```
class Propuesta {
	const property interaccion = 0
	const property conocimiento = 0
	method +(otraPropuesta) = new Propuesta(
		interaccion = self.interaccion() + otraPropuesta.interaccion(),
		conocimiento = self.conocimiento() + otraPropuesta.conocimiento()
	)
	method *(multiplicador) = new Propuesta(
		interaccion = self.interaccion() * multiplicador,
		conocimiento = self.conocimiento() * multiplicador
	)
}

class Evento {
	method felicidadQueLeDariasA(grupo) = grupo.felicidadCon( self.propuestaPara(grupo) )

	method propuestaPara(grupo)
}

class Museo inherits Evento {
	var property tenesDinosaurios = false

	override method propuestaPara(grupo) =
		(self.propuestaBasica() + self.adicionalSiEsChico(grupo)) * self.factorSiHayDinosaurios()

	method propuestaBasica() = new Propuesta(interaccion = 5, conocimiento = 10)

	method adicionalSiEsChico(grupo) = if (grupo.sosChico()) new Propuesta(conocimiento = 5) else new Propuesta()

	// Si se delegaba a otros objetos que digan "1" o "2" según el caso, estaba OK. Pero así era suficientemente bueno:
	method factorSiHayDinosaurios() = if (self.tenesDinosaurios()) 2 else 1
}

// Una opción válida era componer con un objeto género y que éste sepa el valor a multiplicar. Peeero:
// Si se hacía eso, la referencia debía ser constante. Ya que no tiene sentido que un recital cambie de género.
```

```
class Recital inherits Evento {
	override method propuestaPara(grupo) = new Propuesta(interaccion = 10)
}

// Deben ser clases, ya que de lo contrario una persona podría asistir solamente a 1 recital, y no es la idea.
class RecitalDeRock inherits Recital {
	override method propuestaPara(grupo) = super(grupo) * 3
}

class RecitalDeReggaeton inherits Recital {
	override method propuestaPara(grupo) = super(grupo) * (-1)
}

class Reunion inherits Evento {
	override method propuestaPara(grupo) = new Propuesta(
		interaccion = 20,
		conocimiento = 7 * grupo.cantidadDePersonasQuePuedenAprender()
	)
}

class Maraton inherits Evento {
	const property eventos = #{}
	override method propuestaPara(grupo) = self.eventos().sum({ evento => evento.propuestaPara(grupo)})
}

class Viaje inherits Maraton {
	const property dias
	override method propuestaPara(grupo) = super(grupo) + new Propuesta(interaccion = 10 * self.dias())
}

class Grupo {
	const property integrantes = #{}

	method cantidadDeIntegrantes() = self.integrantes().size()
```

```
	method sosChico() = self.cantidadDeIntegrantes() <= 4

	// Está OK hacer count en vez de filter+size, pero preferimos esto último para poder saber quiénes son las personas
	// aprendices en un método separado ( personasQuePuedenAprender() )
	method cantidadDePersonasQuePuedenAprender() = self.personasQuePuedenAprender().size()

	// Había más alternativas. Lo importante era tener un filter, un any, y delegar lo de los temas a la persona
	method personasQuePuedenAprender() = self.integrantes().filter({aprendiz => self.alguienTePuedeEnseniar(aprendiz)})

	method alguienTePuedeEnseniar(aprendiz) = self.integrantes().any({educador => educador.lePodesEnseniarA(aprendiz)})

	method felicidadCon(propuesta) = self.integrantes().sum({ integrante => integrante.felicidadCon(propuesta) })

	method asistiA(evento) = self.integrantes().forEach({integrante => integrante.asistiA(evento)})
}

class Catalogo {
	const property eventos = #{}

	method mejorEventoPara(grupo) = self.eventos().max({ evento => evento.felicidadQueLeDariasA(grupo) })
}

class Persona {
	const property temas = #{}
	const property historial = #{}
	var property personalidad

	method sabes(tema) = self.temas().contains(tema)

	method noSabes(tema) = self.sabes(tema).negate()

	// Se podía pensar en diferencia de conjuntos pero como no está en la guía preferimos any
	method lePodesEnseniarA(otraPersona) = self.temas().any({tema => otraPersona.noSabes(tema)})

	method felicidadCon(propuesta) = self.personalidad().felicidadCon(propuesta)
```

```
	method yaFuisteA(evento) = self.historial().contains(evento)

	method asistiA(evento){
		if (self.yaFuisteA(evento)) {
			throw new ExcepcionAlAsistirAEvento("¡Ya asistió a este evento!")
		}
		self.historial().add(evento)
	}
}

object personalidadExtrovertida {
	method felicidadCon(propuesta) = if (self.teSatisface(propuesta)) propuesta.interaccion() * 2 else 0

	method teSatisface(propuesta) = propuesta.interaccion() > 5
}

object personalidadIntrovertida {
	method felicidadCon(propuesta) = propuesta.interaccion().min( propuesta.conocimiento() )
}

class ExcepcionAlAsistirAEvento inherits Exception {}
```

1) Nueva Maratón con Museo y Recital de Rock:

```
>>> new Maraton(eventos = #{new Museo(tenesDinosaurios = true), new RecitalDeRock()})
un/a  Maraton[eventos=#{un/a  Museo[tenesDinosaurios=true], un/a  RecitalDeRock[]}]
```

2) Introvertido con una propuesta, cambia a extrovertido, y reacciona a la misma propuesta

```
>>> const unaPropuesta = new Propuesta(interaccion = 10, conocimiento = 5)
>>> const shy = new Persona(personalidad = personalidadIntrovertida)
>>> shy.felicidadCon(unaPropuesta)
5
>>> shy.personalidad(personalidadExtrovertida)
>>> shy.felicidadCon(unaPropuesta)
```

3) Mejor evento para cierto grupo

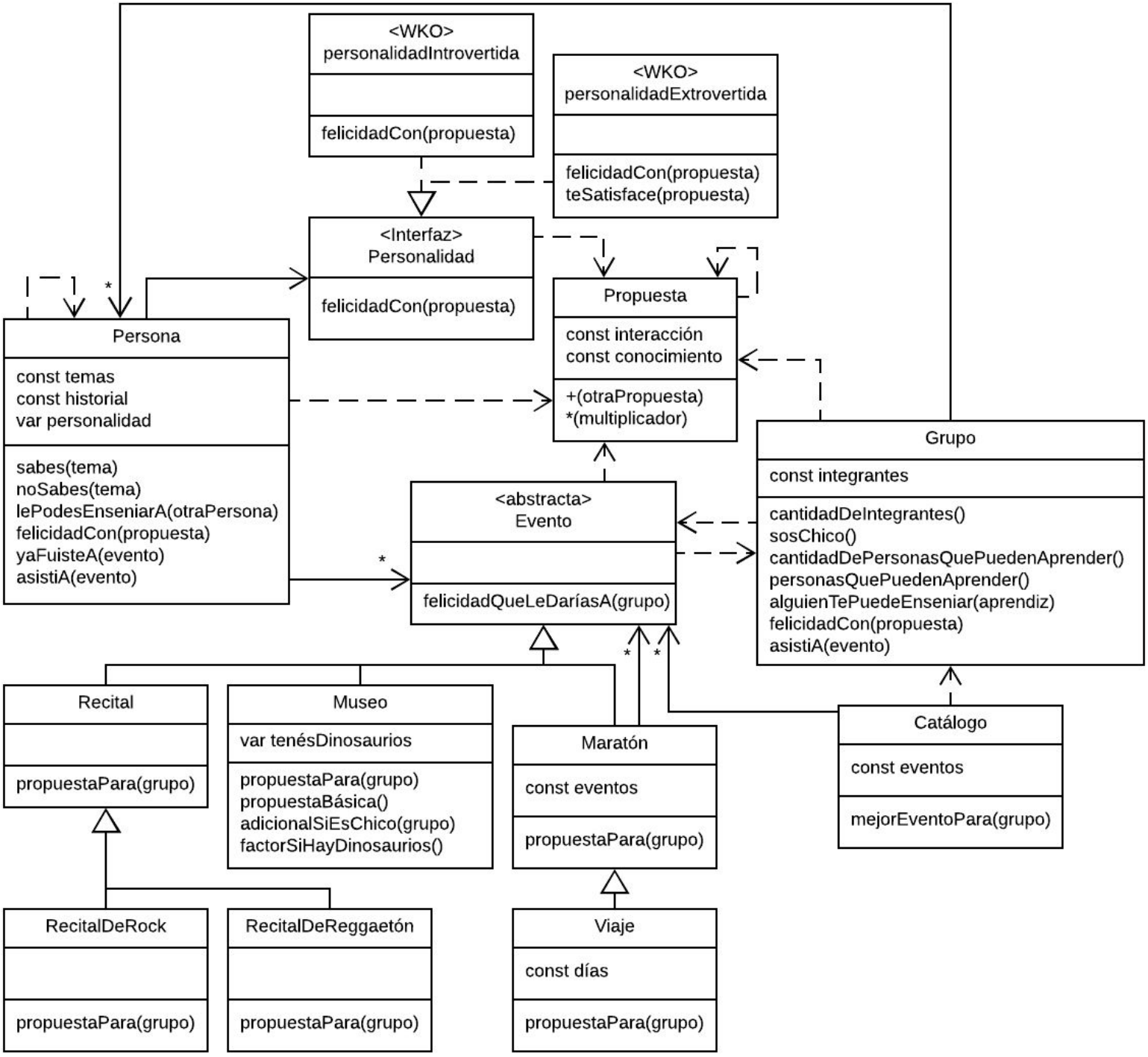
```
>>> const ciertoCatalogo = new Catalogo(eventos = #{new Museo(tenesDinosaurios = true), new RecitalDeRock()})
>>> const ciertoGrupo = new Grupo(integrantes = #{new Persona(personalidad = personalidadIntrovertida), new Persona(personalidad
= personalidadExtrovertida)})
>>> ciertoCatalogo.mejorEventoPara(ciertoGrupo)
un/a  RecitalDeRock[]
```

4) Pruebas sobre asistencia al evento

```
describe "Asistencia a eventos" {
	const shy = new Persona(personalidad = personalidadIntrovertida)
	const recitalDeAcdc = new RecitalDeRock()

	test "Si alguien no asistió a eventos, tranquilamente puede asistir a uno, y queda en su historial" {
		assert.notThat(shy.yaFuisteA(recitalDeAcdc))
		shy.asistiA(recitalDeAcdc)
		assert.that(shy.yaFuisteA(recitalDeAcdc))
	}

	test "Si alguien ya asistió a cierto evento, no puede volver a asistir al mismo (se lanza excepción)" {
		shy.asistiA(recitalDeAcdc)
		assert.throwsExceptionWithType(new ExcepcionAlAsistirAEvento(), { shy.asistiA(recitalDeAcdc)})
	}

	// Una alternativa era tener un solo .that y usar all dentro, pero esto era suficiente
	// Otra alternativa es que el grupo sepa decirnos si todos fueron al evento.
	// Pero lo importante es decirle que asista al evento, y ver de forma sencilla si sus miembros tienen el historial OK
	test "Si un grupo asiste a un evento, todos sus integrantes lo registran en su historial"{
		const mrBig = new Persona(personalidad = personalidadExtrovertida)
		const grupito = new Grupo(integrantes = #{shy, mrBig})
		grupito.asistiA(recitalDeAcdc)
		assert.that(shy.yaFuisteA(recitalDeAcdc))
		assert.that(mrBig.yaFuisteA(recitalDeAcdc))
```

```
        }
}
```

5) Diagrama de clases y WKO’s



6) Se usaron tanto herencia como composición.

Herencia: Todos los eventos heredan de la clase Evento (o de otros eventos). Convenía más con herencia que con composición, porque queda más simple (para codificar y para instanciar las entidades)60 y refleja bien que un evento no puede mutar y pasar a ser otra cosa que no tenga nada que ver con lo definido originalmente.

Composición: Las personas se componen con cierta personalidad. Encaja muy bien porque deben poder cambiar de personalidad dinámicamente, y con herencia no se podría.

7) Los nuevos eventos deben:

- Heredar de Evento así saben la felicidad que le darían al grupo.
- Definir qué propuesta le harían a cierto grupo.

El concepto clave para esto es el **polimorfismo**. Esto se ve en el método “mejorEventoPara(grupo)” del catálogo. El catálogo necesita poder preguntarle a cualquier evento **qué felicidad** le da a cierto grupo, y para ello, internamente, cada evento hará una propuesta personalizada a su manera. Sin embargo, el catálogo seguirá sin saber con cuál evento está hablando (los trata a todos igual, como si tuviesen la “misma forma”).