Melodias/órdenes:

- De catálogo
- Personalizada

Ordenes:

- precio
- ¡validación!

```
// 1. Como cliente, poder registrarme cargando los datos básicos de
contacto (nombre, apellido, email) y dirección de recepción de las
órdenes.

class Cliente
      ...datos dichos arriba...

// ejemplo de invocación:
new Cliente(....)
```

```
// 2. Como cliente, poder consultar el catálogo de melodías y ordenar
una o más cajitas musicales para la dirección actual del cliente.


class Catalogo / RepositorioMelodia
      // es un singleton

      List<Melodia> getMelodias() // retorna la melodias del catalogo

RepositorioMelodia.getMelodias() NO
RepositorioMelodia.INSTANCIA.getMelodias()
RepositorioMelodia.getInstance().getMelodias()

// opción 1
class Cliente
      list<orden> ordenes

      agregarOrden(Orden unaOrden)
          ordenes.add(unaOrden)


// opción 2
class Orden
      Cliente cliente

      constructor(....)
```

```
// opción 3 NO

class Cliente
      // NO
// es un man in the middle / objeto sistema / confusion entre rol y
objeto / missplaced method
      agregarOrden(orden)
            RepositorioOrdenes.INSTANCIA.agregarOrden(orden)
```

// 3. Como cliente, poder consultar el estado de una orden que realicé.

¿Qué estados pueden surgir?

- PENDIENTE (de validación) <- esto quedó en evidencia en el punto 8
- PEDIDA
- EN PROCESO DE FABRICACION / ASIGNADA A FABRICA
- ~~FABRICADA~~

```
// opción 1
class Orden
      Estado getEstado()

enum Estado { PEDIDA, ASIGNADA }

// opción 2 NO
class Orden
      Estado verEstado() / consultarEstado()

// opción 3 NO
class Cliente
      Estado obtenerEstado(Orden orden)
            if not orden.cliente == this throw Exception
            orden.getEstado()


// Ejemplo de uso
orden = cliente.getOrdenes()
orden.obtenerUnaAlAzar().getEstado()

// otro ejemplo posible (mejor, porque es más sencillo)
orden.getEstado()
```

```
// 4. Como cliente, poder consultar el tiempo estimado que queda hasta
que la orden llegue a mi dirección.

// Si o sí, hagan que dependa de alguna forma del estado

// Opción 1: pensar a los objetos estado como una familia polimórfica
que sepa darte el tiempo restante
// Opción 2: pensar a los objetos estado como una familia polimórfica (o
no) que sepa darte la fecha de finalización y desde la orden, hacemos la
resta

 // este punto es mas o menos algorítmico, pero lo importante no es el
algoritmo en
// sí sino cuales son sus dependencias

// Opción NO:
class OrdenPersonalizada
      getTiempoRestante() 5
class OrdenCatalogo
      getTiempoRestante() 15
```

```
// 5. Como cliente, poder consultar la dirección a la que llegó o
llegará una orden que realicé, independientemente de cuál sea mi
dirección actual

{Desnormalizar la dirección}
```

```
// 6. Como persona administradora, poder dar de alta fábricas de cajas
musicales en el sistema, y actualizarlas cuando sea necesario.

La clave de este punto es NO hacer objetos sistema

// Solución Gran NO
class Administrador
      void registrarFabrica(fabrica)
            RepoFabricas.INSTANCIA.agregarFabrica(fabrica)

      void actualizarFabrica(fabrica)
            RepoFabricas.INSTANCIA.agregarFabrica(fabrica)

// OJO
// Solución NO
class RepositorioFabricas
      ...es un singleton, tiene un getter de fabricas y un método para
```

```
agregar una instancia...

      void registrarFabrica(nombre, direccion, contacto, etc, etc)
            fabricas.add(new Fabrica(.....)) // OJO esto lleva a un long
paramter list
```

```
// 7. Como persona administradora, poder consultar cuales son las
fábricas que podrían aceptar una orden (es decir, que no supere la
capacidad de producción de la fábrica).

class RepoFabricas
      getFabricasEnCondicionesDeAceptarOrden()
            fabricas.filter(fabrica -> fabrica.puedeAceptarOrden())

// opcion 1
class Fabrica
    boolean puedeAceptarOrden()
      this.capacidadProduccion < this.cantidadDeOrdenesTomadas()

// opcion 2 <- esta opción se cae por el punto 9
class Fabrica
     boolean puedeAceptarOrden()
        this.capacidadProduccion < this.cantidadDeOrdenesTomadas
```

```
// 8. Como persona administradora, poder revisar (típicamente una o dos
veces al día) las órdenes pendientes y asignarles una fábrica a su
elección que esté en condiciones de aceptar la orden

// opcion 1 (puede ser, pero ojo que ya tienen los estados)
class RepositorioOrdenes
      List ordenesPendientes // <- ahora que vimos el punto 9,
   // esta alternativa está aun mas complicada
List ordenesProcesadas // + una gestión de estos estados (que hay
explicar)

      getOrdenesPendientes()
            this.ordenesPendientes

// opcion 2 (más sencillo)
class RepositorioOrdenes
      getOrdenesPendientes()
            this.ordenes.filter(orden -> orden.estaPendiente())

class Orden
      estaPendiente()
```

```
            this.estado == PENDIENTE

// Importante: tratar de gestionar los estados como una única idea
// Notar que la validación, la asignación a fábrica y la fabricación son
todas etapas
// de un mismo proceso
```

```
// 9. Como persona administradora, poder consultar todas las órdenes
asignadas a una fábrica en un determinado período.

class Fabrica
    List ordenesTomadas

    void asignar(orden) // alternativa: orden.serAsignadaA(fabrica)
          // alternativa
          orden.setEstado(ASIGNADA)
          // alternativa
          orden.asignar()
          ordenesTomadas.add(orden)

    int cantidadDeOrdenesTomadas()
      this.ordenesTomadas.size()

    // opcion 1
    List<Orden> ordenesAsignadasEnPeriodo(LocalDate inicio, LocalDate
fin)
// OK
      ...filter que esté en el período ...
            ...OJO con no delegar...
      // NO
           ordenes.filter(orden -> orden.fechaPedido() > inicio and
orden.fechaPedido() < fin)

    // opcion 2
    List<Orden> ordenesAsignadasEnPeriodo(Period periodo)
          ordenes.filter(orden -> orden.fechaPedido().isIn(period))


// IMPORTANTE: Una fecha NUNCA se debe representar como entero o como
string
// ni tampoco debe ser partida en sus componentes
```

```
// 11. Como cliente, poder saber el precio de una orden, el cual se
calcula de forma diferente si es de una orden de catálogo o
personalizada.

{implementar de forma polimórfica el precio entre las subclases o
implementaciones de la orden}
```

```
// 10. Como cliente, poder realizar una orden personalizada, subiendo el
fragmento de audio.

// IMPORTANTE: no tener jerarquías paralelas.
// En otras palabras, si subclasifican las órdenes,
// no subclasifican las melodías. Y lo mismo al revés

class Orden
      ... acá ponen es estado y COMPORTAMIENTO común ...
      // ejemplo:
      getTiempoRestante()
      // probablemente alguna parte de este código sea concreta y venga
acá.

// PARTE 1 (validación)

// Opcion 1
class OrdenPersonalizada

     boolean esValida()
         this.valida

     // Esto lo va a realizar una persona física (humana)
     // El comportamiento se desencadena desde la UI
     // Además, este comportamiento NO se desencadena al realizar el
pedido
     // si no en un momento posterior
     void marcarComoValida() {
        this.valida = true
        this.notificarValidada(....)
     }

     // idem marcarComoInvalida

class OrdenDeCatalogo

      boolean esValida()
           return true
```

```
// Opcion 2 GRAN NO
class OrdenPersonalizada

     boolean esValida()
         this.valida

     void validar() {
        if (condicion extraña) { // Esta línea es incorrecta, dado que
la decisión la toma una persona, no el sistema
            this.valida = true
      this.notificarValidada(...)
        }  else {
            this.valida = false;
            this.notificarInvalida(...)
        }


// PARTE 2 (conversión, que es previa a la validación)

// Solución NO

class OrdenPersonalizada
      constructor(..., conversor, byte[] audio)
      this.midi = conversor.convertir(audio)


// entre el orden = y el retorno van a pasar varios minutos
// en el medio le usuarie no va a ver ningún tipo de progreso
// la conexión HTTP probablemente se corte
// y por si fuera poco, el constructor no termina y "nunca" llega a
entregar un objeto nuevo
orden = new OrdenPersonalizada(....)
return orden


// Solución 1 (poco robusta)
class OrdenPersonalizada
     constructor(...conversor, audio)
          new Thread {   this.midi = conversor.convertir(audio)  }.run()
          // otra:
          Executor.invokeLater(() ->  this.midi =
conversor.convertir(audio))

// Solución 2 (más robusta)
```

```
class OrdenPersonalizada
     constructor(..., audio)
          this.audio = audio

    // y llamar esto desde un main de un proceso "tipo" cron
    void convertir(MIDIConverter conversor) // o una interfaz adaptada
      this.midi = conversor.convert(this.audio)
            this.estado = PENDIENTE_VALIDACION
            this.notificarFinDeConversion(....)

class Main {
     public static void main() {
          conversor = ....
          RepoOrdenes.INSTANCE
.pendientesDeConversion()
.each(orden -> orden.convertir(conversor))
    }



// ejemplo final
// del caso de uso de subida de orden personalizada

// en un primer momento, desde el proceso WEB que desencadena la acción:
orden = new OrdenPersonalizada(...) // opcionalmente agregarla a un
repositorio o al cliente

// luego, desde OTRO proceso cron que correría todo el tiempo (por
ejemplo, una vez por minuto)

// finalmente, cuando el cliente se entera de que la conversión
finalizó, vuelve al sistema WEB y desde dicho proceso:
orden.marcarComoValida()
// o
orden.marcarComoInvalida()
```