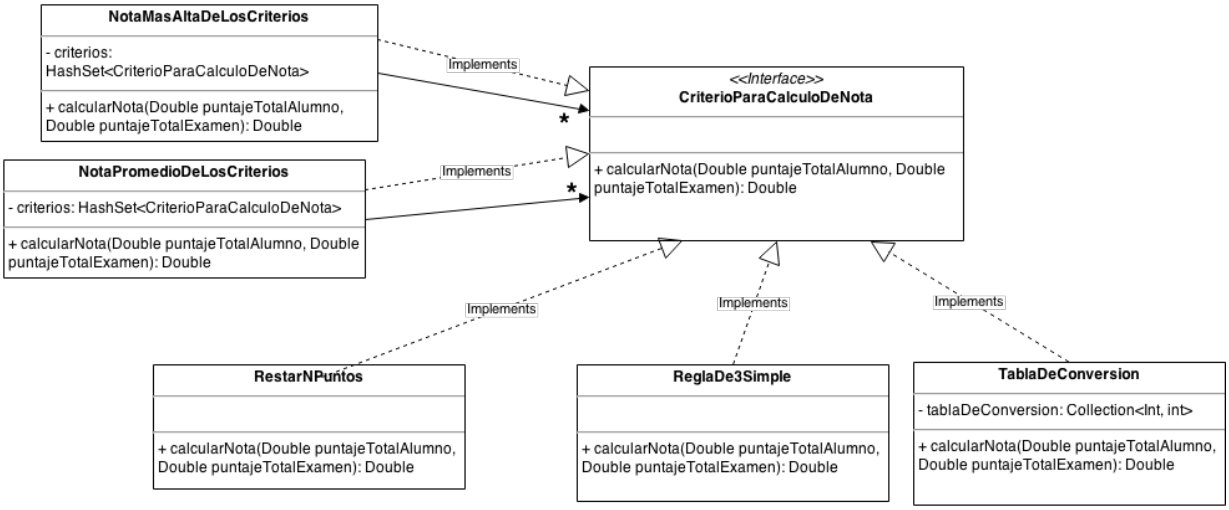
Criterios para calcular la nota:



Elegimos un tipo de diseño donde separamos los distintos criterios en varias clases individuales y no en una sola usando un atributo como tipo de criterio e IFs por las siguientes razones:

- Al separar los criterios en varias clases se genera una abstracción mejor con respecto al dominio (varios criterios que no comportamiento en común). Además es más cohesiva, porque aunque todos los métodos dentro de la solución de un Tipo de criterio seleccionando su comportamiento con IFs/Case apunten a devolver una nota, es todavía más cohesivo si cada clase solo se encarga de implementar su criterio. Dejando de hacer falta un método solamente para seleccionar el criterio correspondiente.
  Estas dos ventajas hace que tanto a nivel diseño como codigo se de una solución más expresiva.
- Es más flexible porque tanto para extender el diseño agregando nuevos criterios o modificando los existentes no tengo que modificar código en común con otros criterios.
- Para los criterios que dependen de otros criterios es necesaria un atributo con una colección de criterios pero para los criterios que no dependen de ningún otro es innecesaria.

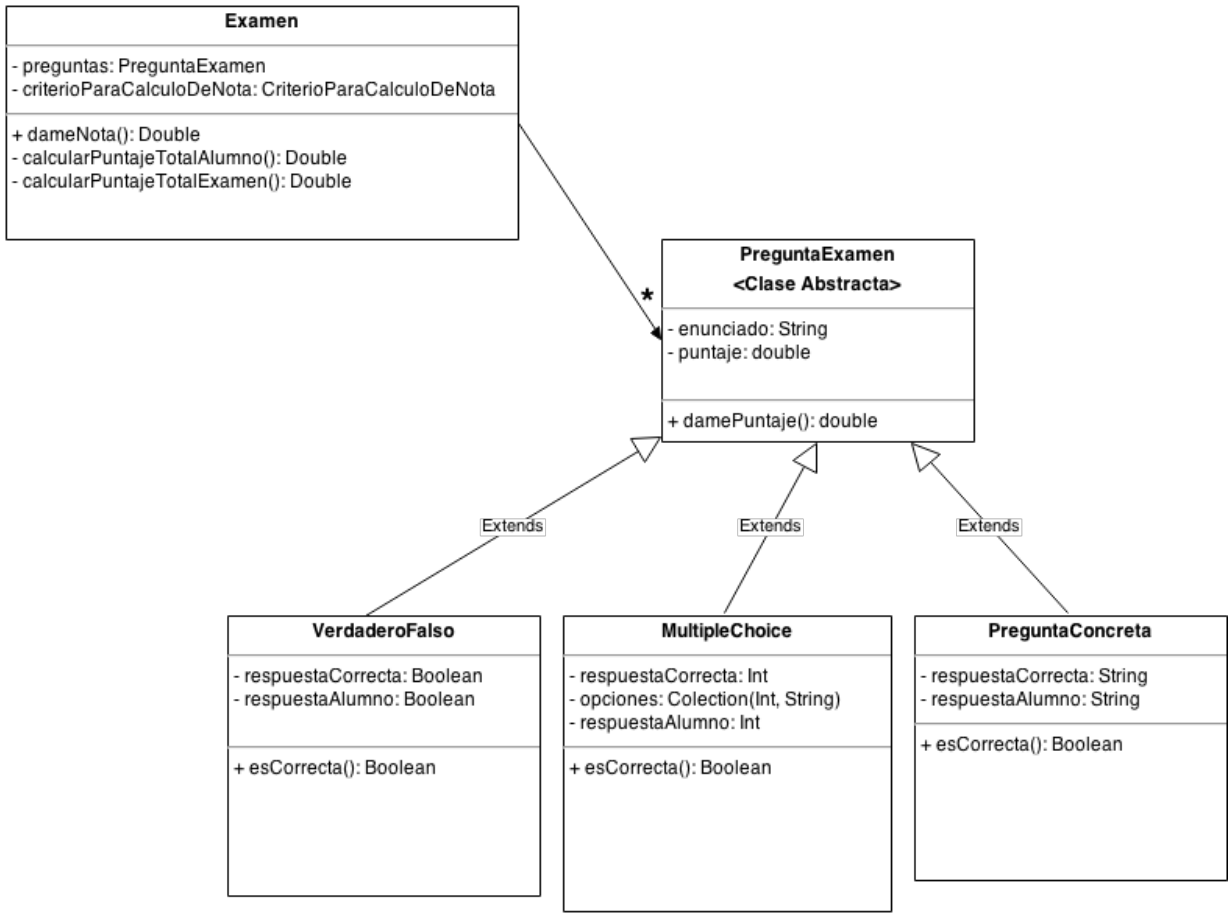
Se utilizó una interfaz en lugar de una clase abstracta porque no hay atributos ni comportamiento en común entre todos los criterios.

Por último no creímos necesario usar una clase de la que hereden las clases “NotaMasAltaDeLosCriterios” y “NotaPromedioDeLosCriterios”. Aunque tienen en común el atributo de coleccion de criterios y el comportamiento de obtener las notas de los criterios que dependen, el código que se ahorra es mínimo (usando las herramientas que tiene Java 8 para manejo de colecciones de orden superior) por lo que se estaría agregando una abstracción que haría más complejo el diseño innecesariamente.

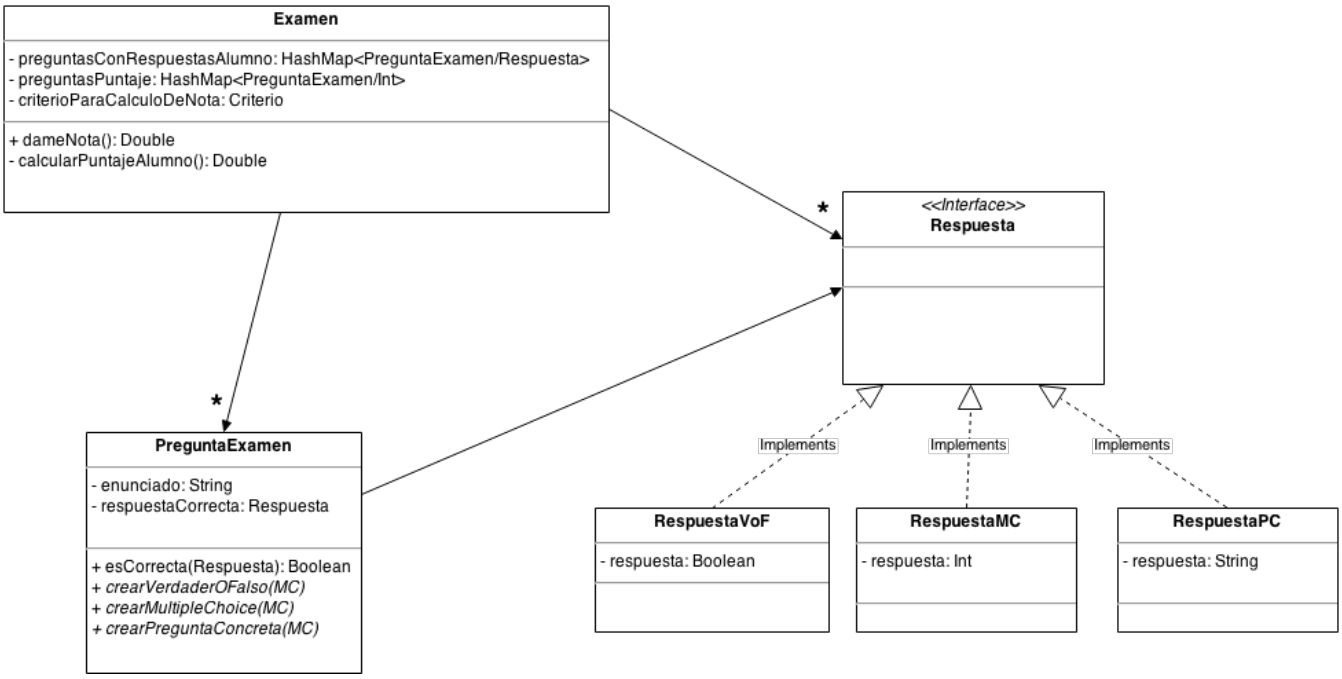
Tipos de preguntas y Examen:

En este punto tuvimos el impedimento que no se nos ocurría una manera óptima de separar la respuesta del alumno de la clase PreguntaExamen y agregarla como un atributo en el Examen manteniendo la relación con la pregunta que le correspondía (esto era para que se pueda reutilizar el objeto Pregunta en el mismo parcial pero para distintos alumnos, así como también el mismo objeto pregunta para muchos parciales diferentes).
Pensábamos hacer un HashMap<PreguntaExamen/RespuestaAlumno> en la clase Examen para relacionarlos pero no sabíamos como hacer para que la RespuestaAlumno soporte a la vez Boolean (VoF), Int (Multiple Choice) y String (Pregunta Concreta).
Principalmente por este motivo implementamos una solución que no cumplia ese requerimiento (ver mas abajo diagrama 1), y porque además entendimos que seguirle dedicando tiempo a ese punto no valía la pena siendo el objetivo principal del TP la práctica de testing automatizado.
Pero escribiendo este documento se nos ocurrió una forma de implementar una solución que posiblemente cumpliera el requerimiento mencionado (ver abajo diagrama 2).

1) Opción implementada



2) Opción que resuelve los requerimientos



De todas formas si se deja de lado el importante detalle de que la solución del diagrama 1 no cumple todos los requisitos, creemos que de todas maneras tiene un aporte realizar la siguiente comparación entre estas dos soluciones:

- La solución del diagrama 1 es más simple desde el lado del algoritmo que hay que implementar obtener los puntajes de las respuestas correctas (inclusive si se implementara como en el diagrama 2 un HashMap<PreguntaExamen/Int> para poder reutilizar los puntajes en los exámenes iguales y además desacoplando el puntaje de la pregunta para que puedan tener distintos puntajes en distintos exámenes). Se podría decir también que la clase Examen es menos cohesiva en el diagrama 2 porque esta tomando más responsabilidades.

  Cabe aclarar que lo dicho en el párrafo anterior tiene validez si no tomamos el requerimiento no cumplido del diagrama 1. O sea si se toma ese requerimiento no se le puede objetar KISS o YAGNI a la solución 2 porque fue una complicación necesaria para cumplir un requerimiento actual.

  Pero respecto a la clase PreguntaExamen del diagrama 2 es más simple por no estar separada en subclases.

- La solución del diagrama 2 es menos flexible ya que el Examen al tener la responsabilidad de obtener el puntaje si llegara a cambiar la forma de calcularlo para

cada tipo de pregunta (ej. si al VoF se agrega el atributo justificación y en caso de ser incorrecta se baja el puntaje a la mitad) se complica más que en el diagrama 1 que solo hay que redefinir un método.
Si surge un cambio de ese estilo pero para definir si una pregunta fue contestada correctamente en ambas soluciones no habría mayor problema porque se soluciona creando una subclase y redefiniendo el método.
De todas formas esta cualidad estos planteamientos no tiene demasiada relevancia porque no se tiene certeza que se puedan dar estas situaciones e iría en contra de YAGNI.
El hecho de haberlo separado en subclases en la solución 1 solo fue por una imposibilidad de poder resolver la necesidad de tener los atributos de respuesta con distintos tipos. No fue por una motivación de hacer más flexible la solución.

- La solución del diagrama 1 no solo tiene el problema del consumo excesivo de memoria por tener que instanciar para cada examen todas las preguntas y sus puntajes nuevamente si no que al tener un exceso de redundancia en la información hay más probabilidad de surgir errores a nivel negocio por hacer una carga de puntajes a una pregunta errónea o cargar una pregunta mal en un examen. Situación que es difícil de validar desde el programa por lo que lo hace menos robusta.

  En el diagrama 2 se presenta una situación al momento de agregar una respuesta del alumno en el HashMap<PreguntaExamen/Respuesta> en la cual por equivocacion se podria cargar una respuesta de un tipo que no corresponda y al momento de la comparación entre la respuesta del alumno y la respuesta correcta (dependiendo de cómo esté implementado el método esCorrecta(respuesta)) se detectaría rápidamente porque surgiría un error de tipos, lo cual es un error que afecta menos a la robustez de la solución. Este problema en el diagrama 1 no podría pasar porque los constructores de las preguntas ya obligan a que se carguen con tipos iguales las respuestas.