FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1 — Smalltalk

[7507/9502] Algoritmos y Programación III
Curso 1
Segundo cuatrimestre de 2018

Alumno:	Diaz,Nicolas Andres
Número de padrón:	101369
Email:	diaz.nicolasandres@gmail.com

Índice

1. Introducción

El presente informe reune la documentación de la solución del primer trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación de un sistema de calendario y eventos en Pharo utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso.

2. Supuestos

Debido a que ciertas especificaciones no fueron provistas por la cátedra y en ciertas ocasiones el criterio de cómo abordar alguna situación quedaba a cargo del alumno, se tuvieron en cuenta los siguientes supuestos.

- No puede existir otras persona/recurso con el mismo nombre.
- No pueden existir dos eventos con el mismo nombre.
- Si una persona esta ocupada y es agregada a un evento. Asistirá solo al último, ya que solo se puede estar en un lugar a la vez.
- El evento semanal por lo menos tiene que acontecer una vez.

3. Modelo de dominio

Existe una clase principal llamada Calendario, en la cual están los métodos principales, que se encargan de gestionar el agregado de objetos Persona, Recurso (los cuales heredan comportamiento de la clase Asistente).A su vez también conoce e interactúa con los objetos Evento, ya sea para agregarlos al calendario o removerlos. Para delegar comportamiento se dividió el agregado de eventos en simples y semanales en clases correspondientes, en el cual se hace claro ejemplo de polimorfismo sin herencia. A su vez estas delegan compartimiento a las clases hijas de Asistente. Estas tienen conocimiento de los eventos a los cuales asistirían. Clases utilizadas:

- Calendario.
- Persona.
- Recurso.
- EventoSimple.
- EventoSemanal.

4. Diagramas de clase

En este diagrama de clase podemos observar los métodos de las clases que utilice, los cuales se detallaran en la sección implementación. La clase Calendario contiene un diccionario de asistentes en el cual se encontraran objetos Persona/Recurso. A su vez contendra un diccionario en el cual están los objetos EventoSimple y EventoSemanal. Estos dos ultimas clases poseen un diccionario de Personas/Recurso, y estas ultimas un diccionario donde key es la fecha del evento y el valor el nombre del mismo.

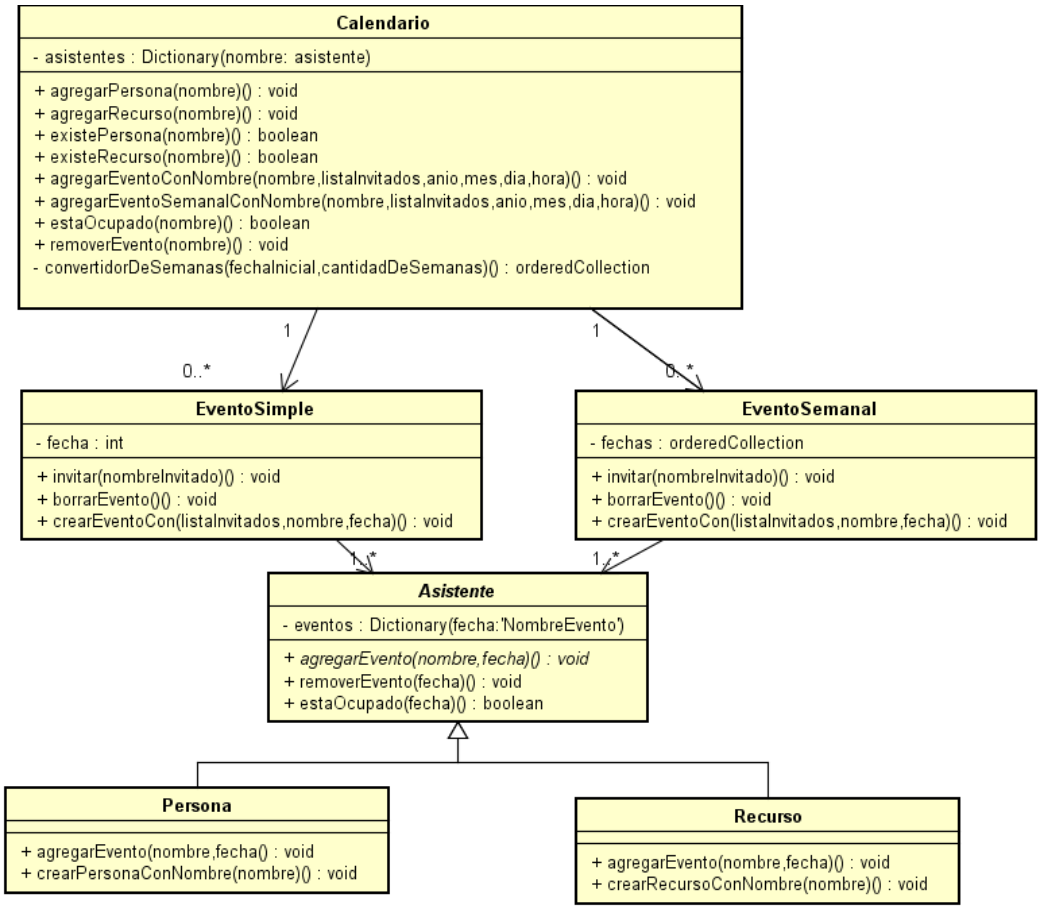

Figura 1: Diagrama del Calendario.

5. Detalles de implementación

Aquí detallare las implementaciones del trabajo, en algunos casos añadiendo una porción de codigo para su mejor entendimiento.

5.1. Implementación de Calendario

La clase Calendario tiene como atributos un Dictionary de asistentes se almacenan de la siguiente manera (nombrePersona/nombreRecurso: objeto Persona/Recurso)y uno de eventos (nombreEvento: eventoSimple/eventoSemanal).

```
initialize
asistentes := Dictionary new.
eventos := Dictionary  new.
```

Metodos agregarPersona / agregarRecurso.

Crea un objeto del tipo Persona/Recurso y lo añade al diccionario asistentes. ej: agregarPersona

```
persona := Persona crearPersonaConNombre: nombrePersona.
asistentes  at: nombrePersona put: persona.
```

Metodos existePersona/existeRecurso.

Comprueba si existe la key con el nombre del Recurso/Persona y retorna true en caso verdadero, false caso contrario. ej: existeRecurso

```
existeRecurso: nombreRecurso
  ^ (asistentes includesKey: nombreRecurso)
```

Metodo estaOcupado.

Recibe el nombre de la persona o el recurso y la fecha correspondiente. Convierte la fecha al tipo DateAndTime para facilitar su manejo. Y envia el mensaje al objeto persona/recurso delegándole la responsabilidad y este responderá si esta ocupado o no.

```
fechaEvento := DateAndTime year: anio month: mes day: dia hour: hora minute: 0.
 ^(asistentes at: nombre) estaOcupado: fechaEvento.
```

Metodo privado convertidorDeSemanas.

Recibe la fecha y la parsea al tipo DateAndTime, junto a la cantidad de semanas que durara el evento. Devuelve una lista con las fechas de los eventos distanciados por 7 días.

```
(semanas-1) timesRepeat: [
fechaAux   := fechaAux  +(Duration days: 7).
listaFechas add: fechaAux ].
```

Metodo agregarEventoConNombre.

Recibimos una lista de strings con el nombre de la personas y recursos invitados al evento, junto a la fecha del mismo. Creamos una ordered collection en la cual vamos a añadir la referencia al objeto persona/Recurso que recibimos por parámetro accediendo a el vía diccionario con la key: Nombre. Creamos un objeto del tipo EventoSimple, al cual le pasamos la orderedCollection, junto a la fecha parseada y este se encargara de crear el evento. Por ultimo insertamos este objeto evento en el diccionario del calendario

```
listaObjetosAsistentes := OrderedCollection new.
listaAsistentes do: [:nombreAsistente | listaObjetosAsistentes
add: (asistentes  at: nombreAsistente) ].

evento:= EventoSimple  crearEventoCon: listaObjetosAsistentes
conFecha:fechaEvento   conNombre: nombre.

eventos at: nombre put: evento.
```

Metodo agregarEventoSemanalConNombre.

Su comportamiento es parecido al de agregarEventoConNombre, pero con la diferencia que crea un objeto del tipo eventoSemanal y en vez de pasarle una unica fecha, utiliza el método convertidorDeSemanas y le pasa una lista de fechas, junto al orderedCollection de personas/recursos invitados y es la instancia de eventoSemanal quien se encarga de crear el evento. Esta misma se almacena en el diccionario de eventos junto a los simples.

```
listaFechas := self  convertidorDeSemanas: fechaEvento  cantidadSemanas: semanas.
evento:= EventoSemanal crearEvento: listaObjetoAsistentes
conFechas:listaFechas   conNombre: nombre.

eventos at: nombre put: evento.
```

Metodo removerEvento.

Este es un caso en el cual se puede apreciar claramente el uso del polimorfismo sin herencia, ya que en el diccionario eventos se encuentran dos objetos capaces de responder al mismo mensaje a pesar de tener un comportamiento distinto. Accedemos al evento a través de la key nombre en el diccionario y a este se le delega el borrado de si mismo.

```
(eventos at: nombreEvento) borrarEvento.
```

5.2. Implementacion eventoSimple

Evento simple tiene un Dictionary en el cual guarda una referencia a los Asistentes(Personas/Recursos), al inicializarlo se le asigna la fecha y nombre de evento y luego se prodece a llamar el método invitar(se detalla mas adelante).

```
initialize
    evento := EventoSimple new.
    evento setNombre: nombreNuevo.
    evento setListaAsistentes: listaAsistentes.
    evento setFecha: fechaNueva.
    evento invitar.
```

Metodo invitar.

Procede a recorrer el orderedCollection de asistentes y a cada objeto persona/Recurso le delega la responsabilidad de asignarse el evento enviando el mensaje agregarEvento.

```
invitar
     asistentes do: [ :eachAsistente | eachAsistente agregarEvento: nombre conFecha: fecha]
```

Metodo borrarEvento.

Procede a recorrer el orderedCollection de asistentes y a cada objeto persona/Recurso le delega la responsabilidad de borrar quitar ese evento de su lista.

```
borrarEvento
     asistentes  do: [:each | each removerEvento: fecha ].
```

5.3. Implementacion eventoSemanal

El constructor es muy similar al de eventoSimple, con la diferencia que recibe una lista de fechas, que luego invitar se encarga de utilizarla para crear los eventos.

Metodo invitar.

Recorre la lista de fechas y por cada fecha tiene el mismo comportamiento que el método invitar de eventoSimple.

```
invitar
     fechas do: [:fecha | asistentes do:
         [ :eachAsistente | eachAsistente agregarEvento: nombre conFecha: fecha]].
```

Metodo borrarEvento.

Recorre la lista de fechas y por cada fecha tiene el mismo comportamiento que el método borrar de eventoSimple.

```
borrarEvento
    fechas do: [:fecha | asistentes do:
    [ :eachAsistente | eachAsistente removerEvento: fecha]].
```

5.4. Implementacion clase Asistente

Una de las decisiones que tuve que tomar para esta parte del trabajo era si utilizar herencia o no, como quedo demostrado en las clases eventoSimple y eventoSemanal se puede aplicar polimorfismo sin la misma. Pero opte por si utilizarla debido a que se recomendaba que no haya codigo respetido, y en este caso el comportamiento de ambas clases es casi idéntico, solo varia en que Recursos no admite una superposición de eventos. Esta clase posee un diccionario en el que almacena (fecha: nombreDeEvento). Las clases Persona y Recurso tienen su propio constructor que se detalla en su apartado correspondiente.

Metodo estaOcupado.

Recibe una fecha y corrobora si el evento esta en el diccionario

```
estaOcupado: fecha
    ^ (listaEventos includesKey:  fecha)
```

Metodo removerEvento.

Recibe una fecha y la elimina del diccionario

```
removerEvento: fecha
    listaEventos removeKey:  fecha
```

5.5. Implementación clase Persona

Hereda el comportamiento de la clase Asistente y redefine el método agregarEvento. En el constructor setea el nombre de la persona.

Constructor.

```
persona := Persona new.
persona setNombre: nombre.
```

Metodo agregarEvento.

Agrega al diccionario (fecha: nombreDeEvento)

```
listaEventos at: fecha put: conNombre
```

5.6. Implementación clase Recurso

De igual manera que la clase Persona, hereda el comportamiento de la clase Asistente y redefine el método agregarEvento. En el constructor setea el nombre de la persona.

Constructor.

```
recurso := Recurso new.
recurso setNombre: nombre.
```

Metodo agregarEvento.

Agrega al diccionario (fecha: nombreDeEvento), si este ya estaba ocupado lanza RecursoOcupadoError

```
(listaEventos includesKey:  fecha) ifTrue: [RecursoOcupadoError signal].
listaEventos at: fecha put: conNombre
```

6. Excepciones

YaExistePersonaError Es lanzada cuando intenta crearse una persona con un nombre ya existente

YaExisteRecursoError Es lanzada cuando intenta crearse un recurso con un nombre ya existente

YaExisteEventoConEseNombreError Es lanzada cuando intenta agregarse un evento con un nombre ya existente.

RecursoOcupadoError Es lanzada cuando quiere se quiere agregar un recurso a un evento, pero este ya esta ocupado.

NombreNoEncontradoError Es lanzada cuando se pregunta por el estado de una persona/recurso y este no existe en nuestro diccionario.

EventoInexistenteError Es lanzada cuando se quiere eliminar un evento no existente.

CantidadDeSemanasMenorAUnoError Es lanzada cuando se quiere crear un evento semanal con una cantidad de semanas menor a uno.

7. Diagramas de secuencia

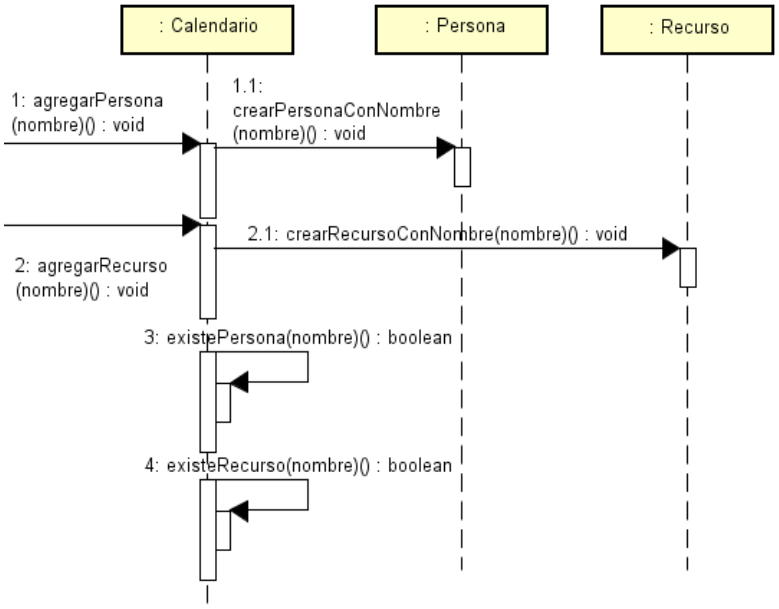

Figura 2: Agregar Persona/Recurso y existen.

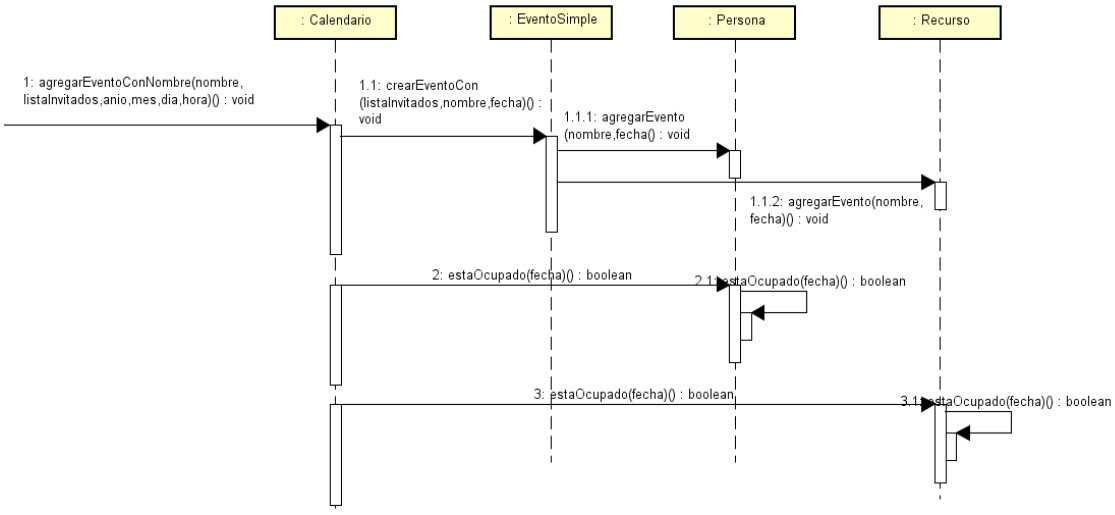

Figura 3: Al agregar persona y recurso a un evento, están ocupados.

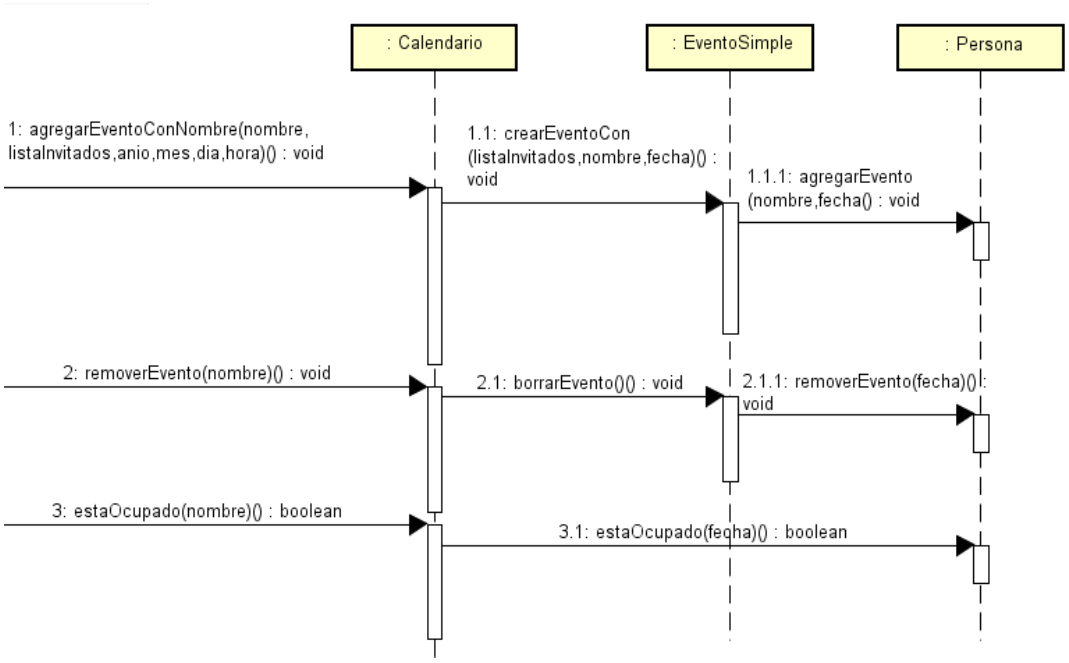

Figura 4: Al remover un evento, la persona no estará ocupada en esa fecha.

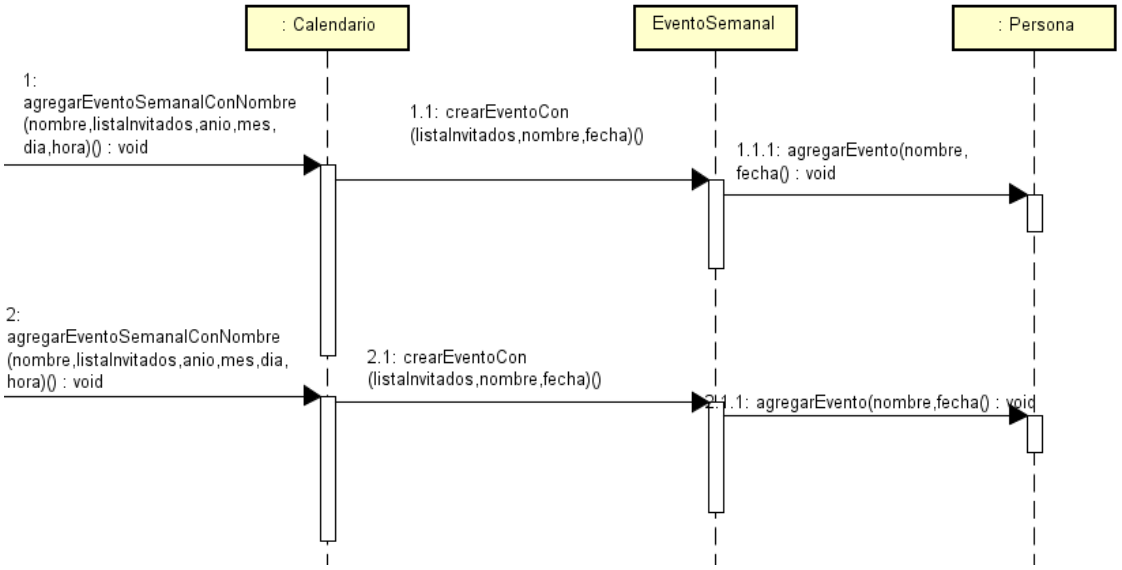

Figura 5: Persona ocupada, admite superposición .

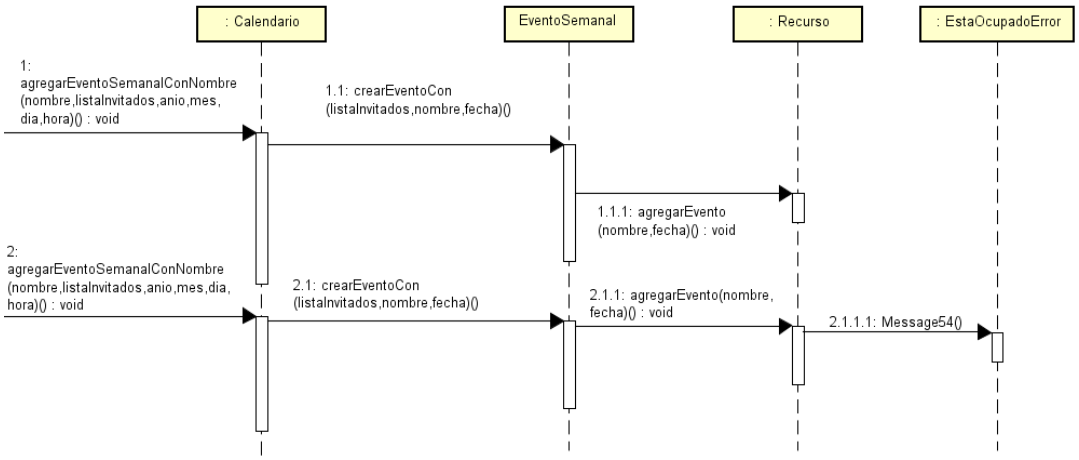

Figura 6: Recurso ocupado, no admite superposición .