:

1. Lista de las clases conceptuales candidatas
2. Representación en un modelo del dominio.
3. Añadir las asociaciones necesarias
4. Añadir los atributos.



*: cero o muchos
 1...*: uno o más
 1..40: de uno a 40
 5 (número exacto)
 3.7.8 (varios números)

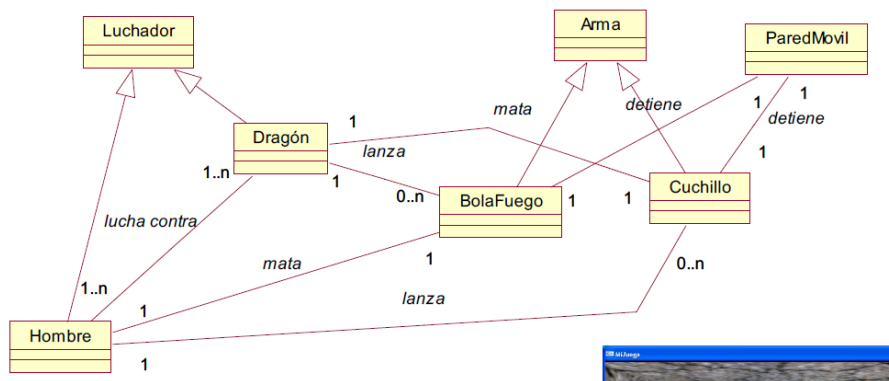
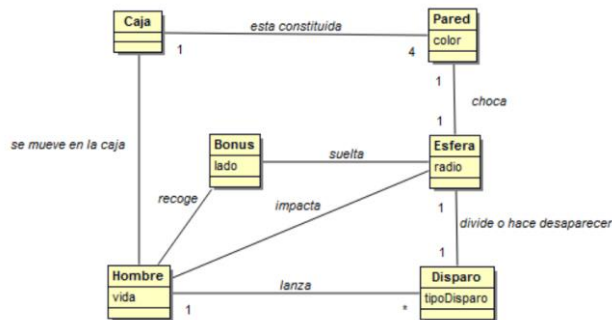
Los atributos deben ser datos simples o tipos de datos
Necesidad de registrar la información

Regla: un concepto complejo tiene múltiples instancias,
un atributo es único.

Guía para presentar un concepto como una clase y no
como un atributo:

1. Si el concepto está constituido por partes.
2. Hay operaciones asociadas con él
3. Es una cantidad con unidades.
4. Tiene otros atributos.
5. Es una abstracción de uno o más tipos (superclase – subclase, relacionado con conceptos de herencia).

No utilizar los atributos como relación entre clases
conceptuales





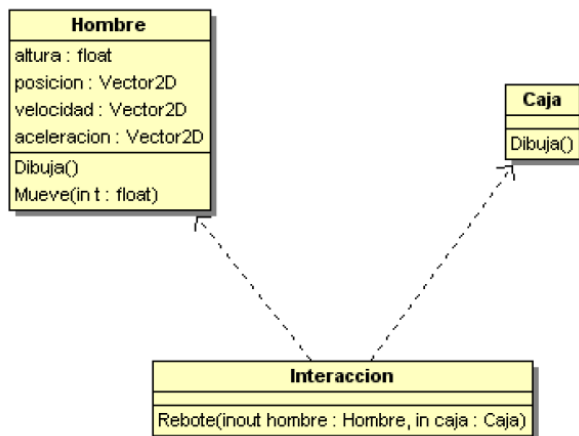
2) UML, diagrama de clases de diseño

a. Clases

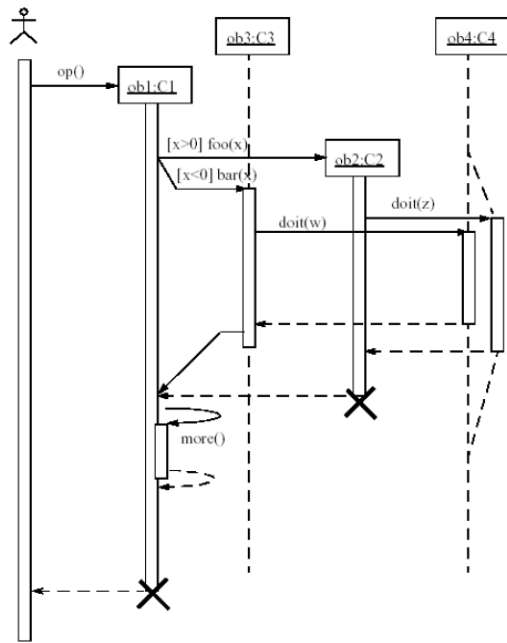
Nombre clase
<p>+ Atributo público: tipo</p> <p>- Atributo privado: tipo</p> <p># Atributo protegido: tipo</p>
<p>(Permiso) Nombre_Método (variable: tipo, ...): (retorno) tipo {static, virtual, ...} (= 0)</p>

b. Flechas

 <i>Nombre_atributo: número</i> →	Atributo de una clase con composición débil o de agregación: - Puede ser compartido por otras clases
 <i>Nombre_atributo: número</i> →	Atributo de una clase con composición fuerte: - No puede ser compartido por otras clases - La vida de la clase contenedora y contenida coinciden
→	Herencia pública: la punta la clase base y el culo la clase derivada
⋯→	Relaciones adicionales, como la amistad o la dependencia También flecha de realización → Clase que contiene una interfaz
⊕→	Clase interna - El más apunta hacia la clase externa y la punta a la clase interna (declarada en el interior de otra)

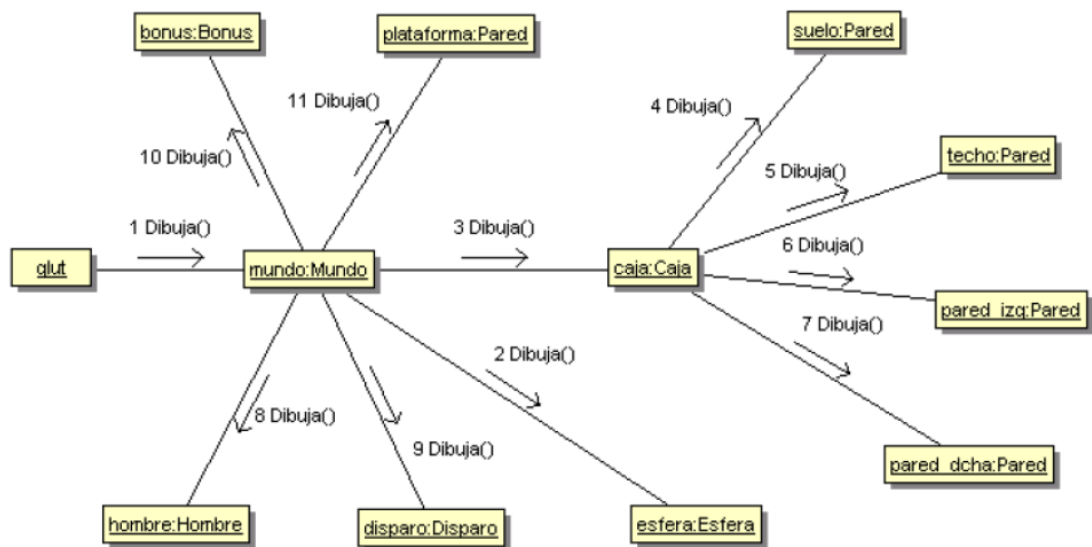


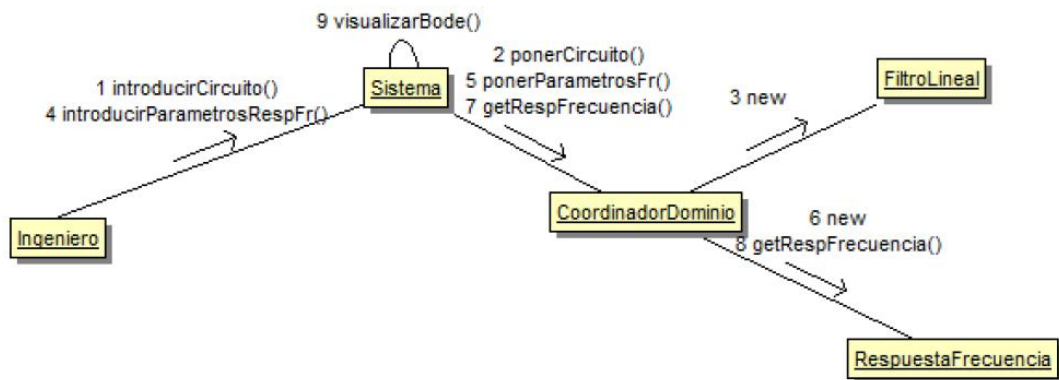
Clases asociativas: Clase que existe solo para un efecto de la clase A sobre la clase B



b. Diagrama de colaboración:

- Relación de roles
- Modela enlaces entre objetos
- Mensaje: expresión, flecha de flujo y número de secuencia





Alguna notación útil

<<abstract>>

<<virtual>>

<<vector>>

<<parameter>>

<<Interface>> → Interfaz

<<1>> → Singleton

1) Patrones GRASP (General Responsibility Assignment Software Patterns)

a. Experto en información

- i. Asignar la responsabilidad al que tenga la información
- ii. Realizar tabla de responsabilidades
- iii. Beneficios
 1. Mantiene el encapsulamiento de la información
 2. Se distribuye el trabajo entre clases haciéndolas más cohesivas y ligeras
- iv. Inconvenientes
 1. Acoplamiento entre paquetes

b. Creador

- i. Asignar a la clase B crear la clase A si se cumple uno de los siguientes casos
 1. B contiene objetos de A
 2. B se asocia con objetos de A
 3. B registra instancias de objetos de A
 4. B utiliza más estrechamente objetos de A
 5. B tiene datos de inicialización que se pasarán a un objeto de A
- ii. Esta relacionado con el patrón factoría

c. Alta cohesión

- i. Asignar una responsabilidad de manera que la cohesión permanezca alta
- ii. Clases con baja cohesión (hace tareas poco relacionadas o hace mucho trabajo)

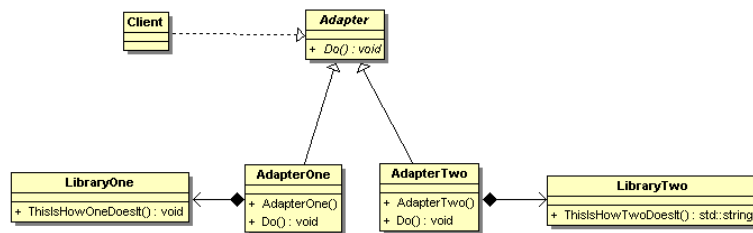
- iii. Una clase con alta cohesión tiene un número relativamente pequeño de métodos con funcionalidad altamente relacionada y no realiza mucho trabajo
- d. Bajo acoplamiento
 - i. Asignar la responsabilidad de manera que el acoplamiento permanezca bajo
 - ii. Clases con alto acoplamiento (depende de muchas clases)
 - 1. Son difíciles de mantener
 - 2. Los cambios en estas clases fuerzan cambios
 - 3. Difíciles de reutilizar
 - iii. Acoplamiento solo con elementos estables (librerías estándar)
- e. Controlador
 - i. Responsable de gestionar un evento de entrada al sistema mediante una clase que represente
 - 1. EL sistema global
 - 2. Un escenario de caso de uso
 - ii. Es una fachada del paquete que recibe los eventos externos
 - iii. No deben tener demasiadas responsabilidades
 - iv. Recibe la solicitud del servicio y coordina su realización delegando a otros objetos
 - v. Crea un objeto artificial que no procede del análisis del dominio → Fabricación pura
- f. Polimorfismo
 - i. Manejar las alternativas basadas en tipos sin realizar comprobaciones del tipo de objeto
 - ii. Interfaces + polimorfismo
- g. Indirección
 - i. Asignar la responsabilidad a un objeto intermedio entre dos o más elementos de manera que no se acoplen directamente
 - ii. La mayoría de intermediarios de indirección son fabricaciones puras
 - iii. Por ejemplo clase interacción del pang
- h. Fabricación pura
 - i. Asignar responsabilidades altamente cohesivas a una clase artificial que no represente un concepto del dominio para soportar
 - 1. Alta cohesión
 - 2. Bajo acoplamiento
 - 3. Reutilización
 - ii. Asume responsabilidades de las clases del dominio a las que se les asignaría esas responsabilidades en base al patrón Experto; pero que no se las da, debido a que disminuiría en cohesión y aumentaría la dependencia.
 - iii. Emplea el patrón indirección
 - iv. Fabricación suele ser static
- i. Variaciones protegidas
 - i. Identificar los puntos de variaciones previstas
 - ii. Asignar responsabilidades para crear una interfaz estable
 - iii. Añadiendo indirección, polimorfismo y una interfaz se consigue un sistema de variaciones protegidas VP

- iv. Las distintas implementaciones ocultan variaciones internas a los sistemas cliente
- v. Tipos
 - 1. Variaciones actuales
 - 2. Puntos de especulación → Podrían aparecer en un futuro
- vi. Constructores privados
- vii. Solo puede mandar mensajes a
 - 1. Él mismo (this)
 - 2. A un parámetro de servicio propio (visibilidad parámetro)
 - 3. A un atributo de él (visibilidad de atributo)
 - 4. A una colección de él (visibilidad de atributo)
 - 5. A un objeto creado en un método propio (visibilidad local)

2) Patrones GoF (Gangs of Four)

a. Adaptador

- i. Resolver interfaces incompatibles creando un objeto adaptador intermedio
- ii. Convertir la interfaz de una clase en otra interfaz que es la que esperan los clientes



```

#include <iostream>
#include <string>
//LibraryOne
class LibraryOne {
public:
    void ThisIsHowOneDoesIt() {
        std::cout << "Using Library ONE to perform the action\n";
    }
};
//LibraryTwo
class LibraryTwo {
public:
    std::string ThisIsHowTwoDoesIt() {
        return "Using Library TWO to perform the action\n";
    }
};

int main(int argc, char* argv[]) {
    IAdapter *adapter = 0;
    std::cout << "Enter which library you use to do operation";
    std::cout << " {1,2}: ";
    int x;
    std::cin >> x;
    if (x == 1) {
        adapter = new AdapterOne();
    }
    else if (x == 2) {
        adapter = new AdapterTwo();
    }
    if (adapter) {
        adapter->Do();
        delete adapter;
    }
    return 0;
};

///////////////////////////////////////////////////
Resultado de consola
Enter which library you wanna use to do operation {1,2}: 1
Using Library ONE to perform the action
/////////////////////////////////////////////////

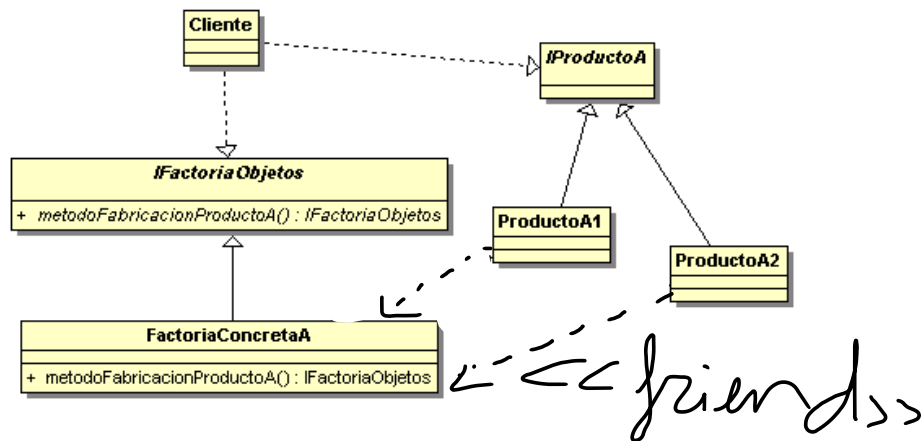
class Adapter
{
public:
    virtual void Do() = 0;
};

class AdapterOne : public Adapter
{
public:
    AdapterOne() {}
    void Do() {
        LibraryOne one;
        one.ThisIsHowOneDoesIt();
    }
};

class AdapterTwo : public Adapter
{
public:
    AdapterTwo() {}
    void Do() {
        LibraryTwo two;
        std::cout << two.ThisIsHowTwoDoesIt();
    }
};
  
```

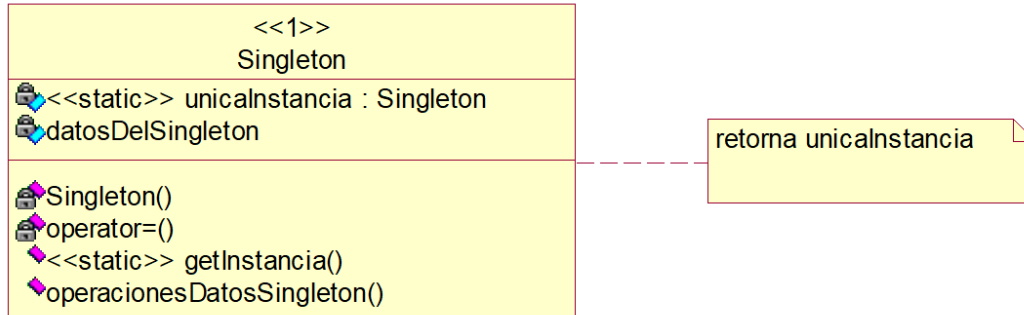
b. Factoria

- i. Crear un objeto de fabricación pura para manejar la creación
- ii. Cuando se hace variación protegida como la decisión es externa se delega la responsabilidad en la factoría
- iii. Ventajas
 1. Separación de responsabilidades en la creación
 2. Oculta la lógica de creación
 3. Permite introducir estrategias para mejorar el rendimiento de la memoria
- iv. Factoria abstracta
 1. Un sistema debe ser independiente de cómo se crean componen y representan sus productos
 2. Un sistema debe ser configurado como una familia de productos entre varios
 3. Solo revelar interfaces y no implementaciones
 4. Elementos
 - a. Cliente → Usa interfaces de fabricación y productos abstractos
 - b. Producto abstracto → Declara una interfaz para un tipo de objeto
 - c. Producto concreto → Objeto producto creado por la factoría correspondiente
 - d. Factoría abstracta → interfaz para operaciones de creación de productos abstractos
 - e. Factoría concreta → Implementa la creación de productos concretos



c. Singleton

- i. Garantizar que una clase tenga una sola instancia global mediante un método estático de la clase que devuelva el singleton
- ii. La factoría solo debe tener una instancia → Evitar que el cliente tenga el control sobre la vida del objeto



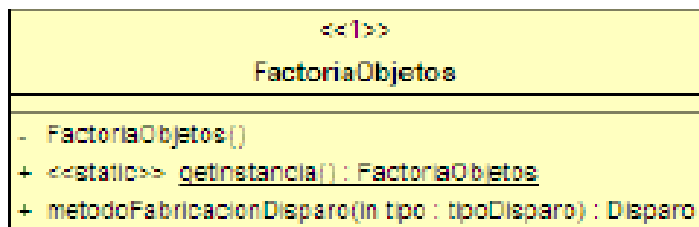
```

#include <iostream>
using namespace std;

class Singleton {
    int i; //Dato por ejemplo
    Singleton(int x) : i(x) { }
    void operator=(Singleton&);// desactivar
    Singleton(const Singleton&);// desactivar
public:
    static Singleton& getInstancia() {
        static Singleton unicalInstancia(47);
        return unicalInstancia; }
    int getValor() { return i; }
    void setValor(int x) { i = x; }
};

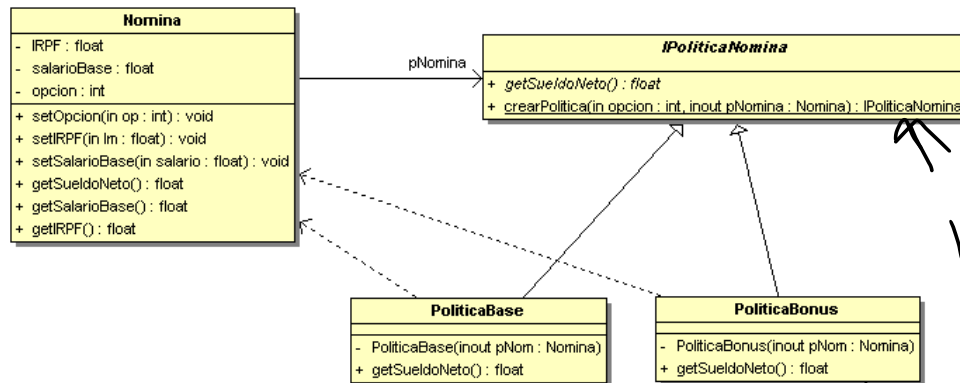
int main() {
    Singleton& s = Singleton::getInstancia();
    cout << s.getValor() << endl;
    Singleton& s2 = Singleton::getInstancia();
    s2.setValor(9);
    cout << s.getValor() << endl;
    return 0;
}
  
```

- La factoría más simple



d. Estrategia

- Diseñar algoritmos relacionados mediante clases independientes con una interfaz común
- Se define un puntero al elemento que contiene la información desde los distintos tipos de estrategia → Relación de realización
- La interfaz tiene la responsabilidad de crear la estrategia → Estrategias constructores privados



```

class Nomina;
class IPoliticaNomina{
public:
    virtual float getSueldoNeto() = 0;
    static IPoliticaNomina * crearPolitica(int, Nomina *);
};

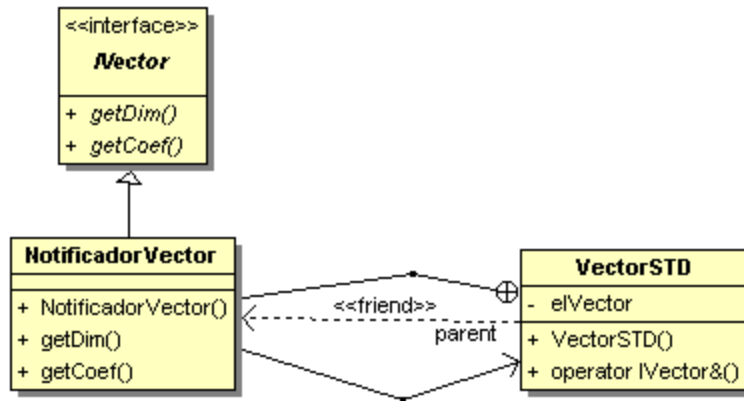
class Nomina {
    float IRPF, salarioBase; int opcion;
public:
    void setOpcion(int op) {opcion = op;}
    void setIRPF(float Im) {IRPF = Im;}
    void setSalarioBase(float salario) {salarioBase = salario;}
    float getSueldoNeto() {
        float resultado = -1;
        IPoliticaNomina *pPolitica =
IPoliticaNomina::crearPolitica(opcion, this);
        if(pPolitica){
            resultado = pPolitica -> getSueldoNeto();
            delete pPolitica;
        }
        return (resultado);
    }
    float getSalarioBase() {return salarioBase;}
    float getIRPF() {return IRPF;}
};

class PoliticaBase : public IPoliticaNomina {
    friend IPoliticaNomina;
    Nomina *pNomina;
    PoliticaBase(Nomina * pNom): pNomina(pNom) {}
public:
    float getSueldoNeto(){ return pNomina->getSalarioBase()*(1-(pNomina-
>getIRPF()/100));}
};

#define BONUS 1.35
class PoliticaBonus : public IPoliticaNomina {
    friend IPoliticaNomina;
    Nomina *pNomina;
    PoliticaBonus(Nomina * pNom): pNomina(pNom) {}
public:
    float getSueldoNeto(){
        return pNomina->getSalarioBase()*BONUS*(1-(pNomina-
>getIRPF()/100));
    }
};
    
```

e. Composición

- Basado en las clases internas
- Evitar el acoplamiento mediante el encapsulado de las partes heredadas



```
class IVector{
public:
    virtual int getDim() = 0;
    virtual float getCoef(int) = 0;
};
```

```
class VectorSTD{
    vector<float> elVector;
    class NotificadorVector;
    friend class VectorSTD::NotificadorVector;
    class NotificadorVector : public IVector {
        VectorSTD *parent;
    public:
        NotificadorVector(VectorSTD *p):parent(p){}
        int getDim()
        {return parent->elVector.size();}
        float getCoef(int i)
        {return parent->elVector[i];}
    } elNotificador;
public:
    VectorSTD(int dim,float *pV):elNotificador(this){
        for(int i=0;i<dim;i++)
            elVector.push_back(pV[i]);
    }
    operator IVector &(){return elNotificador;
};
```

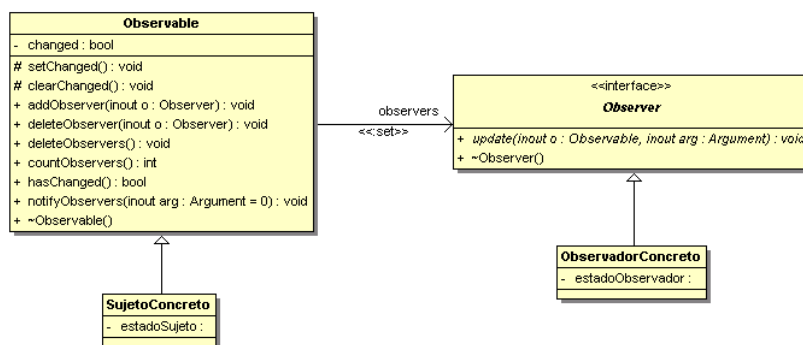
```
#include <vector>
#include <iostream>
using namespace std;
class IVector{};
class VectorSTD();

void showVector(IVector &elVector){
    for(int i=0; i<elVector.getDim();i++)
        cout << elVector.getCoef(i) <<endl;
}

int main(){
    float vector[]={1.0f,2.0f,3.0f};
    VectorSTD miVector(3,vector);
    showVector(miVector);
    return 0;
}
```

f. Observador

- Solucionar el problema de la existencia de múltiples suscriptores a un emisor que dinámicamente puede registrar suscriptores
- Dependencia uno a muchos
- El sujeto y Observador concretos son clases internas




```

class Observable;
class Argument {};
class Observer {
public:
    // Called by the observed object, whenever
    // the observed object is changed:
    virtual void update(Observable*, Argument*) = 0;
    virtual ~Observer() {}
};

class Observable {
    bool changed;
    std::set<Observer*> observers;
protected:
    virtual void setChanged() { changed = true; }
    virtual void clearChanged() { changed = false; }
public:
    virtual void addObserver(Observer& o) {
        observers.insert(&o);
    }
    virtual void deleteObserver(Observer& o) {
        observers.erase(&o);
    }
    virtual void deleteObservers() {
        observers.clear();
    }
    virtual int countObservers() {
        return observers.size();
    }
    virtual bool hasChanged() { return changed; }
    // If this object has changed, notify all observers:
    virtual void notifyObservers(Argument* arg = 0) {
        if(!hasChanged()) return;
        clearChanged(); // Not "changed" anymore
        std::set<Observer*>::iterator it;
        for(i = observers.begin(); i != observers.end(); i++)
            (*it)->update(this, arg);
    }
    virtual ~Observable() {}
};

```

3) Patron decorator

- a. Añadir dinámicamente funcionalidad a un objeto
- b. Permite no tener que heredar sucesivas clases que hereden de la primera
- c. Si se quisiera añadir un nuevo complemento a opciones coche como hay un puntero a la base en cada complemento se va llamando recursivamente a coche que mediante la herencia desenrolla los complementos

