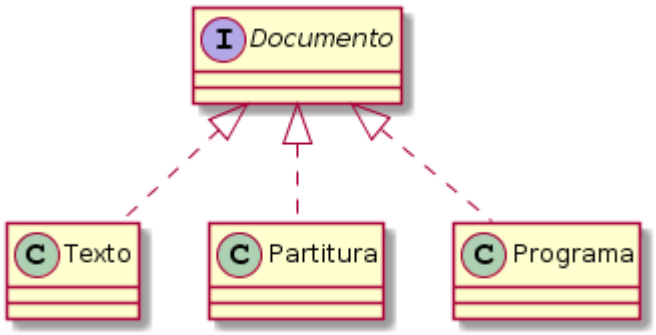
Copia.me: resolución por partes

Copia.me: resolución por partes **1**
- Tipos de documentos 1
- Servicios 2
- Calidades de servicio 3
- Detección automática 4
- Planificación 5
- Reparto de documentos entre los freelancers 5
- Estados 7
- Construcción del servicio 8
- Diagrama de clases completo 9
- Agrupamiento y "densidad" de dudosos 10

Tipos de documentos

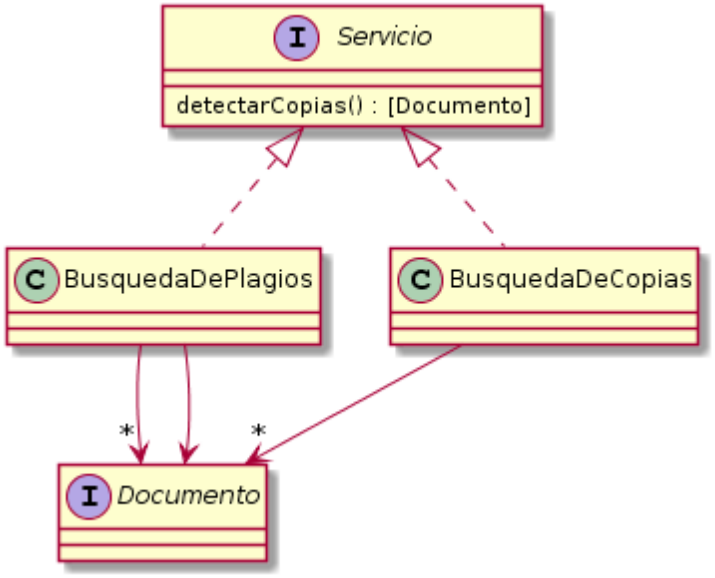
. Por ahora soportaremos textos, programas y partituras pero se espera incorporar más



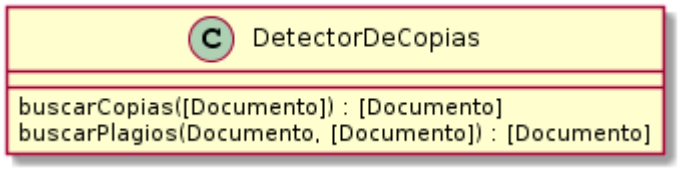
Servicios

El sistema contará con dos grandes funcionalidades (llamadas servicios) (...) Búsqueda de plagios (...) [y] Búsqueda de copias (...):

En este punto podríamos pensar que se tratan de objetos polimórficos....



... o simplemente de dos métodos:



Nota: se podría interpretar que búsqueda de plagios simplemente dice si *alguno* de los documentos es un plagio, y no *cuáles* lo son. Pero si `buscarPlagios` devuelve aquellos que son copias, se puede puede implementar fácilmente un `hayCopias` en términos del éste. Además, `buscarCopias` permitirá fácilmente implementar `buscarPlagios`.

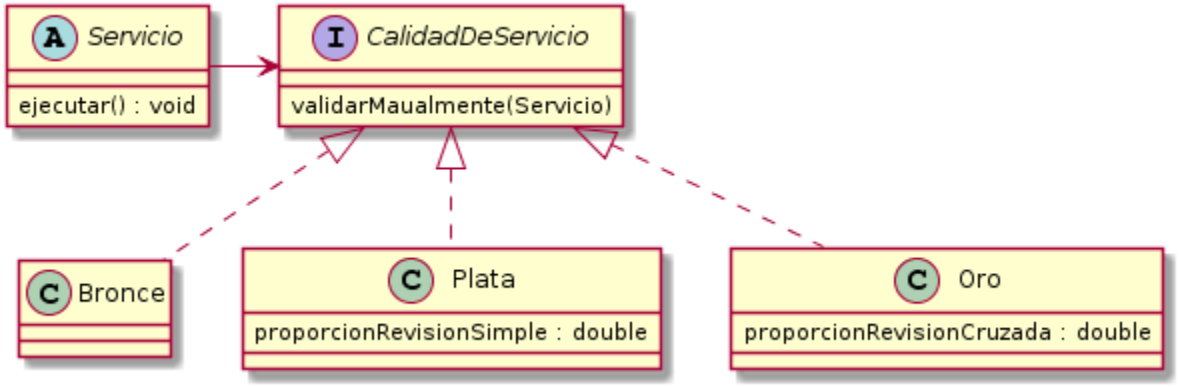
Calidades de servicio

Los usuarios de Copia.me podrán contratar tres niveles de calidad de servicio (de diferente costo): ***Bronce****: sólo realiza detección automática (...),* ***Plata:*** *además de la detección automática, realiza detección manual [simple](...),* ***Oro:*** *además de la detección automática, (...) hace una revisión [manual] cruzada.*

En este punto vemos que hay tres tipos de calidades que nos posibilitan cambiar detalles de implementación de la detección. El hecho de que la detección automática se produce en todos los casos nos da la posibilidad de pensarlo de dos formas:

- La *calidad de servicio* determina cómo evaluar el lote de documentos (y todos las implementaciones existentes realizan la evaluación manual); o bien
- La *calidad de servicio* determina cómo realizar la evaluación automática (y la evaluación manual es realizada por el servicio directamente).

En esta solución iremos por el segundo camino, para evitar la lógica repetida, a expensas de pérdida de flexibilidad:




```
abstract class Servicio
  method ejecutar
    validarAutomaticamente
    calidadDeServicio.validarManualmente(self)
```

Acá valen algunas aclaraciones:

- Dado que el servicio ahora tendrá comportamiento y una calidad de servicio, pasará de interfaz a clase abstracta
- Renombramos y cambiamos la semántica de `detectarCopias` a simplemente `ejecutar`, dado que la la validación manual nos lleva a pensar necesariamente en asincronismo *((...)Cada revisor recibe un email con cada par de documentos asignados(...))*: los revisores serán notificados y en algún momento posterior informarán si los documentos son copias *(Los revisores ingresarán al sistema y cargarán los resultados de sus análisis, siempre en el plazo máximo de 1 día)*. Y dado que el proceso ahora será asincrónico, el método deberá ser void.
- Nos decidimos por una solución en la que el servicio es un objeto, porque:

- Aunque aún no definimos cómo se guardarán los resultados, el problema del asincronismo nos llevará a darle entidad al servicio para poder actualizar posteriormente su estado.
- La validación automática toma tiempo *(los algoritmos automáticos de detección de copias son costosos, así que los ejecutaremos en horas particulares del día)*, así que deberemos cosificar el comportamiento para poder ejecutarlo más tarde.

Detección automática

Vamos a agregar el método polimórfico validar a la jerarquía de documentos. Esta es una Implementación de ejemplo:

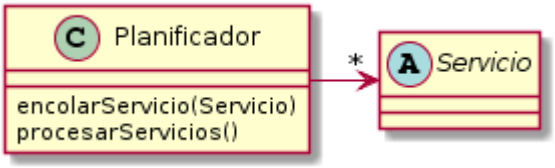
```
class Texto
   method esCopia(otroDocumento)
      return compararSegunDistanciaLeveshtein(otroDocumento.contenido)
```

Por otro lado, para poder implementar la validación desde el servicio, vamos a tener tener que generar todas las posibles comparaciones:

```
class Servicio
  method validarAutomaticamente
     paresDeDocumentosARevisar.each {
        (original, potencialCopia) ->
             // hacer original.esCopia(potencialCopia) y guardar de
             // alguna forma este resultado,
             // ver Documento.validarAutomaticamente
     }

  method paresDeDocumentosARevisar
    // o sea, todos contra todos. Se podría eliminar los pares formados
    // por el mismo documento dos veces, pero es un detalle que no afecta al diseño
     originales.productoCartesianoCon(potencialesCopias)

class BusquedaDePlagios
    // en la búsqueda de plagios sólo hay que
    // comparar el original contra las potenciales copias
    method originales
        [original]

class BusquedaDeCopias
    // acá hay que comparar todos los elementos de la lista de potenciales copias
    // contra sí mismos. Sí, es costoso, lo dijimos en el enunciado :P
    method originales
      potencialesCopias
```

Planificación

Para poder ejecutar asincrónicamente y en un horario particular a los servicios pendientes, necesitaremos un objeto global que nos permita tenerlos a todos, ejecutarlos uno a uno y descartar los que han sido procesados.

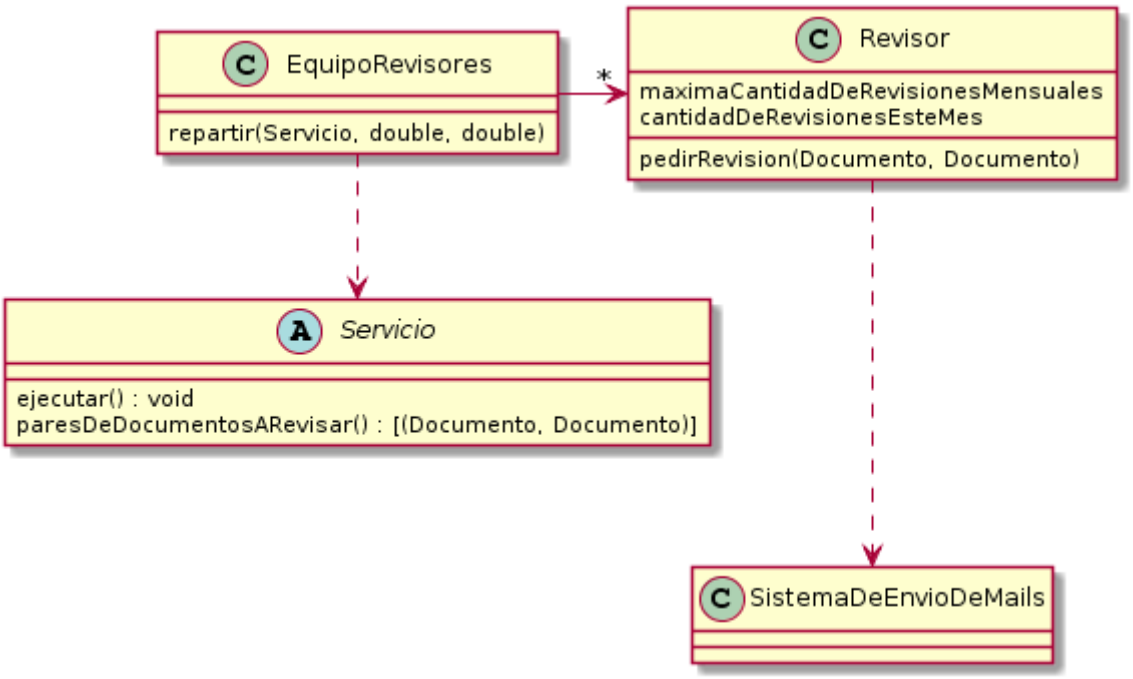



```
class Planificador
   method encolarServicios(servicio)
     servicios.add(servicio)
    //alternativamente, se podría hacer servicio.validarManualmente()

   method procesarServicios
     servicios.each {
        servicio -> servicio.ejecutar()
       // alternativamente, se podría
       // llamar directamente a servicio.validarAutomaticamente()
     }
     // para simplificar la resolución no estamos contemplando accesos concurrentes,
     // ni que se agreguen nuevos servicios mientras se recorre la lista,
     // ni manejando las excepciones que podrían surgir al hacer servicio.ejecutar()
     servicios.clear
```

El método procesarServicios servirá como punto de entrada a un crontab (desde un `main`) o a un timer que se disparará a ciertas horas. En cualquier caso, es importante que:

- el planificador sea un objeto global para poder accederlo fácilmente
- el mensaje procesarServicios no debe tomar parámetros y no debe devolver nada, para que sea fácil de enviar sin necesidad de contexto adicional

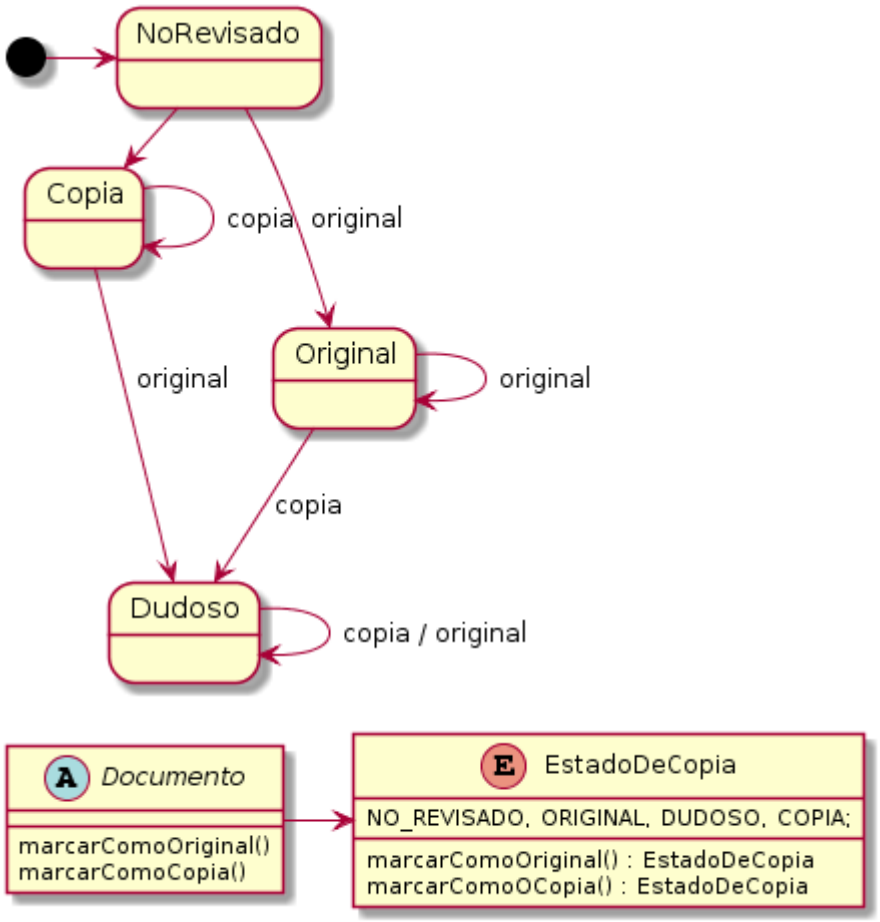
Reparto de documentos entre los freelancers



Nota: en esta parte (que es más algorítmica) no es tan importante el código sino más bien la asignación de las responsabilidades. Pero para la persona, curiosa, contamos cómo se podría implementar:

```
class EquipoRevisores

  method repartir(servicio, proporcionSimple, proporcionCruzada)
    repartirEntre(servicio.paresDeDocumentosARevisar, proporcionSimple)
    repartirEntre(servicio.paresDeDocumentosARevisar, proporcionCruzada)

  method repartirEntre(paresDeDocumentos, proporcion)
    revisores
       .filter { revisor -> revisor.puedeRecibirRevisiones } // si aún no llenó
                                                             // su cupo de revisiones
       .shuffle // si querés mezclarlos para que el reparto sea más equitativo
      .take(paresDeDocumentos.count * proporcion)
      .zip(paresDeDocumentos)
      .each {
         (revisor, parDeDocumentos) -> revisor.pedirRevision(parDeDocumentos)  // esto
enviará el mail
       }
```

Estados



Implementación de ejemplo:

```
enum EstadoDeCopia {
   ORIGINAL {
     //estos mensajes los envía tanto el revisor como el algoritmo de detección de
copias
      method marcarComoOriginal
        self
      method marcarComoCopia
        DUDOSO
   },
  //etc...
```

Nota: el documento ahora es una clase abstracta, para poder tener el estado de copia y los métodos correspondientes para cambiarla. Además, ahora hay que complementar el método esCopia con otro que no devuelva un booleano, sino que modifique el estado. Ejemplo:

```
class Documento
   validarAutomaticamente(otroDocumento)
```

```
      if esCopia(otroDocumento)
          marcarComoCopia()
      else
        marcarComoOriginal()
```

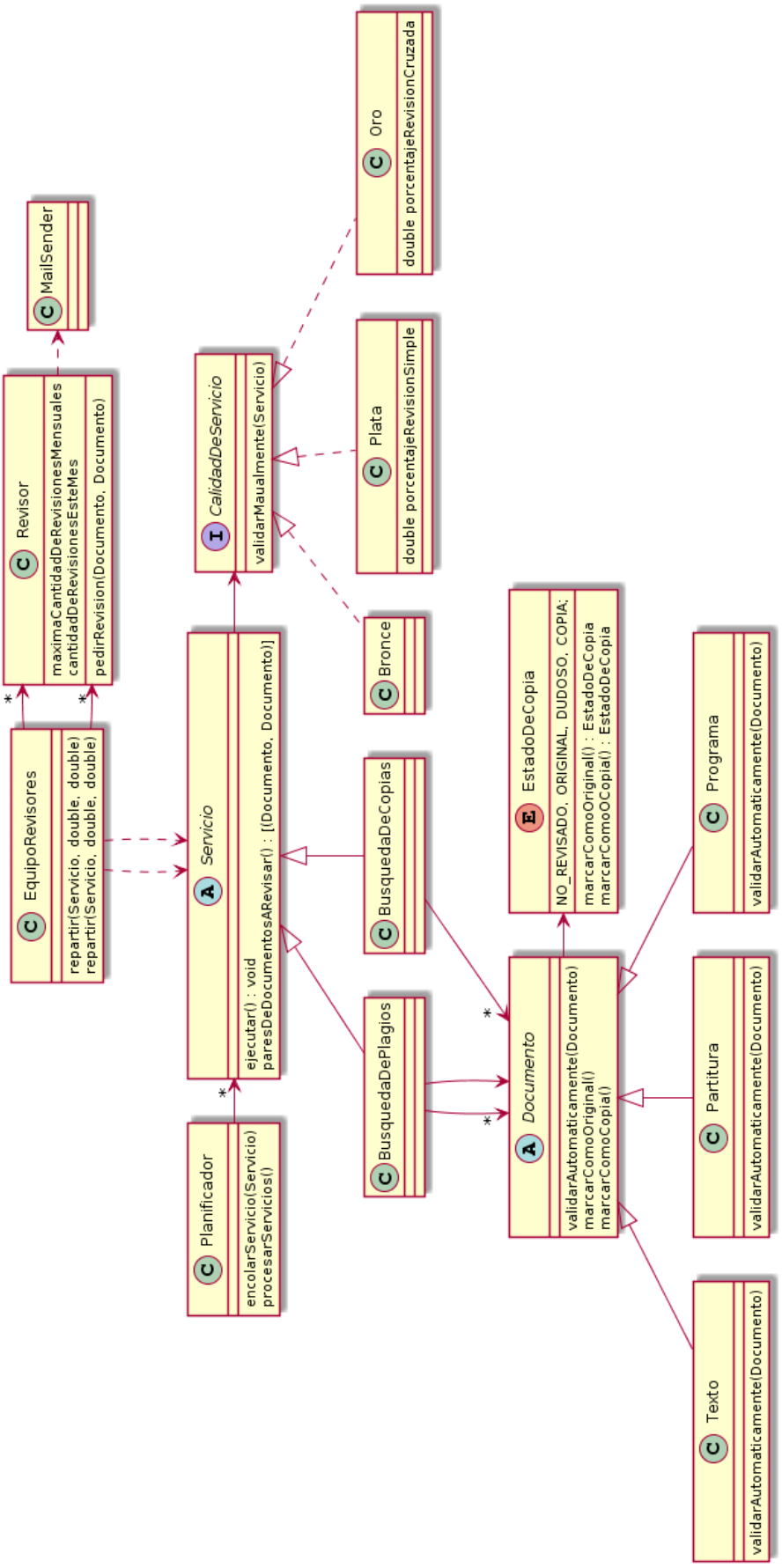
Construcción del servicio

Acá se podría hacer un builder, como por ejemplo:

```
ServicioBuilder.
   .new
   .original(unOriginal)
   .copiasPotenciales(unasCopiasPotenciales)
   .detercarPlagios()
   .calidadPlata(0.5)
   .build()
```

Esta solución depende mucho de la implementación

Diagrama de clases completo



Agrupamiento y "densidad" de dudosos

Este último punto es un análisis un poco más avanzado sobre la solución propuesta (y no un punto esencial del exámen).

La solución, tal como está propuesta y tal como está definido en el enunciado el comportamiento de los estados de copia, tiene dos problemas:

- podemos saber si un documento es una copia, pero no de quién;
- es casi imposible que un documento quede en estado COPIA, dado que que si es original, ninguna de las comparaciones (automáticas o manuales) lo marcará de esa forma, y si es una copia, habrá comparaciones que lo marcarán como COPIA y otras como ORIGINAL (¡porque no es copia de todos los documentos, sino de sólo algunos!).

Ambos problemas están relacionados y se pueden resolver, total o parcialmente, utilizando algún tipo de agrupación estructural o lógica. Por ejemplo:

- Antes de comparar automáticamente, agrupar los pares según su primer documento, y marcar al documento como copia si alguna de las comparaciones dio copia. Para eso habría que transformar la firma de Documento.`validarAutomáticamente` para que tome una lista de documentos, y no simplemente `otroDocumento`.
- Introducir una clase `Revision`: el documento conocerá una lista de revisiones, que estarán conformadas por los dos documentos a comparar, y lo que se marcará como original o copia será la revisión. Luego, el documento podrá decir su estado calculándolo a partir de agregar los estados de sus revisiones (si alguna es copia, será copia, si alguna es dudosa, será dudosa, si alguna es original, será original, si no será no revisado; esto se puede resolver ordenando la lista por prioridad o delegando en el estado). .