DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

User Stories - Tutor: Nicolás Rinaldi

Ingeniería del Software 2

Segundo Cuatriméstre de 2013

Grupo 8

Integrante	LU	Correo electrónico
Julián Sackmann	540/09	jsackmann@gmail.com
Juan Pablo Darago	272/10	jpdarago@gmail.com
Vanesa Stricker	159/09	vanesastricker@gmail.com
Matías Barbeito	179/08	matiasbarbeito@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. User stories **3**

1.1. Sobre la herramienta . . . 3

1.2. Product backlog . . . 3

1.3. Sprint backlog . . . 9

1.4. Justificación del Sprint Backlog . . . 10

2. Detalle de diseño **10**

2.1. Diagrama de clases . . . 10

2.1.1. Consideraciones Generales . . . 10

2.1.2. Clase `ServicioDeEstado` . . . 11

2.1.3. Clase `Seguimiento` . . . 11

2.1.4. Clase `Configurador` . . . 12

2.1.5. Estadísticas . . . 12

2.1.6. Fases, Entrenamiento, Planes y Planificador . . . 13

2.2. Diagramas de secuencia . . . 13

2.2.1. Inicio de un seguimiento . . . 14

2.2.2. Actualización dentro de un seguimiento . . . 14

2.2.3. Crear un servicio de estados . . . 14

2.2.4. Finalización de un seguimiento . . . 15

2.2.5. Listado de estadísticas del usuario . . . 15

2.2.6. Notificar que la velocidad del usuario es demasiado baja dentro de un seguimiento 15

2.2.7. Crear un plan básico . . . 16

1. User stories

1.1. Sobre la herramienta

Para el armado del Product y Sprint Backlogs utizamos la herramienta Agilefant, accesible mediante la URL `http://agilefant.monits.com`. Se dispone de un usuario para el tutor, el cual tiene los siguientes datos de acceso.

- Nombre de usuario: nrinaldi
- Contraseña: monits

1.2. Product backlog

Incluimos a continuación las *user stories* con su valor, *story points* (que usamos como medida de su dificultad), criterio de aceptación y tareas asociadas. Estas son las correspondientes al *product backlog*.

1) Como atleta quiero tener disponible la lista de entrenamientos que ya concluí

   - Story Points: 8
   - Value: 5
   - Criterio de Aceptación:
     - El atleta puede ver una lista donde tiene cada entrenamiento.
     - Al elegir uno de ellos con su celular, se mostrará el detalle del mismo

2) Como atleta quiero poder ver estadísticas de los entrenamientos que ya hice

   - Story Points: 13
   - Value: 5
   - Criterio de Aceptación:
     - El atleta puede ver para una fase de un entrenamiento realizado el recorrido que hizo y verlo en un mapa similar al que tiene disponible cuando hace el recorrido en una fase.
     - El atleta puede ver para cada fase de un entrenamiento, estadisticas sobre su velocidad y distancia.
     - El atleta puede ver cuánta distancia recorrió para cada entrenamiento, cuanto tardo en cada uno, y cual fue su velocidad máxima.
     - Todos estos datos estarán disponibles para el atleta para cada entrenamiento en el detalle de los mismos que obtiene al seleccionarlo en la lista.

3) Como atleta quiero que la aplicación siga mi fase dentro del plan para que me avise si lo estoy siguiendo o tengo que modificar mi marcha.

   - Story Points: 8
   - Value: 21
   - Criterio de aceptación:
     - El atleta debe poder saber en que fase de que entrenamiento esta.
     - El atleta debe recibir una notificación de que debe aumentar la marcha si está yendo más lento que el plan.
     - El atleta debe recibir una notificación de que debe disminuir la marcha si está yendo más despacio que el plan.
     - El atleta debe recibir una notificación en intervalos regulares si se está manteniendo en una marcha aceptable.
     - Cada alerta por marcha inválida se repetirá a intervalos regulares mientras persista la condición.
     - En la interfaz gráfica debe aparecer un detalle de la razón de la marcha inválida (es decir, debe indicar si esta yendo más lento o rápido, y la diferencia entre la marcha ideal y la que lleva).
   - Tareas:
     - Investigar cómo reproducir una canción en cada formato estándar (mp3, wav, etc) dado.
     - Codificar la lógica para que según el tipo de alerta se eliga un sonido a reproducir (potencialmente leyendo de almacenamiento el mismo) y se lo reproduzca.
     - Codificar la lógica para según el tipo de alerta se eliga una canción aleatoria de las disponibles, se la lea de disco y reproduzca.
     - Codificar la lógica para que si la velocidad no esta en el rango, se envíe una alerta y se muestre en pantalla la diferencia y según ese rango, cuan “grave” es el nivel de alerta.
     - Testear para los 3 tipos de condiciones de marcha válida e inválida.

4) Como atleta quiero poder publicar los recorridos de los entrenamientos en redes sociales y aplicaciones de geolocalización.

- Story Points: 10
- Value: 1
- Criterio de aceptación
  - El atleta puede seleccionar en qué red social o aplicación publicar.
  - El atleta puede seleccionar qué recorrido puede publicar.
  - El atleta puede escribir un mensaje a agregar además de los datos de su entrenamiento.
  - La publicación es visible por los demás miembros de la red social de acuerdo a las reglas de privacidad de la misma.
  - Solo los datos explícitamente indicados por el usuario son publicados en la red social correspondiente.
  - El atleta puede decidir si quiere que se muestre el recorrido que realizó, la velocidad con la que corrió, etc., para cada tipo de dato a compartir.

5) Como atleta quiero poder ingresar los datos de mi estado físico.

- Story Points: 5
- Value: 8
- Criterio de aceptación
  - El atleta podrá ingresar su peso en kilogramos.
  - El atleta podrá ingresar su altura en cm.
  - El atleta podrá especificar mayores detalles usando categorías basadas en si ya corrió una carrera o no, ya corrió un maratón o no, su mejor marca de distancia en una carrera y en un maratón.
  - La aplicación guardará registro del valor actual de ambos datos.

6) Como atleta quiero poder ingresar mi frecuencia semanal con la que puedo entrenar.

- Story Points: 1
- Value: 5
- Criterio de aceptación
  - El atleta debe ingresar la frecuencia semanal con la que puede entrenar.
  - Los valores ingresados deben ser una cantidad de días entre 1 y 7.
  - La aplicación guardará registro del valor ingresado.

7) Como atleta quiero poder ingresar el objetivo de mi entrenamiento.

- Story Points: 8
- Value: 13
- Criterio de aceptación
  - El atleta debe ingresar sus objetivos propuestos entre las opciones:
    - Correr 5 km sin tiempo.
    - Terminar un maratón olímpico.
    - Correr 7 km en 35 minutos.

  y otras opciones y posibilidades decididas durante la implementación

8) Como atleta quiero poder ingresar el plazo estipulado para mi entrenamiento si así lo deseo.

- Story Points: 2
- Value: 5
- Criterio de aceptación
  - El atleta puede elegir un plazo estipulado para la finalización de cada uno de sus objetivos.

- Investigar que tipo de granularidad y que tipo de duraciones se pueden soportar: intervalos válidos, etc.

9) Como atleta quiero que las notificaciones de velocidad de la aplicación sean acordes al de batería seleccionado.

   - Story Points: 8
   - Value: 5
   - Criterio de aceptación:
     - Si el atleta eligió un consumo bajo, las notificaciones son pitidos y ocurren cada 1 minuto.
     - Si el atleta eligió un consumo alto de batería, las notificaciones son temas musicales preelegidos por la app y ocurren cada 10 segundos.
     - Para otros niveles de batería se determinará también una frecuencia de notificaciones y calidad de las mismas en el momento de la implementación.

10) Como atleta quiero que las actualizaciones de posición sean acordes al nivel de batería seleccionado.

    - Story Points: 8
    - Value: 2
    - Criterio de aceptación:
      - La posición se actualiza cada 10 segundos si el nivel de consumo batería elegido es alto.
      - La posición se actualiza cada minuto si el nivel de consumo de batería es bajo.
      - Para los demás niveles de batería también se indicará una frecuencia de actualización de posición al momento de la implementación.

11) Como atleta quiero poder ajustar la opción de consumo de batería.

    - Story Points: 3
    - Value: 2
    - Criterio de aceptación:
      - El atleta puede seleccionar dentro de los niveles disponibles, como mínimo bajo, medio y alto.
      - La aplicación debe poder correr más tiempo bajo un plan de consumo bajo que en uno alto.
      - El atleta puede determinar que impacto tiene en las funcionalidades de la aplicación el cambio de consumo de batería.

12) Como atleta quiero que la aplicación me de un plan de entrenamiento en base a los datos.

    - Story Points: 13
    - Value: 21
    - Criterio de aceptación
      - Si el atleta estableció como objetivo que desea correr una maratón, el sistema creará un plan concentrado en larga duración y velocidad constante.
      - Si el atleta estableció que desea correr una determinada cantidad de kilómetros en un cierto tiempo, el sistema creará un plan con entrenamientos de velocidad progresivamente más difíciles hasta alcanzar el objetivo.
      - Si el atleta no estableció requerimientos ni de distancia ni de tiempo, el programa devolverá una serie de entrenamientos recreativos.
      - Si el atleta se encuentra en buen estado físico, los entrenamientos constarán de fases con mayor exigencia.
      - La duración y velocidad devueltas serán inversamente proporcionales al peso del corredor, de acuerdo a los criterios de salud vigentes.

- La frecuencia semanal del plan de entrenamiento se corresponderá con la ingresada por el corredor al momento de dar la especificación de sus objetivos.

13) Como corredor quiero poder ver la velocidad promedio y la duración de cada fase de un entrenamiento para saber el criterio con el que la aplicación mide mi performance corriendo.

   - Story Points: 8
   - Value: 13
   - Criterio de aceptación
     - El atleta debe poder elegir un entrenamiento de los que la aplicación ha preparado.
     - El atleta debe poder examinar las fases de un entrenamiento.
     - El atleta debe ver para cada fase un rango de velocidades en km/h que son aceptables.
     - El atleta debe poder ver para cada fase, cuanto tiempo dura la misma en minutos.
   - Tareas:
     - Investigar cómo mostrar datos numéricos de velocidad y duración por la interfaz del celular, y como actualizar la vista cuando estos cambian.
     - Investigar un algoritmo para lograr calcular la velocidad promedio a medida que llegan los datos.
     - Testear que el promedio calculado es correcto incluso considerando actualizaciones de velocidad y tiempo poco frecuentes (por ejemplo en un modo de batería bajo).
     - Crear una vista para mostrar los datos
     - Implementar la lógica para calcular los datos de velocidad promedio.

14) Como atleta quiero que la aplicación me avise de la próxima fase del plan si ya pasó el tiempo.

   - Story Points: 8
   - Value: 21
   - Criterio de aceptación
     - La aplicación genera una notificación auditiva cuando se termine el tiempo de la fase actual y se pase a otra.
     - La aplicación genera una notificación cuando se terminó la última fase del entrenamiento.
     - No se genera esa notificación particular por otro motivo.

15) Como atleta quiero poder ver mi posición en el mapa en tiempo real.

   - Story Points: 13
   - Value: 21
   - Criterio de aceptación:
     - El atleta puede ver su posición actualizada a intervalos regulares en la pantalla.
     - Si se apaga la pantalla o bloquea el teléfono, al reanudar la aplicación la actualización de la posición se reanuda en forma automática.
     - Si se pierde señal de geolocalización, se notifica al usuario.
     - El atleta puede ver un *timestamp* en cada lugar donde se actualizó su posición.
     - El atleta puede desactivar geolocalización si lo desea
   - Tareas:
     - Investigar como obtener el tiempo actual del celular.
     - Implementar la lógica para obtener la posición actual del teléfono
     - Incorporar el uso de mapas de otras fuentes en la aplicación.
     - Agregar una vista con un mapa en la pantalla.
     - Implementar lógica para centrar ese mapa en una posición indicada.

- Investigar qué pasa cuando se pasa la aplicación al background y cuando vuelve, implementar la lógica que mantenga actualizando la aplicación incluso en background.
- Testear que efectivamente la posición se actualice al desplazar el teléfono
- Testear que el mapa se centre correctamente.
- Implementar la lógica para dibujar un recorrido en el mapa dados los puntos y un *timestamp* de los mismos.
- Si se pierde la señal de geolocalización, se notifica al usuario.

16) Como corredor quiero poder inicializar el seguimiento de un entrenamiento de los que me dio la aplicación y que están dentro del plazo para poder empezar a correr bajo el plan obtenido.

- Story Points: 13
- Value: 21
- Criterio de aceptación:
  - El atleta puede elegir un plan de la lista de disponibles y cuyo plazo no haya expirado.
  - El atleta puede seleccionar que el plan empiece a correr, y el mismo empezara el seguimiento en la primer fase.
  - El atleta puede detener el entrenamiento en cualquier momento.
  - El seguimiento termina de acuerdo a las fases del plan.
- Tareas
  - Implementar una vista de seguimientos filtrados por plazo.
  - Implementar que la selección del entrenamiento inicie la aplicación de seguimiento para el mismo.
  - Implementar que se pueda detener un plan de entrenamiento en cualquier momento del mismo.
  - Investigar maneras de ordenar la lista según relevancia de entrenamientos.

1.3. Sprint backlog

A continuación detallamos las *user stories* que incluimos en el actual *sprint*. Las mismas son:

- 3) Como atleta quiero que la aplicación siga mi fase dentro del plan para que me avise si lo estoy siguiendo o tengo que modificar mi marcha.
- 13) Como corredor quiero poder ver la velocidad promedio y la duración de cada fase de un entrenamiento para saber el criterio con el que la aplicación mide mi performance corriendo.
- 14) Como atleta quiero poder ver mi posición en el mapa en tiempo real.
- 16) Como corredor quiero poder inicializar el seguimiento de un entrenamiento de los que me dio la aplicación y que están dentro del plazo para poder empezar a correr bajo el plan obtenido.

y a continuación incluimos el detalles de las tareas involucradas y la dificultad asociada a cada una.

- Investigar cómo reproducir una canción en cada formato estándar (mp3,wav, etc).

  Dificultad: 3 - Horas: 5
- Investigar cómo guardar canciones en el teléfono y como volver a leerlas de almacenamiento permanente.

  Dificultad: 3 - Horas: 8
- Codificar la lógica para que según el tipo de alerta se elija un sonido a reproducir (potencialmente leyendo de almacenamiento el mismo) y se lo reproduzca.

  Dificultad: 5 - Horas: 8
- Codificar la lógica para que si la velocidad no esta en el rango, se envíe una alerta y se muestre en pantalla la diferencia y según ese rang, cuan “grave” es la alerta.

  Dificultad: 3 - Horas: 8
- Testear para los 3 tipos de condiciones de marcha válida e inválida.

  Dificultad: 2 - Horas: 5
- Investigar cómo mostrar datos numéricos de velocidad y duración por la interfaz del celular, y como actualizar la vista cuando estos cambian.

  Dificultad: 3 - Horas: 2
- Investigar un algoritmo para lograr calcular la velocidad promedio a medida que llegan los datos.

  Dificultad: 2 - Horas: 2
- Testear que el promedio calculado es correcto incluso considerando actualizaciones de velocidad y tiempo poco frecuentes (por ejemplo en un modo de batería bajo).

  Dificultad: 5 - Horas: 3
- Crear una vista para mostrar los datos.

  Dificultad: 3 - Horas: 2
- Implementar la lógica para calcular los datos de velocidad promedio.

  Dificultad: 5 - Horas: 2
- Investigar como obtener el tiempo actual del celular.

  Dificultad: 3 - Horas: 1
- Implementar la lógica para obtener la posición actual del teléfono.

  Dificultad: 8 - Horas: 2
- Incorporar el uso de mapas de otras fuentes dentro de la aplicación

  Dificultad: 3 - Horas 1

- Agregar una vista con un mapa en la pantalla.

  Dificultad: 5 - Horas: 1

- Implementar lógica para centrar ese mapa en una posición indicada.

  Dificultad: 3 - Horas: 1

- Investigar qué pasa cuando se pasa la aplicación al *background* y cuando vuelve, implementar la lógica que mantenga actualizando a la aplicación incluso en background.

  Dificultad: 3 - Horas: 2

- Testear que efectivamente la posición se actualice al desplazar el teléfono.

  Dificultad: 2 - Horas: 2

- Testear que el mapa se centre correctamente.

  Dificultad: 2 - Horas: 1

- Implementar la lógica para dibujar un recorrido en el mapa dados los puntos y un *timestamp* para cada uno.

  Dificultad: 5 - Horas: 5

- Implementar una vista de seguimientos filtrados por plazo

  Dificultad: 3 - Horas: 3

- Implementar que la selección del entrenamiento inicie la aplicación de seguimiento para el mismo.

  Dificultad: 5 - Horas: 5

- Implementar que se pueda detener un plan de entrenamiento en cualquier momento del mismo.

  Dificultad: 3 - Horas: 3

- Investigar maneras de ordenar la lista según relevancia de entrenamientos.

  Dificultad: 2 - Horas 5

1.4. Justificación del Sprint Backlog

Decidimos utilizar este Sprint Backlog puesto que consideramos que realizamos el trabajo requerido por el trabajo práctico, al mismo tiempo que estamos dando valor al usuario. En total, con los estimativos de horas realizados, tenemos 121 horas y 30 minutos, que considerando los 45 días para el trabajo práctico nos deja y descontando fines de semana nos deja entonces un total de aproximadamente por día para cada uno. Puesto que los cuatro miembros del grupo trabajan (sea en la industria o en la facultad como ayudantes) y cursan, nos pareció un buen estimativo para una cantidad razonable de trabajo. Esto por supuesto no considera la época de parciales, donde no nos fue posible seguir trabajando en el proyecto debido a compromisos de materias.

2. Detalle de diseño

2.1. Diagrama de clases

2.1.1. Consideraciones Generales

La figura **??** detalla el diagrama de clases realizado para el diseño orientado a objetos de la aplicación. El mismo fue realizado utilizado la semántica para un lenguaje dinámico e incluye tanto las clases correspondientes al modelo de negocios como las correspondientes a los controladores de las vistas del modelo `MVC`. Es necesario aclarar que éstas últimas no están completamente especificadas en el diagrama puesto que no nos pareció relevante para el trabajo práctico. Están por una cuestión de claridad de las distintas pantallas y cómo éstas, mediante los controladores, utilizan los objetos del modelo de negocios para lograr el objetivo de la aplicación. En el diagrama los controladores se muestran relacionados con aristas no dirigidas y sin aridad. Estas asociaciones representan “saltos”

entre vistas y relaciones entre controladores, no la canónica relación de conocimiento que denotan las uniones en los diagramas de clases UML. Esto es porque en iOS esta relación de conocimiento entre controladores se implementa a través del *NavigationController*. Incluirlo en el diseño no sólo acoplaría mucho el diseño a una plataforma particular, sino que elevaría su complejidad notablemente. Creemos que esta pequeña licencia en la semántica de UML nos permite expresar lo que deseamos (la relación entre los controladores del modelo `MVC`) manteniendo al diseño independiente de plataforma y acotada su complejidad.

Similarmente, no incluimos las clases correspondientes a las vistas propiamente dichas porque nos pareció que no aportaba en lo más mínimo a la claridad del diseño. Lo mismo aplica para la reproducción de sonido y la visualización en el mapa.

Del mismo modo, tampoco incluimos clases tales como

- unReal
- unEntero
- unBoolean
- unaFecha

porque los consideramos partes de la implementación de `Cocoa/Objective C` utilizados para la demo en iPhone y que son parte de cualquier librería standard.

2.1.2. Clase `ServicioDeEstado`

El objeto `ServicioDeEstado` tiene como principal función la de informar a los distintos componentes del programa el estado actual del teléfono en movimiento. A intervalos de tiempo regulares, configurados en última instancia por el consumo seleccionado de la batería, un objeto `ServicioDeEstado` le manda a todos los objetos a los que se hayan suscripto a él un paquete de actualización que incluye la velocidad a la que se está moviendo el dispositivo y su posición actual.

Esto corresponde al patrón `observer`. El propósito de esta decisión de diseño es que identificamos dos responsabilidades que no deseábamos acoplar: la obtención de una secuencia regular de mediciones y la utilización de dichas mediciones para diversos propósitos. En particular identificamos que dichas mediciones podían ser usadas para varios objetivos, tales como:

- Mostrar en tiempo real en el mapa la posición actual.
- Obtener estadísticas sobre el entrenamiento.
- Mostrar la velocidad actual de desplazamiento.
- Determinar el pasaje de fases dentro de un entrenamiento.
- Mostrar la velocidad a la que debería desplazarse el corredor para lograr la distancia implícita pautada en la fase.

Dado que esto corresponde con el patrón observer, decidimos utilizarlo para resolver el problema [**?**].

Se pueden observar ejemplos de uso de el `ServicioDeEstados` en los diagramas de secuencia **??** y **??**.

2.1.3. Clase `Seguimiento`

Un objeto `Seguimineto` tiene como funcionalidad principal la de mantener el estado de la fase actual del entrenamiento en curso. Para eso se apoya en otro objeto, el `IteradorEntrenamiento` que,

como su nombre lo indica, es un iterador de un objeto `Entrenamiento`, que mantiene cuál es la fase actual.

El patrón correspondiente a esto es el `iterator` ([**?**]). Decdimos utilizarlo puesto que consideramos que no es necesario que el seguimiento conozca al entrenamiento de forma total, sino que simplemente pueda iterar sus fases como corresponda. Esto reduce el acoplamiento entre el `Seguimiento` y el `Entrenamiento`, disminuyendo el impacto de futuras modificaciones a alguna de las antedichas clases.

El `Seguimiento` encapsula, por ejemplo, el cálculo de velocidad tentativa y además se ocupa de realizar las notificaciones auditivas funcionales a informar al corredor los cambios de velocidad necesarios y el cambio de fase dentro de un entrenamiento.

2.1.4. Clase `Configurador`

El `Configurador` encapsula la funcionalidad de modificar el comportamiento de la aplicación ante distintas configuraciones de nivel de batería. Al encapsular esta idea, se hace necesario que pueda influir sobre los objetos que realizan mediciones, para personalizar un *trade-off* entre precisión y consumo.

Para lograr esto, se utilizó el patrón `Factory` ([**?**]) para generar distintos objetos `Posicionador`, `Timer`, `Notificador` y `MedidorDeVelocidad`. Nuestro adaptación de este patrón al problema particular del TP fue realizar tres subclases de configurador, una para cada tipo de batería. Esto provee extensibilidad, ya que si se quisiera agregar otra configuración de batería, sólo sería necesario crear una nueva clase que implemente la interfaz definida por `Configurador`.

Entendemos que esto no es la mejor forma de resolverlo, ya que dificulta el cambio dinámico de estado de batería, puesto que hay que destruir el configurador y crearlo de nuevo. Podría mejorarse implementando un patrón `Decorator` (sea recursivo o iterando sobre una lista de métodos a aplicar). Sin embargo, dado que el usuario no puede cambiar la configuración de la batería mientras corre y que sea de la forma actual o con `Decorator` hay que destruir y crear un nuevo `Posicionador`, `Timer`, etc. decidimos dejarlo con el diseño actual, que es mucho más sencillo y, por ende, fácil de entender.

2.1.5. Estadísticas

La representación de las estadísticas en el diseño presentado tiene una complejidad no menor en pos de proveer extensibilidad y usabilidad.

En primer lugar se define una interfaz `Estadistica` que define que toda estadística debe implementar un método que reporte su valor. Además, las clase `Estadistica` hereda de `SuscriptorDeSerticioDeEstado`. Esto implica que todas las estadísticas deben implementar el método *actualizar (unEstado)* para poder suscribirse a las notificaciones del `ServicioDeEstados` y actualizarse pertinentemente (consideramos que es responsabilidad de cada estadística saber cómo actualizarse cuando le llega un nuevo paquete de `Estado`).

Por ejemplo, la estadística de velocidad máxima reporta como resultado final el máximo de las velocidades a las que se desplazó el corredor y su actualización cuando llega un nuevo paquete de `Estado` compara la nueva velocidad con la almacenada y se queda con la mayor.

La interfaz `Estadística` hereda dos subclases: `EstadisticaGeneral` y `EstadisticaDeEntrenamiento`, que, como sus nombres los indican, representan las estadísticas generales y las asociadas a un entrenamiento particular respectivamente. El motivo de esta separación es que podría haber estadísticas que no tengan sentido considerarlas para un entrenamiento particular (por ejemplo, si en el futuro se implementara almacenar información climática y nutricional a los entrenamientos, uno podría querer una estadística del tipo: “Velocidad promedio los martes lluviosos en los que entrené después de haber comido pastaflora”). No pondemos estadísticas para las fases porque nos pareció que en este punto no tenemos suficiente conocimiento del dominio como para determinar si las estadísticas por fase eran relevantes.

Para agregar un nuevo tipo de estadística, sólo es necesario determinar su tipo (general o de entrenamiento) y crear una nueva clase que implemente los métodos *actualizar(unEstado)* y *resultado() : unReal*.

Para mostrar las estadísticas se implementa la clase `EstadisticaMostrable`, que se construye con un objeto de la clase `Estadistica` y un nombre y tiene como único método "mostrar()", que devuelve un objeto apto para imprimir por pantalla en la plataforma que se esté implementando el modelo (en nuestro caso actual, un NSString). Se consideró hacer más extensible este diseño modificando la forma en la que se muestra una estadística utilizando un patrón `visitor`. Esto permitiría mucha más flexibilidad a la hora de mostrar o exportar una estadística. Esta idea no fue descartada por completo, pero la consideramos demasiado compleja para la primer iteración del `Scrum`. Es uno de los cambios a realizar en siguientes iteraciones.

2.1.6. Fases, Entrenamiento, Planes y Planificador

Para abstraer la lógica de planes del resto del sistema, decidimos utilizar una interfaz que denominamos `Planificador` que se ocupa de devolver un `Plan`. Un `Plan` consiste en una secuencia de `Entrenamientos` y un plazo que es el que le determina su validez (por ejemplo, cuando ya no logramos el objetivo en el plazo planteado, no tiene sentido mantener este plan en la lista de los realizables para el usuario). Cada `Entrenamiento` consiste en una serie de `Fases`. Cada `Fase` contiene una velocidad mínima y máxima que forman el rango aceptable para correr, y una duración. De esta manera, no tenemos acoplamiento entre la lógica que produce un plan y el resto del sistema, porque solo se conocen en `Plan` que es algo desconectado de la lógica de negocio que lo trajo al mundo.

Dado que desconocemos del dominio lo suficiente como para poder realizar una implementación fidedigna de acuerdo a parámetros médicos y deportivos, decidimos tratar de proveer un diseño extensible pero dentro de las limitaciones indicadas en el trabajo práctico. Para ello incluimos en el diseño un planificador básico.

El `Planificador Básico` emplea los datos indicados en el trabajo práctico para obtener el plan: Las características de la persona (peso y altura), un objetivo(que consiste en indicar cuantos kilómetros se desea correr, y opcionalmente en cuanto tiempo desea poder correrlo), su disponibilidad (en horas por semana), un plazo estipulado (como una fecha límite) y su estado actual (cuantos kilómetros fue lo máximo que corrió y en cuanto tiempo lo hizo).

Decidimos restringirnos a este diseño puesto que no nos pareció que tuviéramos el suficiente conocimiento como para realizar más detalles, y no tomamos una postura de pensar más en extensibilidad puesto que el trabajo práctico no indica que pueda haber cambios en este área.

Algunos de estos factores nos pareció pertinente incluirlos como parte de un `PerfilCorredor` y tomarlo como parte del `Corredor` (que además de esto incluye todos sus planes). Los demás factores son específicos a cuando se crea un plan, por lo tanto son conocidos y creados por el `NuevoPlanControlador` que es el que toma estos datos al preguntarle al usuario cuando este desea crear un nuevo plan.

El objetivo puede o no tener un tiempo asignado de corrida. Para esto, delegamos esta responsabilidad en un objeto `TiempoParaDistancia`. Las subclases de este implementan distinto comportamiento. Dado el conocimiento que tenemos del dominio, decidimos que lo que nos interesaba de este valor es compararlo con otro, para saber si es menor o igual que otro. En este caso, un limite sin restricciones es mayor a cualquiera y nunca es menor a otro. De esta manera se puede implementar cualquier tipo de algoritmo en el que sea necesario utilizar rangos de tiempos (sea cual sea).

2.2. Diagramas de secuencia

Para clarificar algunas de las partes del código que consideramos más difíciles de plasmar en un diagrama estático, incluimos diagramas de secuencia para diversas partes del trabajo práctico. Estas partes corresponden a diversas situaciones donde mostramos como podría ser una implementación de la lógica secuencial que se desarrolla.

2.2.1. Inicio de un seguimiento

Este diagrama de secuencia muestra el proceso de inicio de seguimiento. A grandes razgos, el controlador de fase crea un servicio de estado (que será le encargado de mandar las actualizaciones de estado a otros componentes). Esta interacción es un poco compleja para ponerla en el mismo diagrama, así que se la separó en el diagrama "Crear un servicio de estados", que se puede ver en la sección 2.2.3.

Luego se procede a suscribir a todos los objetos que deben recibir actualizaciones de estado al controlador (creándolos si no existían), para lo que se le manda repetidas veces el mensaje *suscribir (unObjeto)* al `ServicioDeEstado`. Antes de eso, se suscribe al `ControladorDeSeguimiento` al `Seguimiento` para que este pueda informarle al primero cuando se finalizó el seguimiento. Esto es posible porque el `ServicioDeEstados` implementa la interfaz `SuscriptorDeFinSeguimiento`.

Finalmente, se inicializa el servicio de estados, enviándole el mensaje *iniciarLectura ()*, lo que provoca que éste (que implementa la interfaz `SuscriptorDeTimer`), le envíe un mensaje *suscribir (self)* al timer. De esa forma queda suscripto el notificador de estados al timer y éste le enviará notificaciones a intervalos regulares.

Suponiendo que el usuario ya realizó la configuración inicial y tiene algún plan con entrenamientos seteados, el camino por el que debe navegar para llegar a este diagrama de secuencia es:

- Seleccionar uno de sus planes de la lista de planes disponibles.
- Seleccionar un entrenamiento de la lista de entrenamientos del plan.
- En esa pantalla (donde se listan las fases correspondientes al entrenamiento) se presiona "comenzar".

2.2.2. Actualización dentro de un seguimiento

Este diagrama muestra la secuencia de mensajes intercambiados a la hora de realizar una actualiación dentro de una fase, mientras el usuario está corriendo. Los pasos para llegar a este mensaje son los mismos que el anterior, esperando el tiempo correspondiente a la frecuencia de actualización que se haya realizado.

En primer lugar, el objeto `Timer` recibe un mensaje *tic ()*. Este mensaje está en el diseño para que éste quede agnóstico a la implementación de la plataforma sobre la que se está trabajando. En el caso particualr de *iOS*, este mensaje no existe sino que se le especifica al sistema operativo que un método en particular (en este caso el *actualizar*) debe ser invocado regularmente cada cierto tiempo. Cuando el `Timer` recibe ese mensaje, le informa al `ServicioDeEstado` (pues es el único que está suscripto al `Timer`) enviándole el mensaje *actualizar ()*.

Al recibir este mensaje, el `ServicioDeEstado` le solicita a sus colaboradores internos la velocidad y posición actual del teléfono (mediante los mensajes *velocidadActual ()* y *posición ()* respectivamente) y crea un nuevo `Estado` con esa información.

Finalmente le envía a cada uno de sus suscriptores el mensaje *actualizar (unEstado)*, para que estos puedan realizar sus respectivas acciones con la nueva información del estado.

2.2.3. Crear un servicio de estados

En este diagrama se puede observar el proceso de creación de un objeto `ServicioDeEstado`, que se crea al comenzar el entrenamiento, como se puede ver en la sección 2.2.1. Para crear el `ServicioDeEstado` se pasa como parámetro un `Configurador`, del que se obtienen y setean como colaboradores internos el `Timer`, el `Posicionador` y el `MedidorVelocidad`. El configurador crea dinámicamente estos elementos a medida que se le son solicitados y los devuelve.

Una vez que todos los elementos están seteados, se devuelve el recién creado `ServicioDeEstado`, sin que éste se suscriba al `Timer`.

Para la creación del servicio de estado se nos planteó una disyuntiva:

- Una opción era no guardar el `Timer` como colaborador interno del `ServicioDeEstado`, sino que simplemente cuando a la hora de crear el `ServicioDeEstado`, se pida al `Configurador` el `Timer` y el `ServicioDeEstado` se suscriba en ese momento. Esta alternativa (no elegida) tenía la ventaja de que evitaba almacenar como colaborador interno al `Timer` y nos ahorraba el método *iniciarLectura ()*. Sin embargo tiene un defecto: si la frecuencia de actualización era demasiado rápida, podía llegar a llamarse al mensaje *actualizar ()* del servicio de estado (por un *tic ()* del sistema operativo) antes de que todos sus colaboradores internos estuvieran registrados, potencialmente perdiendo información de los primeros estados.
- La alternativa que se nos ocurrió para esto es que se almacene el `Timer` como colaborador interno del `ServicioDeEstado` y que una vez que el controlador termine de suscribir a quienes corresponda, llame a un método del `ServicioDeEstado`(*iniciarLectura ()*) que se ocupe de suscribirlo al `Timer`. Elegimos esta alternativa porque pensamos que en un futuro puede existir algún objeto (por ejemplo una nueva `Estadistica`) para el que sea importante no perder los estados iniciales.

2.2.4. Finalización de un seguimiento

Este diagrama muestra cómo es el proceso de finalización de un seguimiento cuando concluye un entrenamiento. El `Seguimiento`, encargado de seguir el proceso de entrenamiento, se da cuenta de que el entrenamiento finalizó (puede haber sido porque el usuario cumplió el tiempo y distancia correspondientes a la última fase del entrenamiento en curso o porque apretó el botón de "detener" en la vista de seguimiento) y se le informa al `ServicioDeEstados` que deje de tomar lecturas del `Timer`. Para eso, le envía el mensaje *finLectura ()*.

Acto seguido, el controlador informa al `StoreEstadisticasGenerales` que guarde las estadísticas generales que se estuvieron actualizando durante el entrenamiento (mediante el mensaje *guardar ()*) y procede a crear un nuevo objeto `EntrenamientoConcluido` para almacenarlo en su correspondiente `StoreEntrenamientoConcluido`. Para eso, obtiene el `Recorrido` del `GeneradorDeRecorridos` y, usando sus colaboradores internos `Plan` y `CollectionEstadisticasEntrenamiento`, llama al *new (unPlan, unaColeccionDeEstadisticas, unRecorrido)* de la clase `claseEntrenamientoConcluido`. Esto le devuelve un objeto `EntrenamientoConcluido`, que agrega al `StoreEntrenamientoConcluido`.

2.2.5. Listado de estadísticas del usuario

2.2.6. Notificar que la velocidad del usuario es demasiado baja dentro de un seguimiento

En este diagrama se observa el conjunto de mensajes que ocurren durante el un entrenamiento cuando el corredor se está desplazando más rápido de lo que su fase actual indica. Como se explicó en la sección 2.2.2, cuando el `Timer` recibe el hipotético mensaje *tic ()*, le avisa al `ServicioDeEstado` y éste le reporta a sus suscriptores (en particular al `Seguimiento`), el estado actual del corredor.

Cuando el `Seguimiento` recibe el mensaje *actualizar (unEstado)*, extrae del parámetro la velocidad actual a la que se está desplazando el teléfono (enviándole al `Estado` el mensaje *velocidadActual ()*) y le pide a su colaborador interno `IteradorDeFase` que le informe cuál es la fase actual, a la que le pregunta las velocidades mínima y máxima que corresponden a esa fase (mensajes *velocidadMinima()* y *velocidadMaxima ()* respectivamente).

Con esos tres datos, el `Seguimiento` verifica que la velocidad actual esté en rango y, suponiendo que no lo está (que está por encima de la máxima) le envía un mensaje a su colaborador interno `Notificador` diciéndole que notifique que la velocidad actual es muy alta (*notificarVelAlta ()*).

2.2.7. Crear un plan básico