FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1 — Smalltalk

[7507/9502] Algoritmos y Programación III
Curso 2
Primer cuatrimestre de 2018

Alumno:	STROIA, Lautaro E
Número de padrón:	100901
Email:	lautaro.stroia@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 Calendario en Pharo utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso.

2. Supuestos

Uno de los supuestos que tuve que tomar a la hora de realizar el TP, fue darme cuenta que la duración de los eventos simples era de solo 1 hora, ya que había pruebas que así lo especificaban, por lo que a esos eventos tuve que ponerles esa duración y así lograr pasar las pruebas.

Otro de los supuestos fue gestionar a los recursos y personas dentro de una clase abstracta Invitado, asi cada subclase hereda los mensajes implementados en la clase madre, o la clase madre le delega el comportamiento a cada subclase hija para así responder de distinta manera al mensaje.

Otro detalle es que, en mi modelo, no me pareció mal que haya varias personas con el mismo nombre, ya que una misma persona puede tener varios eventos a su nombre en el mismo día o en el resto del calendario, no así con los recursos, que no pueden superponerse en una misma fecha.

3. Diagramas de clase

En el diagrama de clase, muestro como todos los mensajes del programa se comunican entre las entidades: **Calendario, Evento y las subclases de la clase Invitado: Recurso y Persona.**

Ademas, se muestran las clases herederas de la clase abstracta Invitado: **Recurso** y **Persona**, y la heredera de la clase Evento (**EventoSemanal)**(se cumple la relación "es un" entre cada clase heredera y la clase madre).

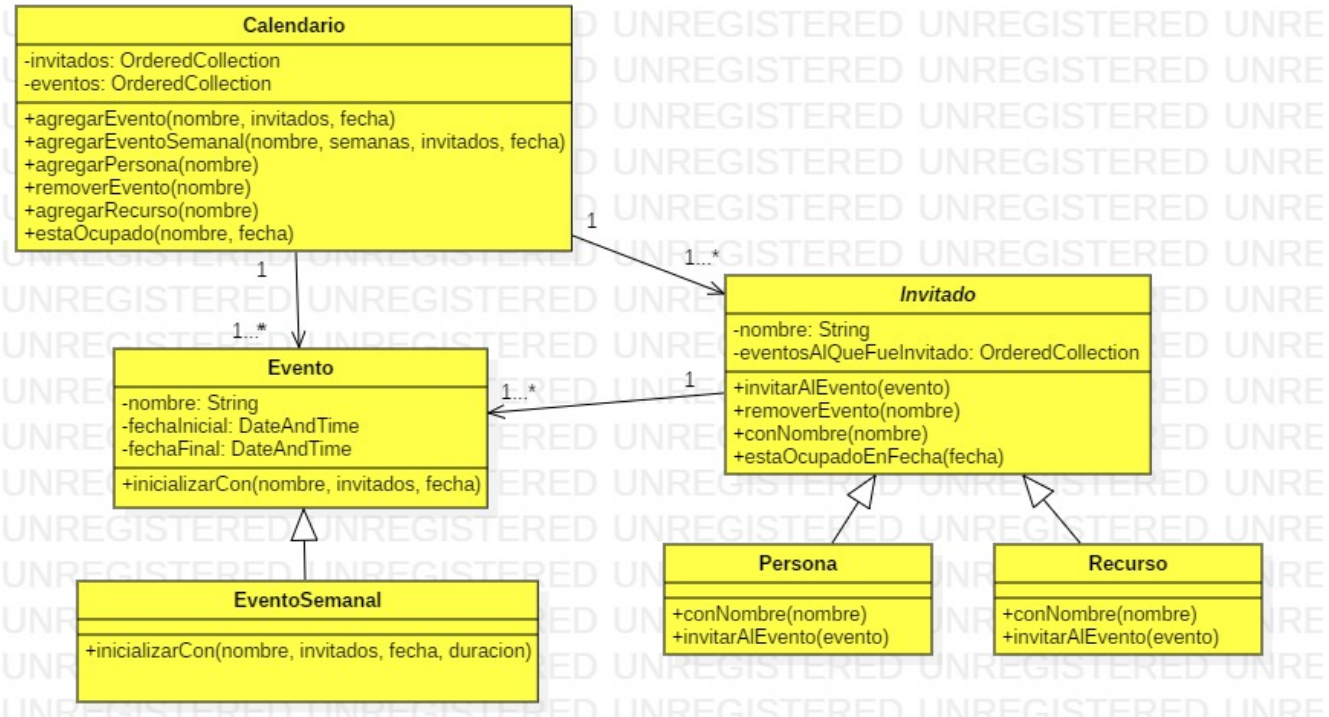


Figura 1: Diagrama del Calendario

4. Detalles de implementación

4.1. Calendario

Al empezar a modelar esta clase, no se me ocurría una forma sencilla de manejar a los recursos y personas sin romper el encapsulamiento. Por esto, se me ocurrió que Calendario tenga una colección de invitados (que incluye objetos del tipo Persona y Recurso), y una colección donde se guardan los eventos (sean simples o semanales).

4.2. Evento

Lo complicado de esta clase fue hacer que la clase madre y heredera puedan responder al mismo mensaje pero de manera distinta sin repetir código. Como cada evento tiene una duración distinta, lo que se me ocurrió fue implementar dos mensajes: **inicializarConNombre: conInvitados: enFecha:** e **inicializarConNombre: conInvitados: enFecha: conDuracion:.**

Otro detalle es que cada evento no conoce su listado de invitados: se delegó este comportamiento a cada invitado, así cada uno sabe a qué evento fue invitado.

4.3. Invitado

Esta clase abstracta implementa los mensajes que luego heredará cada clase hija: **Recurso y Persona.** Hay mensajes con el mismo nombre pero que cada subclase lo implementa necesariamente de distinta forma a la otra, por lo que en la clase abstracta se delega el comportamiento de ese mensaje a cada subclase; como así también hay mensajes que cada subclase responde de la misma manera, entonces se implementaron en la clase Abstracta y cada subclase puede utilizarlo mediante herencia.
Otro detalle que me parece importante aclarar, es que me pareció correcto que cada invitado sepa a qué eventos fué invitado, y no al revés, ya que al querer chequear si un invitado esta ocupado en un evento con cierta fecha, basta con que cada invitado sepa responder a ese mensaje y no tener que andar recorriendo cada evento, cada invitado de ese evento e ir preguntando sus fechas.

5. Excepciones

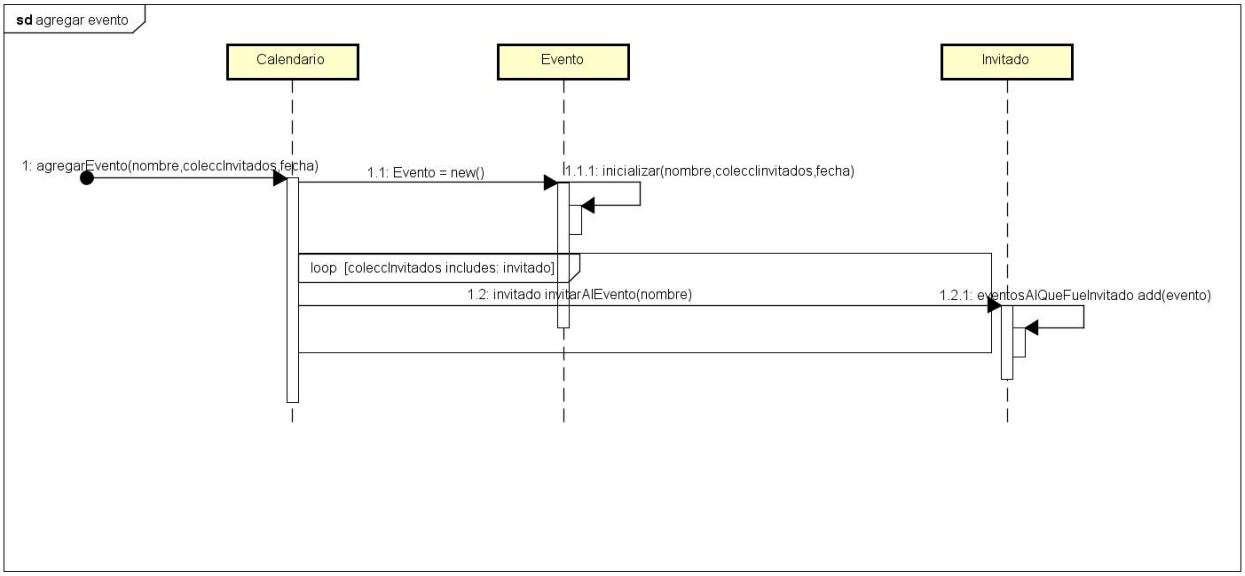
PersonaSinNombreError Al inicializar una instancia de la clase Persona, si el parámetro recibido por el método conNombre: es un string vacío, se lanza esta excepción.
RecursoSinNombreError Al inicializar una instancia de la clase Recurso, si el parámetro recibido por el método conNombre: es un string vacío, se lanza esta excepción.
RecursoOcupadoError Esta excepción es lanzada cuando se quiere agregar un evento al calendario con un recurso que ya esta siendo utilizado por otro evento en esa fecha.
EventoSinFechaError Se lanza cuando se quiere agregar un evento cuya fecha es desconocida, mediante el mensaje que inicializa una instancia de Evento o Evento semanal , el cual es invocado al momento de agregar un evento al calendario.
EventoSinNombreError Se lanza cuando se quiere agregar un evento cuyo nombre es vacío, mediante el mensaje que inicializa una instancia de Evento o Evento semanal , el cual es invocado al momento de agregar un evento al calendario.
EventoSinInvitadosError Se lanza cuando se quiere agregar un evento cuyos invitados estan representados por una colección vacía, mediante el mensaje que inicializa una instancia de Evento o Evento semanal , el cual es invocado al momento de agregar un evento al calendario.
EventoInexistenteError Se lanza si, al momento de querer remover un evento con cierto nombre del calendario, el evento no existe en él.

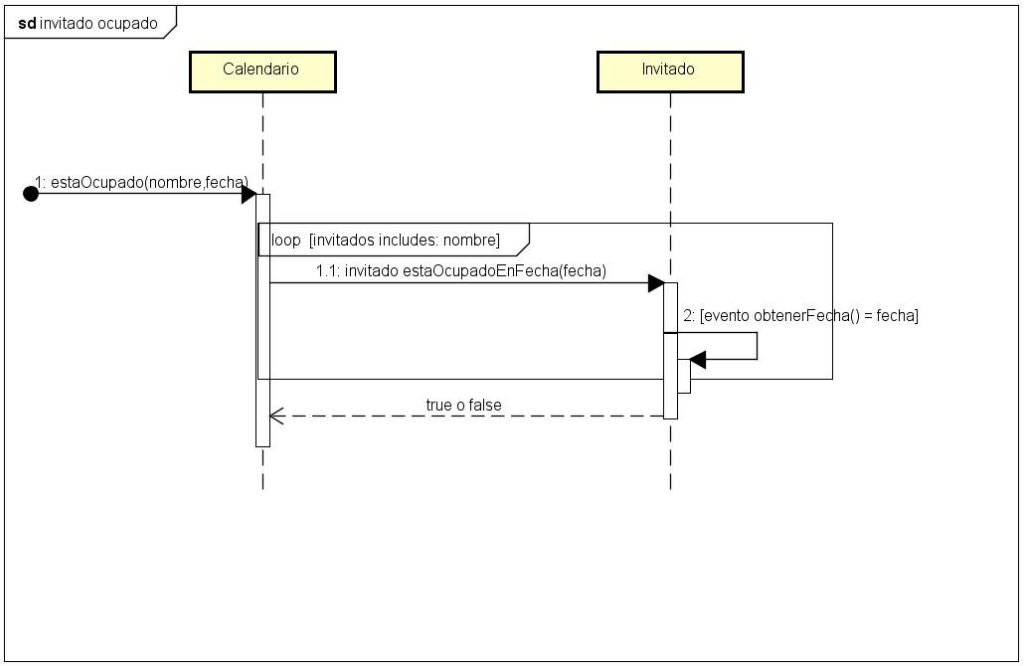
6. Diagramas de secuencia

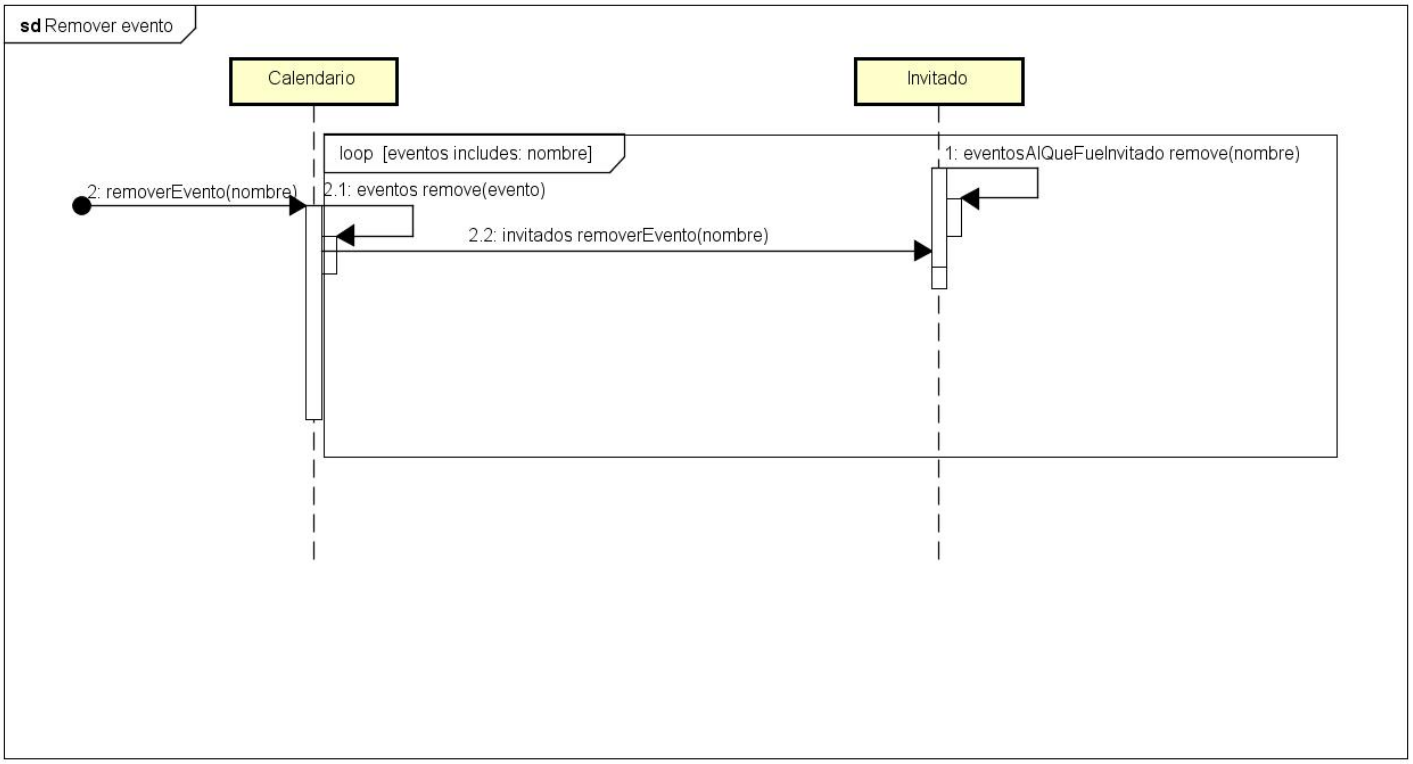
El primer diagrama de secuencia representa el flujo del programa al querer agregar un evento simple o semanal al calendario.

El segundo diagrama representa el flujo del programa al querer chequear si un invitado (recurso o persona) está ocupado en determinada fecha.

El tercer diagrama representa el flujo del programa al querer remover un evento con cierto nombre del calendario.






sd Remover evento
Calendario
Invitado
loop [eventos includes: nombre]
1: eventosAlQueFueInvitado remove(nombre)
2: removerEvento(nombre)
2.1: eventos remove(evento)
2.2: invitados removerEvento(nombre)