**Entrega 1**

**Métodos de entrada**

* **Punto 1**

1. Indicar si un usuario es válido: *persona.usuarioValido*

* **Punto 2**

1. Averiguar el índice de masa corporal o IMC de un usuario: *persona.imc*
2. averiguar si un usuario sigue una rutina saludable: *persona.sigueRutinaSaludable*

* **Punto 3**

1. Hacer que un usuario agregue una receta: persona.agregarReceta(unaReceta)

**Nota:** en el caso de querer agregar una receta válida la receta es agregada, de lo contrario se lanza un error.

1. Conocer las condiciones preexistentes para las que una receta es inadecuada: *receta.recetaInadecuadaPara*

* **Punto 4**

1. Saber si un usuario puede ver a una receta dada: *receta.puedeVermeOModificarme(unaPersona)*
2. Saber si un usuario puede modificar una receta dada: *receta.puedeVermeOModificarme(unaPersona)*
3. Modificar una receta dada, respetando la validación del item anterior: *persona.modificarReceta (unaReceta, unaModificación)*

**Nota:** en el caso de cumplir la validación la receta es modificada, de lo contrario se lanza un error.

* **Punto 5**

1. Poder construir una receta con subrecetas: *receta.agregarSubreceta(unaReceta)*

**Nota:** En el caso de ser receta compuesta la agregará sin problemas, de ser receta simple se lanza un error.

**Decisiones de diseño relevantes:**

* *Utilización del atributo dueño en vez de las clases recetaPública y recetaPrivada:*

Siguiendo la descripción del dominio para el trabajo solicitado vemos que, las potenciales clases recetaPública y recetaPrivada, implementarían los mismos métodos actuando de la misma manera, y sólo diferirían en lo que refiere a quién puede verlas o modificarlas. Dicho esto, consideramos de mayor utilidad y sencillez el uso de un atributo que refleje el dueño, y el acoplamiento de los métodos utilizados para la correcta resolución de la problemática a través de esta estructura. Obteniendo una única clase receta que posee métodos tales que le permiten saber si un usuario puede o no ver o modificar a la misma y realizar las modificaciones correspondientes según el caso.

* *Uso de un command para modificar una receta:*

Optamos por la utilización del patrón command a la hora de modificar una receta. Esto es así porque, el hecho de pasar mensajes por parámetro tratando a los mismos como objetos (que en este caso especifican cual va a ser la modificación a efectuar sobre la receta) nos pareció adaptarse de muy buena manera al requerimiento funcional de este punto.

* *Uso de strategy para el manejo de condicionesPreexistente:*

Debido a que -según el dominio del ejercicio- un usuario puede tener muchas condicionesPreexistentes diferentes, y como consecuencia de que estas condicionesPreexistentes deben saber responder a una cierta cantidad de mensajes en común, optamos por la utilización del patrón strategy.

Elegimos este patrón porque nos permite generar un “contrato” por el cual cada una de las clases que implementen la interfaz “se comprometen” a implementar los métodos definidos en ella, implementándolos según sea conveniente en cada una de estas clases. Logrando así generar clases de condicionesPreexistentes polimórficas que interpretan los mismos mensajes pero se comportan de forma totalmente diferente. Conformando una solución más óptima que una simple herencia, y que nos permite manejar con facilidad y de manera indistinta a cualquier condiciónPreexistente, sea del tipo que sea.

* *Uso de composite para manejo de receta, recetaSimple, recetaCompuesta*

Debido a que en el dominio se explicita que no hay límite de anidamiento de subrecetas nos vimos impedidos de implementar una clase recetaSimple con una sub clase recetaCompuesta (o viceversa) ya que esta solución se hubiera vuelto confusa a la hora de manejar un tipo u otro de receta. En consecuencia, optamos por la utilización del patrón composite, por ser un patrón que tiene como una de ventajas el generar un trato indistinto entre los elementos del patrón, ya sean estos objetos simples (recetaSimple) u objetos compuestos (recetaCompuesta), además de permitir el mismo el anidamiento ilimitado de recetas.

* *Uso de enums:*

Por otro lado destacamos el haber optado por la utilización de enums en vez de strings tanto para las preferencias de una Persona como para las temporadas correspondientes de una Receta.

Esto es así porque el manejo de enums es más preciso que el de strings, ya que justamente los enums son una clase enumerable con una cierta cantidad de valores claramente definidos y explicitados (que es justamente lo que se da a entender en el dominio), lo cual presenta una ventaja clara ante los strings, ya que estos últimos son simplemente una combinación de caracteres: hecho que puede llevar a errores de funcionamiento por una simple error de tipeo.

**Entrega 2**

**Métodos de entrada**

**- Punto 1**

a) Averiguar si una determinada receta puede sugerirse a un usuario o grupo de usuarios:

receta.puedeSerSugeridaA(unaPersona) o receta.puedeSerSugeridaA(unGrupoDeUsuarios)

**Nota:** optamos por una definición polimórfica del método, la cual permite que el mismo método pueda ser invocado tanto para ver si se puede sugerir una receta a una persona o a un grupo, obviamente teniendo en cuenta las diferencias propias que eso implica.

**- Punto 2**

a) Conocer todas las recetas a las que un usuario tiene acceso: persona.ListarTodasSusRecetas

**- Punto 3**

a) El usuario está interesado en una determinada receta deberá marcarla como favorita con el objetivo de dejarla en su historial para las búsquedas: persona.marcarRecetaComoFavorita(unaReceta)

**- Punto 4**

a) Resolver los filtros y el manejo de resultados mediante Decorators: filtroX.resultado

*Decisiones de diseño relevantes:*

* *Profundización en el uso de enums:*

En concordancia con la etapa anterior y como consecuencia de los nuevos requerimientos del sistema, donde nos vimos obligados a comparar elementos del tipo Preferencia (enum) con ingredientes de las recetas (Strings) optamos por llevar a cabo un refactor, el mismo implicó que los ingredientes dejen de tener un nombre del tipo String y que su nombre pase a ser del tipo Preferencia (enum)

* *Creación de la clase GrupoDePersonas:*

Optamos por introducir esta nueva clase ya que nos pareció la solución más adecuada a la extensión del dominio: se acopla perfectamente a los nuevos requerimientos acerca de la visibilidad y posibilidad de modificación de las recetas.

* *Modificación de la privacidad de la receta y creación del strategy ‘PrivacidadReceta’:*
* *Utilización de decorators para aplicar filtros y manejar resultados:*

**Entrega 3**

*Decisiones de diseño relevantes:*

* *Creación de una interfaz RepoRecetas:*

Al presentarse como nuevo requerimiento del sistema que el mismo podía utilizar tanto un repositorio interno como uno externo indistintamente nos vimos ante la necesidad de establecer una interfaz para todos los repositorios de forma tal que tratar con ellos sea algo uniforme.

* *Creación de RepoExternoAdapter:*

Dado que el repositorio externo provisto por la cátedra presentaba la dificultad extra se devolver los resultados en un formato diferente al formato interno a nuestro sistema nos vimos ante la necesidad de crear una clase encargada de interactuar entre el repositorio externo y los pedidos de nuestro sistema, de tal forma que los datos ingresen a nuestro sistema con un formato conocido y manejable.