Librería de combate

Introducción

Con este artículo comienza una serie de tres artículos en los que desarrollaremos juntos una librería de combate. Con esta librería de combate, podremos incluir en nuestros juegos características propias de los juegos de rol, en los que se libran combates basándose en una serie de puntuaciones de vida, de capacidad de ataque ... etc.
Al final de la serie, publicaré en mi página web [1] la citada librería de combate, completa, junto a un ejemplo en el que se mostrarán sus posibilidades. Para cada uno de los artículos, aparecerá también la versión descargable de la librería en su estado en ese momento.
El propósito del ejercicio es comprender cómo funciona inform, conjuntamente con la librería informATE!.
En este artículo se presuponen conocimientos mínimos de InformATE!, que pueden obtenerse del tutorial DocumentATE!.

Definiendo el problema

El propósito del ejercicio es conseguir soportar las múltiples partes de las que consta un combate: básicamente gestión de armas y gestión de atacantes (estadísticas de vida, fuerza, capacidad de ataque ...).
En el artículo de hoy, trataremos de solucionar la primera parte del problema, es decir, el soporte para el uso de armas.

Las armas en tu aventura

En primer lugar, es necesario indicar cómo vamos a incluir el soporte de armas en nuestra librería. Aunque cierto es que de poco sirve definir armas si no tenemos pensado cómo va a afectar al resto de los entes del juego (PSI, jugador). Pero por algún sitio hay que empezar, y desde luego, las armas serán los elementos más importantes.

La clase básica de armas es la siguiente (al menos en esta fase inicial):

```
! ======= Clases de armas ===================================
class arma
with
      descripcion "Podrías utilizarlo como arma."
;
```

No dice gran cosa, eso es cierto. Es más, es probable que quien utiliza una de estas armas, tenga su propia descripción, así que esta clase bien podría ser:

```
class arma;
```

Parece malgastar demasiado esfuerzo el utilizar esta clase 'arma', cuando no aporta nada. Lo que permite en realidad es establecer una jerarquía de tipos de clases. 'arma' por sí misma no hace nada, pero permite derivar otras clases de ella.

```
class arma_defensiva
class arma
private
      poder_defensivo 0
with
      devDefensa [; return self.poder_defensivo; ]
;
```

La clase 'arma_defensiva' deriva de 'arma'. Esto se consigue con la segunda línea "class arma", que indica que arma_defensiva es lo mismo que "arma", y algo más que definimos a continuación. Esto es lo que se conoce como herencia. Aunque no es exacto: arma_defensiva podría redefinir alguna propiedad de "arma" (por ejemplo creando una nueva descripción), con lo cual la herencia no significaría sólo añadir más características, sino también redefinir (o sobreescribir) otras ya existentes.
Otra parte importante es el uso de "private". El sentido que tiene utilizar private es que las propiedades bajo private no son accesibles desde el exterior, sólo desde el mismo objeto. Por ejemplo, si tengo un objeto pistola, ésta puede acceder a sus características private, pero no el objeto jugador, por ejemplo. La razón de crear un método "devDefensa()" es poder acceder a este valor. Entonces ... ¿por qué private?. La razón es que estamos convirtiendo a "poder_defensivo" en una propiedad de sólo lectura, podemos leer su contenido a partir de devDefensa(), pero no hay forma de modificar "poder_defensivo". Esto parece tener bastante sentido. Protegemos esa propiedad del objeto porque creemos que es algo que una vez que se crea el objeto, no va a volver a cambiar.

```
class arma_ofensiva
class arma
private
      poder_ofensivo 0
with
      devOfensa  [; return self.poder_ofensivo; ]
;
```

"arma_ofensiva", de la misma forma que "arma_defensiva", se va a crear a partir de "arma". La utilidad de esto es que, sea el arma defensiva u ofensiva, siempre podremos saber si es un arma sólo con hacer (objeto ofclass arma).

```
class arma_mixta
class arma_defensiva
class arma_ofensiva
;
```

"arma_mixta" (por ejemplo, una espada) permite el ataque y la defensa. Esto es lo que se conoce como herencia múltiple. Un "arma_mixta" tendrá por tanto, devDefensa() y devOfensa(), además de, por supuesto, poder_defensivo y poder_ofensivo.

```
class armaDeFuego
class arma_ofensiva
private
      balas_al_disparar 1,
      maxnumbalas 6,
      calibre 0,
      apuntado nothing,
      conseguro true,
      disparado_en_turno 0,
      disparado false,

      heSidoDisparado [x;
            self.disparado_en_turno = turnos;
            objectloop(x ofclass armaDeFuego) {
                  x.noDisparado();
            }
            self.disparado = true;
      ]
with
      descripcion "Pues sí, es un arma de fuego.",
```

Un tipo interesante de "arma_ofensiva" es el armaDeFuego. Es decir, una pistola o un fusil. En este caso, definimos varias propiedades, que son el número de balas gastadas cada vez que se dispara el arma, el calibre de la misma, a dónde apunta, el último turno en que fue disparada, y si ha sido la última arma de fuego en ser disparada.
"heSidoDisparado()" es un interesante método que será ejecutado por la propia arma cada vez que ésta sea disparada. Así, no tiene sentido que otro objeto ejecute ese método, por lo que lo colocamos en la parte privada de la clase. El método guarda el turno en el que el arma fue disparada por ultima vez, y recorre el resto de armas de fuego para colocar su propiedad de disparo a "false" (falso). Finalmente, coloca su propia propiedad de disparo a "true" (verdadero).
"Objectloop()" es un método muy útil que recorre todos los objetos del juego. Podemos añadir una condición, así "objectloop(x ofclass armaDeFuego)" sólo tiene en cuenta aquellos objetos de la clase armaDeFuego. Es parecido a un bucle for, en el cuerpo del cual "x" representa a un objeto distinto en cada vuelta.

```
devBalasAlDisparar [;
      return self.balas_al_disparar;
],

devDisparadoEnTurno [;
      return self.disparado_en_turno;
],

noDisparado [;
      self.disparado = false;
],

devNumBalas [numbalas x;
        numbalas = 0;

        objectloop(x in self) {
                if (x ofclass BalaArmadeFuego)
                        numbalas++;
        }

        return numbalas;
],
devApuntado [; return self.apuntado; ],
devMaxBalas [; return self.maxnumbalas; ],
devCalibre [; return self.calibre; ],
```

A continuación, tenemos una serie de métodos que nos permiten leer una serie de propiedades de los futuros objetos armaDeFuego. "noDisparado()" es un método que resetea la propiedad disparado a falso. "devApuntado()" devuelve el objetivo apuntado por el arma ... y así sucesivamente. "devNumBalas()" es un método que recuenta las balas que el arma contiene. Más tarde veremos la clase bala. Así, las balas que el arma tiene disponibles estarán dentro de ella, y cada vez que dispara, una de estas balas es eliminada. Así, para saber el número de balas en el arma, sólo será necesario contar los objetos dentro de ella (ya que asumimos que todos son de la clase bala, pues la única forma de meter objetos en un arma es a través del método de carga de balas).

```
apuntar [;
           if (~~(self.antesApuntar()))
                  rfalse;

           if (self in jugador)
                  self.apuntado = otro;
           else {
```

```
                    print "No tienes ", (el) self, ".^";
                    self.apuntado = nothing;
                }

            return self.despuesApuntar();
    ]
```

Comenzamos con las capacidades básicas de las armas. La primera de ellas es que se pueden apuntar a otro objeto. Veremos en la sección gramática que se creará una acción nueva, "apuntar", que permitirá hacer que un arma tenga un objetivo. En esa acción, "uno" será el arma, y "otro" será el objetivo a marcar.

Damos una oportunidad al programador de parar la acción, con un método redefinible que es "antesApuntar()". Si devuelve falso, entonces la acción no se llevará a cabo. Quizás sea un fusil con mira telescópica defectuosa, por ejemplo. "despuesApuntar()" permite realizar algo justo después de apuntar el arma. "devApuntado()" nos permitirá saber dónde está apuntando el arma.

```
    despuesApuntar [;
        if (self.apuntado~=nothing)
                print (_El) self, " apunta a ",
                      (el) self.apuntado, ".^";
        else print (_El) self, " no apunta a nada.^";
        rtrue;
    ],
```

Por ejemplo, en este caso aprovechamos ese método para poder imprimir un mensaje de a dónde apunta el arma.

```
    despuesManipularSeguro [;
        if (self.conseguro)
              print (_El) self, " tiene el seguro cerrado.";
        else  print (_El) self, " tiene el seguro abierto.";
        rtrue;
    ],

    quitaElSeguro [;
          if (~~self.antesManipularseguro())
              rfalse;

          self.conseguro = false;
          return self.despuesManipularSeguro();
    ],

    ponElSeguro [;
          if (~~self.antesManipularseguro())
              rfalse;

          self.conseguro = true;
          return self.despuesManipularSeguro();
    ],
```

Algo muy parecido sucede cuando tratamos de quitarle el seguro al arma. Pero ... ¿cómo encadenamos las acciones con estos métodos? ... lo vemos a continuación:

```
    antes [;
          quitarSeguro: return self.quitaElSeguro();
          ponerSeguro:  return self.ponElSeguro();
          apuntar:      return self.apuntar();
          vaciar:       return self.descargaTodo();
          cargar:       return self.cargaTodo();
```

```
            disparar:     return self.dispara();
      ],
```

El método "antes()" es la clave. El método "antes()" es el que permite capturar las acciones antes de que produzcan sus consecuencias por defecto. Ahora vemos claramente cómo el arma responderá a las acciones que le pida el usuario.

```
      antesDisparo [; rtrue; ],

      despuesDisparo [hubodisparo;
                if (hubodisparo)
                        print "BANG!^";
                else {
                        print (_El) self;

                        if (self.conseguro)
                                print " tiene el seguro puesto, ";

                        if (self.devNumBalas()==0)
                                print " no tiene balas, ";

                        if (self.devApuntado()==nothing)
                                print " y no apunta a ningún sitio.^";
                        else print " aunque apunta a ",
                              (el) self.devApuntado(), ".^";
                }

                return hubodisparo;
      ],

      dispara [disparo toret i;
            disparo = false;

            if (~~(self in jugador)) {
                  print "No tienes ", (el) self, ".^";
                  rfalse;
            }

            ! Gancho a la rutina del usuario
            if (~~self.antesDisparo()) rfalse;

            if (self.devNumBalas() > 0) {
              if ((self.apuntado~=nothing) && (~~self.conseguro))
              {
                  self.heSidoDisparado();
                   disparo = true;
                  for (i=0:i<self.devBalasAlDisparar(): i++)
                      self.quitaBala(child(self), LugarBalasPerdidas);
              }
            }

            toret = self.despuesDisparo(disparo);

            if (toret)
                <atacar self.apuntado>;

            if (disparo)
                self.apuntado = nothing;

            return toret;
      ],
```

Tenemos los métodos "antesDisparo()" y "despuesDisparo()". En el segundo, aprovechamos para imprimir un mensaje significativo, acerca del disparo que acabamos de realizar.
En cuanto al método disparar en sí, primero comprobamos que el arma realmente esté en posesión del jugador, y entonces ejecutamos "antesDisparo()" para saber si podemos disparar. "antesDisparo()" siempre devuelve verdadero, pero podemos modificarlo para que haga otra cosa, si queremos (por ejemplo, que el arma se encasquille temporalmente).
Si es posible disparar (según el seguro, si hay balas y si estamos apuntando a algo), entonces realizamos el disparo. Nos cargamos de dentro del arma el número de balas necesario.
Lanzamos a continuación la acción de atacar al objetivo. Así, tendremos una forma de "disparar" a los objetos que no sean en principio guerreros, al menos, según lo que nuestra librería un "guerrero".
Cada objeto puede entonces reaccionar a la acción "atacar" o permitir que discurra la acción por defecto, "la violencia no es la solución".

Llegados a este punto es necesario explicar todo lo concerniente a las balas.

```
Object LugarBalasPerdidas
with
        descripcion "¿Pero ... cómo has llegado aquí?"
has luz;
```

En esta localidad vamos a guardar las balas que vayamos gastando. Hubiera sido posible realizar, simplemente, un "remove bala", eliminando la bala. Pero, de esta forma, se pueden contabilizar las balas e incluso "resucitarlas" si es necesario.

```
class BalaArmaDeFuego
private
      calibre 0
with
      devCalibre [; return self.calibre; ],
      compatible [arma;
            if (~~(arma ofclass armadefuego))
                  rfalse;

            if (arma.devCalibre() == self.devCalibre())
                  rtrue;
            else rfalse;
      ],
      antes [;
            ponerSobre: if (otro ofclass ArmaDeFuego)
                              return otro.CargaUnaBala();
                        else "No puedes cargar ", (el) self, " en eso.";
      ],
      descripcion [;
            print "Una bala del calibre ", self.devCalibre(), ".^";
            rtrue;
      ]
;
```

Las balas son objetos al margen de las armas. Tienen su propio calibre, y deseamos cargar cada arma con las balas de su propio calibre. Así, un método de "bala" nos dice, pasándole un "arma", si ésta es compatible. Las balas pueden meterse una a una en el arma, por eso tenemos la reacción a ponerSobre en el método antes(). Cuando hacemos "mete uno en otro" o "pon uno en otro", entonces se dispara la acción ponerSobre.

```
      antesCargar [; rtrue; ],
      despuesCargar [numbalas;
print "Has cargado ",
  numbalas, " bala(s).^"; rtrue;
],
```

Estos métodos se dispararán antes y después de haber cargado el arma.

```
      cargaTodo [bala numbalas x;
            numbalas = 0;

            if (~~self.antesCargar()) rfalse;

            do {
              bala = nothing;
              objectloop (x in jugador) {
                  if (x ofclass BalaArmaDeFuego)
                    if (x.compatible(self)) {
                      bala = x;
                      break;
                    }

              }

              if (bala ~= nothing) {
                  if (self.introduceBala(bala))
                          numbalas++;
                  else bala = nothing;
              }
            } until (bala == nothing);


            return self.despuesCargar(numbalas);
      ],
```

El método cargaTodo(), es muy interesante. Recorre el jugador para buscar todas las balas en su posesión, compatibles con el arma, y las introduce en ella. Para eso, basta un objectloop(objetos en jugador) para localizar aquellas balas compatibles. Entonces, una vez con las balas localizadas, la cargamos en el arma, simplemente moviéndola dentro, como veremos a continuación. Si no encontramos ninguna bala, no se carga ninguna, por supuesto.

```
      ! Los siguientes métodos pueden ser utilizables por el usuario
      ! Automatiza el proceso de incorporación de una bala al arma
      ! Y viceversa
      ! La librería los usa.
      introduceBala [bala;
        if (self.devMaxBalas() > self.devNumBalas()) {
                move bala to self;
                rtrue;
        }
        else rfalse;
      ],
      quitaBala [bala receptor;
        if (self.devNumBalas()>0) {
                if (bala == nothing)
                    bala = child(self);
                move bala to receptor;
                rtrue;
        }
```

```
        else rfalse;
      ],
```

Estos métodos automatizan las tareas necesarias para meter/sacar una bala en/de el arma. Así, comprobamos que no nos estemos pasando de la capacidad máxima del arma, y simplemente movemos la bala dentro de ella. "quitabala()", en cambio, es llamada cuando una bala es disparada, y por tanto, hay que deshacerse de ella. También Podrían utilizarse para cargar balas en un arma manejada por un PSI.

```
      cargaUnaBala [bala x numbalas;
            numbalas = 0;

            if (~~self.antesCargar()) rfalse;

                bala = nothing;
                objectloop(x in jugador) {
                  if (x ofclass BalaArmaDeFuego)
                    if (x.compatible(self)) {
                      bala = x;
                      break;
                    }

                }

                if (bala ~= nothing) {
                  if (self.introduceBala(bala))
                          numbalas++;
                }

                return self.despuesCargar(numbalas);
      ],
```

"cargaUnaBala()" hace exactamente lo mismo que cargaTodo(), pero con tan solo una bala.

```
      descargaTodo [bala numbalas receptor x;
            numbalas = 0;

            if (~~self.antesDescargar()) rfalse;

            if (self in jugador)
                    receptor = jugador;
            else    receptor = localizacion;

            do {
              bala = nothing;
              objectloop(x in self) {
                  if (x ofclass BalaArmaDeFuego) {
                      bala = x;
                      break;
                  }
              }

              if (bala ~= nothing) {
                  if (self.quitaBala(bala, receptor))
                          numbalas++;
                  else bala = nothing;
              }
            } until (bala == nothing);
```

```
            return self.despuesDescargar(numbalas);
      ],

      antesDesCargar [; rtrue; ],
      despuesDescargar [numbalas; print "Has descargado ", numbalas, "
bala(s).^"; rtrue; ],
```

"descargaTodo()" hace exactamente lo contrario que "cargaTodo()", vaciar el cargador del arma en el jugador (o quién sea), pasando éste a tener sus balas.

```
      comprobar [ numbalas;
            if (self.conseguro)
                  print "El seguro del arma está echado, ";
            else print "El arma tiene el seguro abierto, ";

            if (self.apuntado ~= nothing)
                  print "tienes a ", (el) self.apuntado, " en el punto de
                        mira, ";
            else if (self in jugador)
                   print "apuntando al suelo, ";

            numbalas = self.devNumBalas();
            if (numbalas > 0)
                  print "y cuentas con ", numbalas, " bala(s).^";
            else print "y está descargada.^";

            print "Es de calibre ", self.devCalibre();
            print " y admite un máximo de ", self.devMaxBalas(), " bala(s).^";
      ]
```

"comprobar()" es un método de información que permite al usuario enterarse de en qué condiciones está su arma.

Gramática

A continuación veremos un resumen de las gramáticas que es necesario definir para poder trabajar con la librería.

```
Extend 'quita' last
        * 'seguro' 'al' noun  -> quitarSeguro
        * 'seguro' 'en' noun  -> quitarSeguro
        * 'seguro' 'del' noun -> quitarseguro
        * 'seguro' 'de' noun  -> quitarSeguro
;

Extend 'pon' last
        * 'seguro' 'al' noun  -> ponerSeguro
        * 'seguro' 'en' noun  -> ponerSeguro
        * 'seguro' 'del' noun -> ponerseguro
        * 'seguro' 'de' noun  -> ponerSeguro
;
```

Extendemos los verbos 'quita' y 'pon' para poder manipular los seguros de las armas.

```
Verb 'descarga' = 'vacia';
```

Definimos un nuevo verbo, 'descarga', pero no nos molestamos en definir una nueva acción, simplemente lo hacemos igual a 'vacía'. Cuando un arma reciba la acción 'vaciar', será momento de descargar todas sus balas.

```
Verb 'carga'
        * noun -> cargar
;
```

No queda más remedio que definir un nuevo verbo para cargar las balas. No existe otra acción que se parezca y tenga sentido, así que es mejor crearla. Cuando un arma reciba la acción cargar, será momento de obtener todas las balas (compatibles en calibre) en ella.

```
Verb 'apunta'
        * noun 'a//' noun -> apuntar
        * 'con' noun 'a//' noun-> apuntar
        * noun 'al' noun -> apuntar
        * 'con' noun 'al' noun -> apuntar
;
```

Debemos poder apuntar un arma contra un objetivo, es decir otro objeto. Cuando esta acción se realice, uno será el arma y otro será el objeto al que se va a apuntar.

```
Verb 'dispara'
        * noun -> disparar
        * 'con' noun -> disparar
;
```

Disparar es una acción sencilla, simplemente necesitamos saber qué arma disparar.

Una aventurilla de ejemplo

Para poder probar la nueva librería, hay una pequeña aventurilla de ejemplo [1] que desgranaremos a continuación. Se trata de nuestro jugador metido en una sala de tiro, donde podrá comprobar diversas armas. En cuanto salga a la calle, el juego se acaba.

```
Constant Historia "Un día duro en la sala de tiro.^^";

Include "EParser";
Include "Acciones";
Include "Responde";                     ! Módulo de respuestas múltiples
Include "combate";                       ! Módulo de combates
Include "Gramatica";
Include "GramaticaCombate";     ! Gramática para el módulo de combates


! ================================= Sitios
responde moverseLocal
private
      elementos
            "Abres la puerta y la cierras a tu paso."
            "Pasas por la puerta ..."
            "Franqueas la puerta."
            "Le cedes el paso a alguien, y franqueas la puerta."
;
```

Utilizaremos la librería de mensajes de respuesta para aderezar los cambios de localidad dentro del local.

```
class lugar
;
```

Una localidad cualquiera. Como en el caso de las armas, lugar no aporta nada ni hace nada. Simplemente, nos permitirá identificar fácilmente en el futuro una localidad.

```
class habLocal
class lugar
with
      antes [;
            Ir: print (string) moverseLocal.dev_Msg(), "^";
                  rfalse;
      ]
has luz
;
```

Estamos dentro de un edificio, y en los edificios suele haber puertas separando las habitaciones. Así que cuando se genere la acción "ir", aprovecharemos para imprimir uno de nuestros mensajes, haciendo ver que ha cruzado una puerta. Por supuesto, podemos colocar puertas de verdad entre las habitaciones, pero esto es una forma de hacerlo tan correcta como otra cualquiera. Obsérvese que devolvemos falso para que la acción "ir" discurra por su curso normal.

```
class calle
class lugar
has luz
;

class mueble
has estatico soporte
;
```

Dos clases que nos servirán para la localidad final y para los objetos que va a haber en la sala de tiro, respectivamente.

```
! ================================= Localidades

habLocal salaDeTiro "Sala de tiro"
with
      al_e pasillo,
      descripcion "La sala de tiro es extraordinariamente larga, aunque quizás
                  un poco estrecha. Hay 5 plazas para tirar, con las dianas
                  a unos 100 metros al norte del lugar de tiro. Al este, se
                  sitúa la puerta de salida."
;

habLocal pasillo "Pasillo"
with
      al_o salaDeTiro,
      al_e Lepanto,
      descripcion  "Un largo y estrecho pasillo lleva al exterior, a la calle
Lepanto."
;

calle lepanto
with
      descripcion [;
            banderafin = 2;
            "Por fín has conseguido salir de ese infierno ...^
             La calle, rebosante de vida, se abre de este a oeste, en una
alegoría
             de tu recién obtenida vida.";
      ]
```

```
;
```

Ya tenemos definidas las tres localidades: la sala de tiro, el pasillo, y la calle lepanto, donde nada más entrar, se acaba la aventura.

```
! ================================== Objetos

mueble mostrador "mostrador" salaDeTiro
with
      nombre 'mostrador' 'mesa',
      descripcion "Pues sí, es una vieja mesa de madera que cubre toda la parte
trasera del local.",
      antes [;
            atacar: if (armaDisparada() ~= nothing)
                    "Le haces un agujero nuevo al mostrador.";
                    else "El mostrador ya está suficientemente viejo y
estropeado.";
      ]
;
```

Aquí comenzamos a apreciar la interacción entre las armas y el resto del mundillo de nuestra aventura. Así, podemos patear el mueble. En tal caso, no se habrá disparado ningún arma, y la función "armaDisparada()" devolverá "nothing" (que es lo mismo que 0 o NULL en C y Java o nil en Pascal). Éste es el curso de acción habitual que sucede cuando tecleamos "golpear mostrador", por ejemplo. Si se ha disparado un arma, entonces está claro que le hemos hecho un agujero nuevo al mostrador, pero no lo tenemos más en cuenta.
"armaDisparada()" sólo hace un objetloop(objetos ofclass armaDeFuego), para encontrar el arma que fue disparada en este mismo turno. Si no encuentra una, entonces retorna "nothing", en otro caso devuelve una referencia al objeto "armaDeFuego" disparado.

```
object diana "diana" salaDeTiro
private
    num_tiros 0
with
        nombre 'diana',
        descripcion [;
                print "La diana de tu calle está preparada para ser
utilizada.^";
                if (self.num_tiros == 0)
                     print "No has disparado a la diana.";
                else print "Has acertado ", self.num_tiros, " disparo(s).";
        ],
        antes [;
            atacar:
                if (armaDisparada() ~= nothing) {
                  if (random(10)<= 8) {
                        self.num_tiros++;
                        "Aciertas a la diana.";
                      }
                       else "Yerras el tiro.";
                } else "Está demasiado lejos.";
        ]
has femenino;
```

Tenemos un objeto mucho más receptivo a nuestros esfuerzo bélicos. Hemos puesto un contador para saber el número de balas que recibe la diana (es decir, los disparos), y los visualizamos a la hora de imprimir la descripción de la diana. La clave, por supuesto, está en el método "antes()". Cuando la diana recibe la acción "atacar", comprueba si el ataque ha venido de un arma de fuego. En tal

caso, contabilizamos una bala más en la diana (aunque puede que el disparo se pierda por ahí, gracias a un pequeño "random()"). Si no, es que hemos puesto algo así como "golpear diana". Y eso no nos vale, claro.

```
armaDeFuego pistolaWPK38 "Walther PK 38" mostrador
private
        poder_ofensivo 20,
        calibre 38,
        maxnumbalas 10
with
     nombre 'pistola' 'walther' 'walter' 'pk',
     descripcion [;
        self.comprobar();
        "^El esmalte negro de la walther reluce peligrosamente.^";
     ]
has femenino
;
```

Creamos el objeto pistola. Redefinimos los valores "poder_ofensivo", "calibre" y "maxnumbalas", para darle los valores que queremos. En la descripción llamamos al método "comprobar()", ya que da automáticamente un montón de información sobre el arma. El resto ya lo hace la librería.

```
BalaArmaDeFuego bala38 "bala del 38" mostrador
private
        calibre 38
with
        nombre 'bala' '38',
        descripcion "Pues sí, una bala del 38"
has femenino;

BalaArmaDeFuego bala45 "bala del 45" mostrador
private
        calibre 45
with
        nombre 'bala' '45',
        descripcion "Pues sí, una bala del 45"
has femenino;
```

Tenemos dos balas, una del 38 y otra del 45. Una es para la pistola, y la otra para el cetme, que veremos a continuación.

```
armaDeFuego Cetme "cetme" mostrador
private
        poder_ofensivo 30,
        calibre 45,
        maxnumbalas 50
with
     nombre 'fusil' 'asalto' 'cetme',
     descripcion [;
        self.comprobar();
        "^Es un fusil de asalto de fabricación española.^";
     ]
;
```

Ahora tenemos el cetme, con sus propiedades. Vemos que no es demasiado diferente a la pistola.

```
arma_mixta espadaToledana "espada" mostrador
private
        poder_ofensivo 15,
```

```
        poder_defensivo 15
with
        nombre 'espada' 'sable' 'toledo' 'toledana' 'espanola',
        with
                descripcion "Es una espada de fabricación Toledana."
;
```

Aquí tenemos una espada, que en momentos próximos de la serie veremos cómo puede ser utilizada. Es el ejemplo perfecto de cómo un arma puede ser de ataque y defensa a la vez. Es poco probable que siga siendo "arma_mixta", probablemente necesitaremos crear una clase más elaborada, que derive de ella. Por supuesto, de poco sirve frente a un arma de fuego (quitando a los ninjas que desvían las balas con sus katanas), pero eso también podremos comprobarlo más adelante en nuestro juego.

```
! ================================== Inicializar
[Inicializar;
      localizacion = salaDeTiro;

      "Un largo y frustrante día de trabajo ...
       esperas poder relajarte en la sala de tiro, afinando tu puntería ...^";
];
```

Finalmente, inicializamos de la forma habitual el juego, mediante la rutina de la librería estándar, "inicializar()".

Conclusiones

En esta primera entrega, nos hemos concentrado en una tipo particular de armas, las armas de fuego. En las siguientes entregas, veremos como modelar el resto de armas posibles, y, sobre todo, los guerreros y sus estadísticas de combate. Veremos también como cambiar al jugador estándar de inform por un guerrero, cosa necesaria para poder hacer que el jugador intervenga en los combates (o no ...).

Referencias

[1] http://usuarios.lycos.es/elarquero/
[2] baltasarq@yahoo.es
[3] http//caad.mine.un/