Índice

Ejercicios Unidad 15 - Herencia

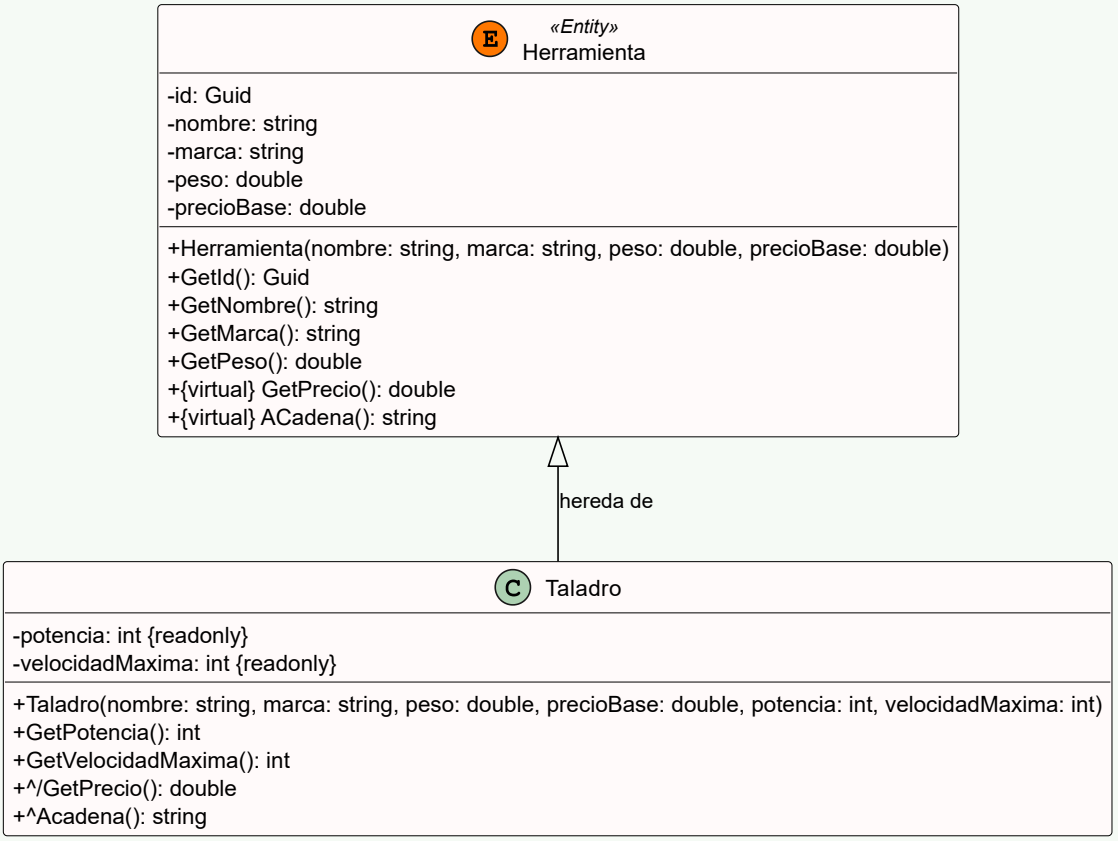
Descargar estos ejercicios

> **Antes de empezar**
>
> Para realizar estos ejercicios, deberás descargar los recursos del enlace de proyecto_poo_roles_herencia. Como puedes ver, la solución está compuesta de varios proyectos. Cada uno de ellos corresponde con un ejercicio, deberás implementar todo el código, tanto de la Main como de los métodos que se piden en cada ejercicio. Cada proyecto contiene el test correspondiente, que deberás pasar para comprobar que has hecho el ejercicio correctamente.

Ejercicio 1. Herencia con herramientas de carpintería

Implementa un sistema de herramientas de carpintería donde tengas una clase base `Herramienta` y una clase derivada `Taladro`. La clase hija debe sobrescribir tanto una propiedad como un método de la clase padre. Para este y el resto de ejercicios, si es posible usar las propiedades autoimplementadas, serán las que elegiremos para que el código quede más compacto.

```
Ejercicio 1: Sistema de herramientas de carpintería

Creando herramientas ....

Martillo, Marca: Stanley, Peso: 0.5, Precio: 25

Taladro Percutor, Marca: Bosch, Peso: 2.3, Precio: 89.25, Potencia: 750, Velocidad Máxi

Presiona cualquier tecla para salir...
```



Requisitos:

- **Clase Herramienta**:
  - Propiedad virtual `Precio` (double) que por defecto devuelve `precioBase`
  - Método `ACadena()` que devuelve una representación de la herramienta
- **Clase Taladro**:
  - Métodos getter para las propiedades específicas
  - Propiedad `Precio` calculada que aplica un descuento del 15% sobre el precio base.
  - `Acadena` debe incluir información técnica: "Taladro percutor profesional - Potencia: {potencia}W, Velocidad: {velocidadMaxima} RPM"
- **Programa principal**:
  - Crear un método estático `GestionHerramientas()` que:
    - Cree una herramienta básica (Martillo, Stanley, 0.5kg, 25.00€)
    - Cree un taladro (Taladro Percutor, Bosch, 2.3kg, 105.00€, 750W, 3000 RPM)
    - Muestre la información de ambas herramientas como en la salida.

Ejercicio 2. Herencia con locales y cines

Implementa un sistema de gestión de locales, locales comerciales y cines utilizando herencia. La clase `LocalComercial` heredará de `Local`, y la clase `Cine` heredará de `LocalComercial`. Deberás sobrescribir el método `ACadena` en cada nivel de la jerarquía para mostrar la información completa de cada tipo de entidad.

```
Ejercicio 2: Sistema de gestión de cines

=== Creando y mostrando cines ===

--- Cine 1 ---
Local:
  Ciudad: Alicante
  Calle: Calle de las Setas
  Número de plantas: 1
  Dimensiones: 500 m²
Local Comercial:
  Razón Social: Cinesa
  Número de Licencia: 12345
Cine:
  Aforo de la sala: 200 personas

--- Cine 2 ---
Local:
  Ciudad: Madrid
  Calle: Gran Vía
  Número de plantas: 3
  Dimensiones: 1200 m²
Local Comercial:
  Razón Social: Yelmo Cines
  Número de Licencia: 67890
Cine:
  Aforo de la sala: 350 personas

--- Cine 3 ---
Local:
  Ciudad: Valencia
  Calle: Plaza del Ayuntamiento
  Número de plantas: 2
  Dimensiones: 800 m²
Local Comercial:
  Razón Social: Kinépolis
  Número de Licencia: 54321
Cine:
  Aforo de la sala: 450 personas

Presiona cualquier tecla para salir...
```

Requisitos:

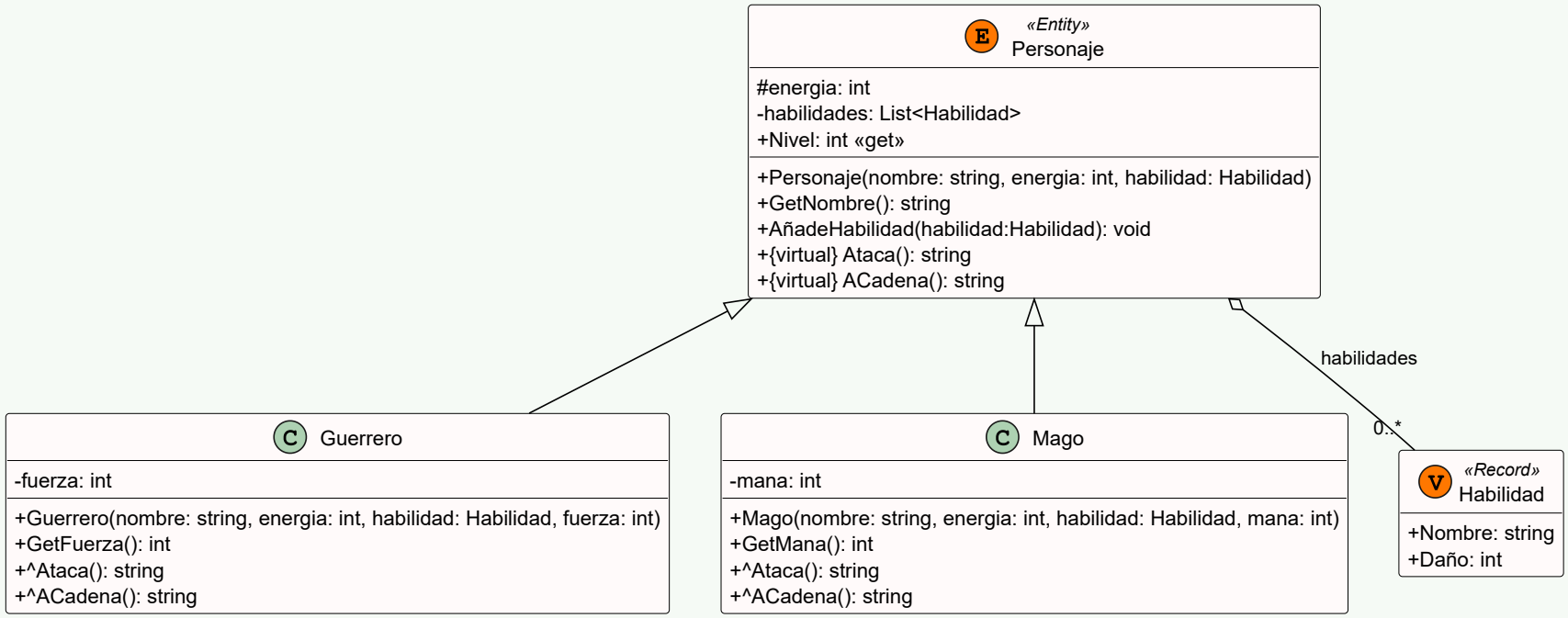
- Value Object **Dimension** con las propiedades `Ancho` y `Largo` (float).
- **Clase Local** (clase base):
  - Propiedades privadas `Ciudad`, `Calle`, `NumeroPlantas` (string), `Dimensiones` (Dimension) con relación de composición.

    - Constructor al que le llegarán los valores de las propiedades necesarios, en el caso de la dimensión llegarán dos valores para crearla.
    - Método virtual `ACadena()` que devolverá un string con los datos de un local. Al mostrar la salida de la dimensión, se multiplicará el ancho por el largo para calcular los metros cuadrados.
- **Clase LocalComercial** (hereda de `Local`):
    - Propiedades adicionales: `RazonSocial`, `NumeroLicencia` (string)
    - Anula `ACadena()` para devolver un string con los datos de un local comercial, aprovechando el código de la clase padre
- **Clase Cine** (hereda de `LocalComercial`):
    - Campo adicional: `AforoSala` (int)
    - Operación `ACadena()` que devolverá un string con los datos de un cine, aprovechando el código de las clases padre
- **Programa principal**:
    - Crear un método estático `GestionCines()` que:
        - Cree una `List<Cine>`
        - Inicialice la lista con datos de 3 cines
        - Muestre por pantalla la información completa de cada uno de los cines mediante un `foreach`, utilizando el método `ACadena`

Ejercicio 3. Herencia con Personajes de un juego

Implementa un sistema de personajes para un juego de rol. Tendrás una clase base `Personaje` y dos clases derivadas, `Guerrero` y `Mago`. Deberás usar herencia, propiedades protegidas, propiedades calculadas y métodos virtuales. Además, la clase base se relacionará con un `record` para definir una habilidad especial.

```
Ejercicio 3: Sistema de Personajes

=== Creando un Guerrero ===
Guerrero creado: Personaje: Conan, Nivel: 10, Energía: 100
  Habilidades:
    * Furia (Daño: 50)
    * Golpe Giratorio (Daño: 30)
  Fuerza: 20

=== Creando un Mago ===
Mago creado: Personaje: Gandalf, Nivel: 12, Energía: 120
  Habilidades:
    * Rayo (Daño: 40)
    * Bola de Fuego (Daño: 70)
    * Escarcha (Daño: 25)
  Maná: 50

Mago con nueva habilidad añadida: Personaje: Gandalf, Nivel: 12, Energía: 120
  Habilidades:
    * Rayo (Daño: 40)
    * Bola de Fuego (Daño: 70)
    * Escarcha (Daño: 25)
    * Telequinesis (Daño: 10)
  Maná: 50

=== Acciones de los personajes ===
--- Guerrero ---
Conan ataca con Furia! (fuerza 20 + mitad habilidades = 60)

--- Mago ---
Gandalf ataca con Bola de Fuego! lanza Rayo con fuerza 40 y se apoya con: (maná 50 + to

Presiona cualquier tecla para salir...
```


«Entity»
Personaje
#energia: int
-habilidades: List<Habilidad>
+Nivel: int «get»
+Personaje(nombre: string, energia: int, habilidad: Habilidad)
+GetNombre(): string
+AñadeHabilidad(habilidad:Habilidad): void
+{virtual} Ataca(): string
+{virtual} ACadena(): string
habilidades
0..*
Guerrero
-fuerza: int
+Guerrero(nombre: string, energia: int, habilidad: Habilidad, fuerza: int)
+GetFuerza(): int
+^Ataca(): string
+^ACadena(): string
Mago
-mana: int
+Mago(nombre: string, energia: int, habilidad: Habilidad, mana: int)
+GetMana(): int
+^Ataca(): string
+^ACadena(): string
«Record»
Habilidad
+Nombre: string
+Daño: int

Requisitos:

- **Record Habilidad**: Propiedades de solo lectura `Nombre` y `Daño`.
- **Clase Personaje**:
  - El constructor recibe: `nombre`, `energia`, `List<Habilidad> habilidades`. Valida que haya al menos una; en caso contrario se creará una habilidad por defecto con daño `0` y nombre `No Hábil`.
  - Propiedad calculada `Nivel = energia / 10`.
  - Método virtual `Ataca()` que usará la habilidad principal para el cálculo del ataque. La habilidad principal será la que mayor daño tenga.
  - `ACadena()` debe usar un `StringBuilder` para construir la salida multi-línea de las habilidades, como se muestra en la salida.
- **Clase Guerrero**:
  - Añade propiedad `Fuerza` como entero.
  - `Ataca()` a partir del ataque base del personaje, se le añade la `fuerza + (el total de las habilidades / 2)`.
- **Clase Mago**:
  - Añade propiedad `Mana` como entero.
  - `Ataca()` a partir del ataque base del personaje, lanza la primera habilidad con su fuerza y se apoya con : `mana + habilidadPrincipal.Daño`.
- **Programa principal (`GestionPersonajes`)**:
  - Crear listas de habilidades para cada personaje (mínimo 2 para ejemplificar):
    - Guerrero: Furia (50), Golpe Giratorio (30).
    - Mago: Bola de Fuego (70), Rayo (40), Escarcha (25).
  - Instanciar personajes y mostrar su representación (`ACadena()`).
  - Añade una habilidad más al personaje Mago.
  - Mostrar ataques indicando el uso de la habilidad principal (la primera en la lista).

Ejercicio 4. Polimorfismo de datos con herramientas avanzadas

Extiende el sistema de herramientas del Ejercicio 1 añadiendo nuevas subclases y un nuevo método virtual común que permita **demostrar el polimorfismo de datos**.

```
Ejercicio 4: Polimorfismo con herramientas

Creando inventario...

Martillo, Marca: Stanley, Peso: 0,5, Precio: 25
Taladro Percutor, Marca: Bosch, Peso: 2,3, Precio: 89,25, Potencia: 750, Velocidad Máxi
Sierra Circular, Marca: Makita, Peso: 4,1, Precio: 132, Potencia: 1400, Hoja: 185mm
Lijadora Orbital, Marca: Dewalt, Peso: 1,8, Precio: 54, RPM: 12000, Disco: 125mm

Mostrando usos...
Martillo: Golpea superficies para ensamblar o fijar piezas.
Taladro Percutor: Perfora materiales duros a alta velocidad.
Sierra Circular: Realiza cortes rectos en madera y tableros.
Lijadora Orbital: Lija y suaviza superficies de madera.

Accediendo a métodos específicos:

Taladro Percutor => Perforar 8mm diámetro, 60mm profundidad OK
Sierra Circular => Cortar("Madera", 18mm) OK
Lijadora Orbital => Pulir(2.5 m2) tarda 2fs

Presiona cualquier tecla para salir...
```

Requisitos:

- Debes reutilizar el código del ejercicio 1 y añadir en la clase padre un **método virtual nuevo** `Usa` que devuelva una descripción genérica del uso según la salida anterior en las clases correspondientes.
- Nuevas subclases a añadir:
  i. **SierraElectrica** con las propiedades específicas `potenciaW` (int), `diametroHojaMm` (int) y un método propio `Corta(material: string, grosorMm: int): string`. En el Precio, aplica un recargo ambiental del 10% sobre el `precioBase` si la potencia es > 1300W; si no, deja el precio base.
  ii. **Lijadora** con las propiedades específicas `rpm` (int), `diametroDiscoMm` (int) y un método propio `Pule(superficieM2: double): double` que devuelve los segundos estimados

(tiempoSeg = (superficieM2 * 60) / (rpm * 0.012)). En cuanto al Precio, aplicará un descuento ecológico del 10% si `rpm < 10000`, si no mantiene el precio base.

iii. A la clase **Taladro** se le añadirá un método propio `Perfora(diametroMm: int, profundidadMm: int): string`.

- **Programa principal**: (`GestionHerramientasPolimorfismo`)
  - Crear una lista con las instancias de las siguientes herramientas:
    - Martillo (sin cambios respecto al Ej.1)
    - Taladro Percutor (como Ej.1)
    - Sierra Circular (Makita, peso 4.1kg, base 120€, potencia 1400W, hoja 185mm) => precio final 132€ (recargo 10%)
    - Lijadora Orbital (Dewalt, peso 1.8kg, base 54€, rpm 9000, disco 125mm) => precio final 48.60€ (con descuento, se muestra con dos decimales)
  - Recorre la lista con un foreach mostrando la información de las herramientas conforme se ve en la salida (extraída del ACadena).
  - Muestra los usos de todas las herramientas, mediante un foreach
  - Intenta usar los métodos específicos de cada herramientas. Explica (en comentarios junto al uso) que métodos como `Pulir`, `Cortar`, `Perforar` NO se pueden llamar desde la referencia `Herramienta`.
  - Usa (`switch` o `is`) para invocar los métodos específicos y mostrar resultados reales.

Ejercicio 5. Red de albergues de viaje

Implementa un sistema simplificado de gestión de una pequeña red de albergues de viaje. Se desea almacenar distintos tipos de albergues y calcular de forma polimórfica si admiten una reserva y el precio final por noche según su tipo, temporada y ocupación.

```
Ejercicio 5: Red de albergues (versión interactiva)

--- MENÚ GESTIÓN ALBERGUES ---
[1] Añadir albergue
[2] Registrar reserva
[3] Mostrar estado
[4] Mostrar información complementaria
[ESC] Salir
Seleccione opción: 3

=== ESTADO ACTUAL ===
Montaña Verde -> Ocupación: 3/40 (7%) Precio base: 22.00€ Precio actual: 25.30€
City Hostel   -> Ocupación: 5/80 (6%) Precio base: 18.00€ Precio actual: 18.00€
Surf Point    -> Ocupación: 10/55 (18%) Precio base: 20.00€ Precio actual: 23.00€

Pulse una tecla para continuar...
--- MENÚ GESTIÓN ALBERGUES ---
{se omite el menú por simplificación}
Seleccione opción: 2
Listado de albergues:
 (0) Montaña Verde
 (1) City Hostel
 (2) Surf Point
Elija índice: 0
Plazas a reservar: 8
¿Temporada alta? (S/N): S
[Reserva] Montaña Verde: 8 plazas (ALTA) => ACEPTADA. Ocupación ahora 11/40 (27%)

--- MENÚ GESTIÓN ALBERGUES ---
{se omite el menú por simplificación}
Seleccione opción: 4
Información complementaria:
Montaña Verde => Clima previsto: Nieve ligera
City Hostel => Sin información complementaria
Surf Point => Oleaje estimado: Moderado

--- MENÚ GESTIÓN ALBERGUES ---
{se omite el menú por simplificación}
Seleccione opción: ESC
Saliendo del gestor...
```

Descripción general:
Se parte de una clase base `Albergue` y tres subclases: `AlbergueRural`, `AlbergueUrbano` y `AlbergueCostero`. Todas comparten la capacidad, reservas y cálculo del porcentaje de ocupación; cada una aplica reglas distintas para aceptar reservas y ajustar el precio final por noche.

Entidades de tipo Value Object:

- `Direccion` con las propiedades Ciudad (string) y Pais (string).
- `Reserva` con las propiedades Plazas (int) y EsTemporadaAlta (bool)

Clase base Albergue:

- Que tendrá las siguientes propiedades privadas si no se especifica lo contrario:
  - `Nombre` (string, readonly)
  - `Capacidad` (int, readonly, > 0)
  - `Direccion` (Direccion, composición y readonly)
  - `Servicios` (List)
  - `Reservas` (List, protegida y de tipo agregación)
  - `PrecioBase` (double y protegida)
  - `Precio` propiedad pública calculada (double, readonly y redefinible)
  - `PlazasOcupadas` propiedad calculada (int, readonly y protegida) que devolverá el número total de reservas.
  - `PorcentajeOcupacion` propiedad calculada (int, readonly y pública) que devolverá el porcentaje de ocupación.
- Métodos públicos si no se especifica lo contrario:
  - `AgregaServicio` que agrega un servicio que le llega como entrada al albergue, evitando duplicados.
  - `AñadeReservaInterna` método privado al que le llega una reserva y actualiza la ocupación.
  - `AdmiteReserva` método protegido y redefinible al que le llega el número de plazas y un boolean indicando si es temporada alta, devolverá un boolean admitiendo o no la reserva dependiendo de si todavía quedan plazas en el albergue (teniendo en cuenta las que se quieren reservar), o si la reserva es válida *plazas > 0*:
  - `RegistraReserva` método público al que le llega el número de plazas y un boolean indicando si es temporada Alta, con esta información llamará a `AdmiteReserva` y si la reserva es posible la añadirá y devolverá un boolean confirmando que ha sido registrada o false en caso contrario.

- `CalculaPrecioActual` método público y redefinible que devolverá el precio como doble. Por defecto devolverá el precio base.
- `InformacionComplementaria` método público y redefinible que devolverá una cadena con datos de contexto externo (clima, oleaje, eventos urbanos, etc.). Devolverá como base "Sin información complementaria".
- `Anula el ToString` para que conseguir una salida como la que se muestra con: nombre, tipo, capacidad, porcentaje y servicios.

Reglas específicas:

1. `AlbergueRural` :
   - Precio: si temporada alta en alguna reserva (existe alguna `EsTemporadaAlta`) y ocupación > 20% => `PrecioBase * 2.15` si no `PrecioBase * 1.15` ya aplicado por incluir desayuno y cena.
   - AdmiteReserva: además de la base, rechaza peticiones > 50% de la capacidad total en una sola reserva.
   - InformacionComplementaria devuelve un texto de clima simulado.
2. `AlbergueUrbano` :
   - Precio: si ocupación >= 60% => `PrecioBase * 1.10` si no `PrecioBase` .
   - AdmiteReserva: permite reservas grandes mientras queden plazas (no añade restricciones extra).
3. `AlbergueCostero` :
   - Precio: si es temporada alta en la última reserva registrada => `PrecioBase + 3` y si ocupación > 40% añade +2 extra.
   - AdmiteReserva: rechaza si temporada ALTA y plazas solicitadas > 60% de la capacidad restante.
   - InformacionComplementaria devuelve un texto sobre el oleaje.

Programa principal:

- En el programa principal inicializa una lista con los 3 albergues de ejemplo y sus servicios, y registra las reservas iniciales.
- Crea un método `GestionAlbergues` que muestra un MENÚ hasta que el usuario pulse la tecla ESC que permita:
  i. "Añadir albergue", que llamará a un método `AñadeAlbergue` que pedirá los datos y añadirá un albergue a la lista.
  - Pedir tipo (R/U/C).
  - Pedir nombre, capacidad (>0), precio base (>0), ciudad y país.
  - Permitir introducir servicios separados por coma.

ii. "Registrar reserva", que llamará a un método `ReguistraReserva` que:
   - Listar albergues con índice.
   - Pedir índice válido, plazas (int >0) y si es temporada alta (S/N).
   - Llamar a `RegistraReserva` e indicar ACEPTADA o RECHAZADA.

iii. "Mostrar estado", muestra directamente el estado como en salida.

iv. "Mostrar información complementaria", muestra la información complementaria.

v. "ESC",Finaliza el bucle.

Ejercicio 6. Sistema de dibujo con herramientas

Crea un proyecto con **los TAD necesarios** para que el siguiente código perteneciente a la Main, pueda ser ejecutado sin problemas:

```
Console.WriteLine("Ejercicio 3: Sistema de dibujo con herramientas");
Console.WriteLine();
Compas compas = new Compas();
Circulo circulo = compas.DibujaCirculo(3.5f);
Rotulador rotulador = Estuche.GetRotuladores()
                      [
                          new Random().Next(0, Estuche.NUMERO_ROTULADORES)
                      ];
rotulador.Rotula(circulo.Perimetro());
Pincel pincel = new Pincel();
pincel.Color= Color.Verde;
pincel.Pinta(circulo.Area());
Console.WriteLine("\n¡Dibujo completado con éxito!");
Console.WriteLine("Presiona cualquier tecla para salir...");
Console.ReadKey();
```

```
Ejercicio 6: Sistema de dibujo con herramientas

Dibujado un círculo de radio 3,5 cm
Rotulado el perímetro de 21,99 cm de color Negro.
Pintada el área de 38,48 cm² de color Verde.

¡Dibujo completado con éxito!
Presiona cualquier tecla para salir...
```

Requisitos:

- El circulo tendrá un atributo radio.
- El rotulador tendrá un atributo color de tipo enumerado y solo rotula perímetros.
- Habrá una clase estática Estuche con un solo método también estático que devolverá un array de rotuladores con colores creados de forma aleatoria.
- El pincel también tiene un atributo color y solo pinta áreas.