FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1 — Smalltalk

[7507/9502] Algoritmos y Programación III
Curso 1
Primer cuatrimestre de 2023

Alumno:	Del Carpio Sanchez, Alejandro
Número de padrón:	99655
Email:	adelcarpio@fi.uba.ar

Índice

1. Introducción

El presente informe reúne 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 un sistema de viajes en Pharo utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso.

2. Supuestos

El trabajo se realizó con los siguientes supuestos que fueron surgiendo a lo largo de la realización del trabajo ya que estos casos no están contemplados en las especificaciones entregadas.

- Los países añadidos como Mercosur son representativos ya que debido a conflictos políticos no se puede saber con certeza cuales pertenecen o si éste llegue a dejar de existir, pero como fin didáctico para la **realización** del trabajo supongo que está correcto
- El usuario puede registrarse con nombre y destino de viaje mas no puede modificarlo.
- No se puede ingresar un kilómetro cero
- Los kilómetros se supone enteros y no flotantes
- Los destinos de viaje se suponen conocidos por el usuario, es decir, la validación del país o ciudad inexistentes no están contemplados.
- Los destinos de viaje se suponen coherentes es decir el usuario sabe que la ciudad queda en ese país no se admite errores geográficos.
- Puede haber más de dos usuarios cada uno con su destino de viaje
- Supongo que el mismo usuario no intentará registrarse más de una vez, esto incluye el mismo destino de viaje, kilómetros, nacionalidad y nombre.
- Las ciudades del país se suponen que se registran sin tilde esto para evitar dificultades con distinto sistema operativo y el mismo Smalltalk.
- Se supone usuarios distintos al consultar un viaje es decir de distinto nombre
- Se supone que la agencia de viajes es nacional, es decir de Argentina.

3. Diagramas de clase

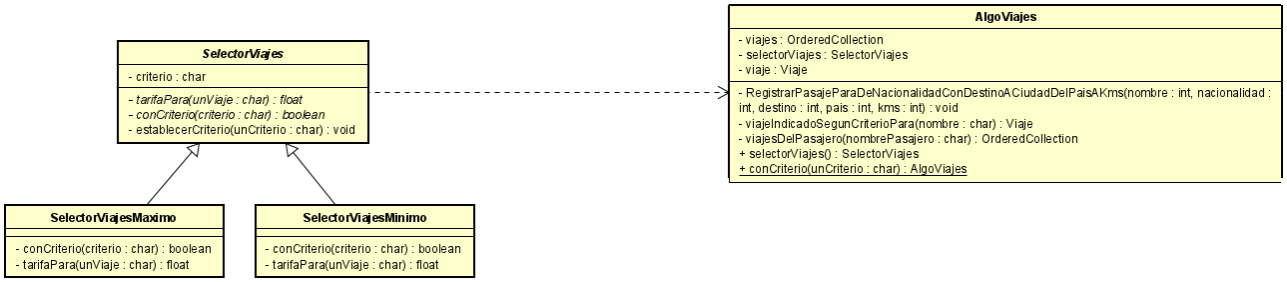

Figura 1: Dependencia de la clase AlgoViajes con SelectorViajes

Este recorte muestra el diagrama de clases de los métodos más importantes de AlgoViajes y SelectorViajes donde se muestra la dependencia de una a otra esto porque AlgoViajes crea un objeto de SelectorViajes (lo instancia) y lo devuelve como respuesta en uno de sus métodos.

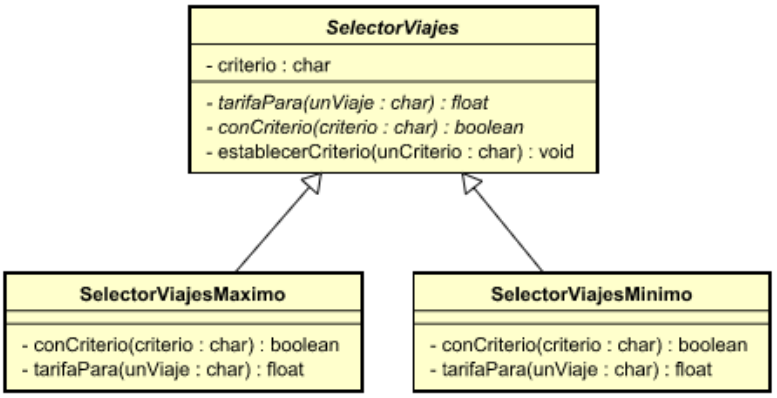

Figura 2: Diagrama de clases de SelectorViajes

Este recorte muestra la sección de SelectorViajes en ella se muestra la herencia con dos clases hijas SelectorViajesMaximo y SelectorViajesMinimo donde cada una responde al mensaje del método abstracto TarifaPara: este retorna la tarifa.

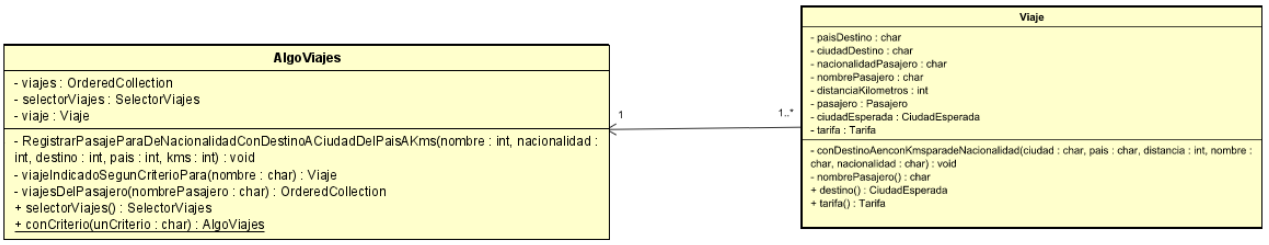

Figura 3: Asociación de Viaje con AlgoViaje

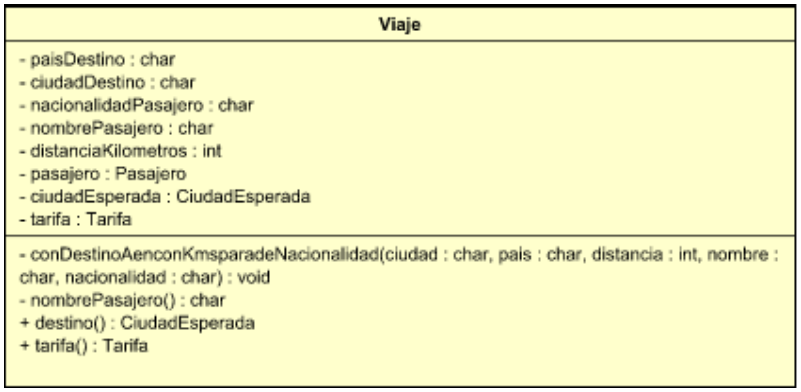

Figura 4: Clase Viaje

Tanto la Figura 4 como Figura 3 se trata de mostrar la Clase Viaje. En Figura 3 se muestra la asociación de AlgoViajes con Viaje porque de uno de sus métodos proporciona servicios a la clase además se puede observar su multiplicidad que se pensó del siguiente modo: AlgoViajes (o Agencia de Viajes) puede tener uno o muchos viajes .

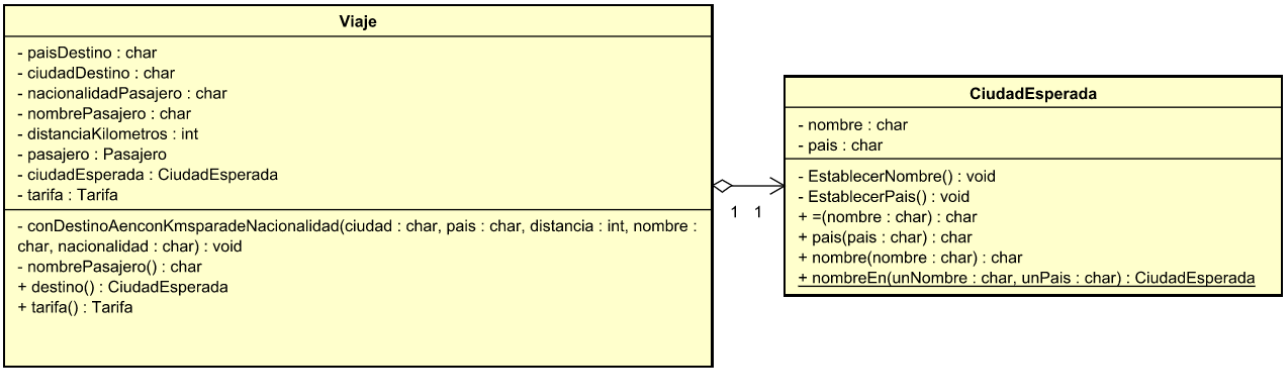

Figura 5:Viaje se compone de Ciudad Esperada y su multiplicidad es de uno a uno

CiudadEsperada fue proporcionada por la cátedra y en ella se puede notar una clase estática ya que esta está en Class Side se representó de esa forma por una presentación realizada en clase. En la relación se pensó en una “agregación navegable” o en inglés *aggregation to navigable association* este tipo de relación tiene un por lado lo de agregación junto a su multiplicidad nos dice: Un Viaje tiene una CiudadEsperada ahora por la flecha indica la dirección de la navegabilidad es decir que se puede acceder a la clase desde donde proviene.

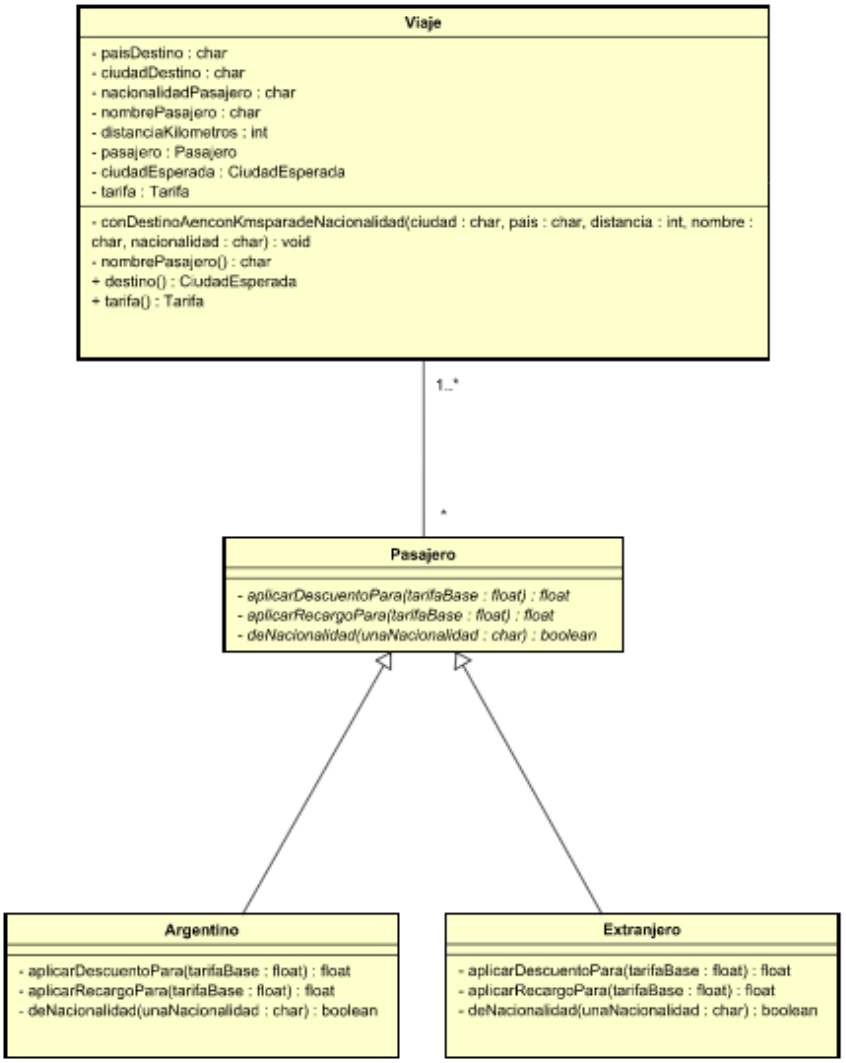

Figura 6 :Viaje tiene una asociación bidireccional con la clase Pasajero

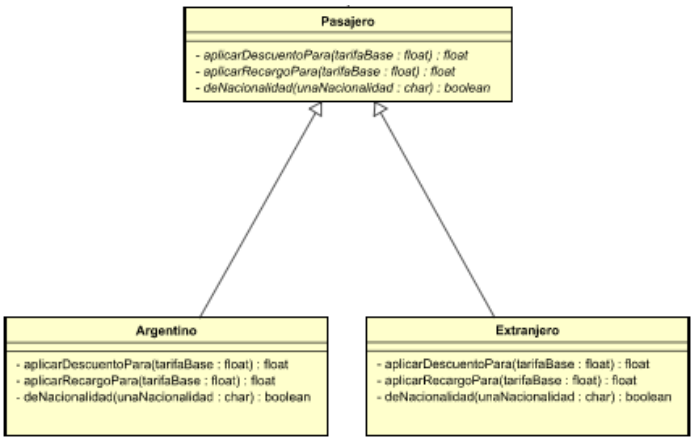

Figura 7: Clase Pasajero, Argentino y Extranjero como clases hijas

Tanto en la figura 6 como la figura 7 muestra la clase Pasajero lo primero a notar en la relación bidireccional que se pensó con la clase Viaje esto por el siguiente razonamiento: uno o más viajes puede tener muchos pasajeros y muchos pasajeros pueden tener uno o muchos viajes. En la figura 7 muestra la herencia que se aplicó a esa clase aquella tiene dos clases hijas Argentino y Extranjero. Un dato no menor los métodos abstractos donde dicha consecuencia vuelve a la clase en abstracta.

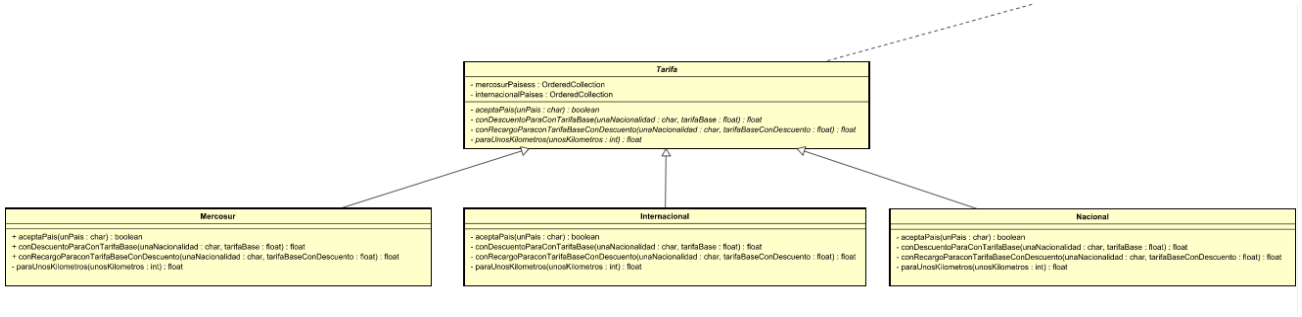

Figura 7: Clase Tarifa y sus hijas

la Figura 7 si bien no se nota con claridad porque se intentó que se notará sus métodos. Se dejó a propósito la línea punteada ya que está asociada con la clase Viajes. En esta clase se pensó en una herencia donde sus clases hijas son Mercosur, Internacional y Nacional.

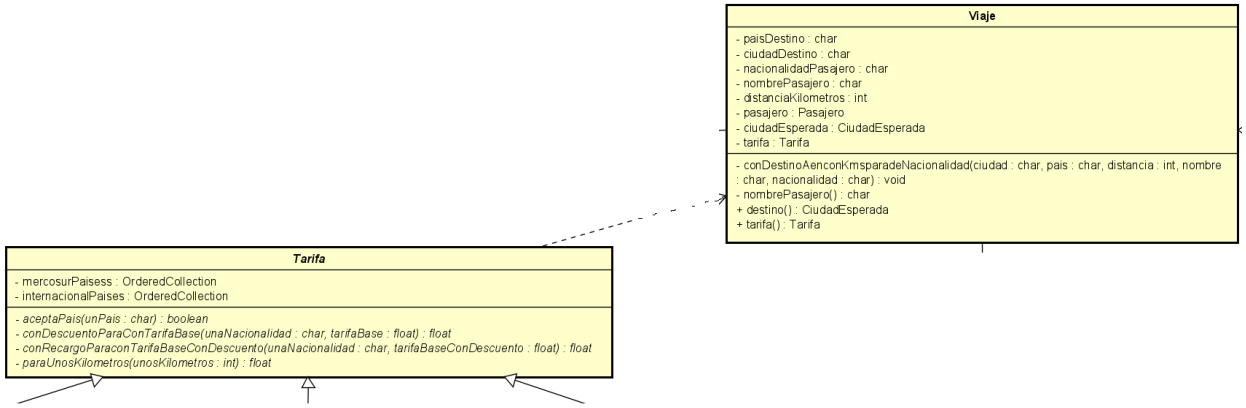

Figura 8: relación entre Tarifa y Viaje.

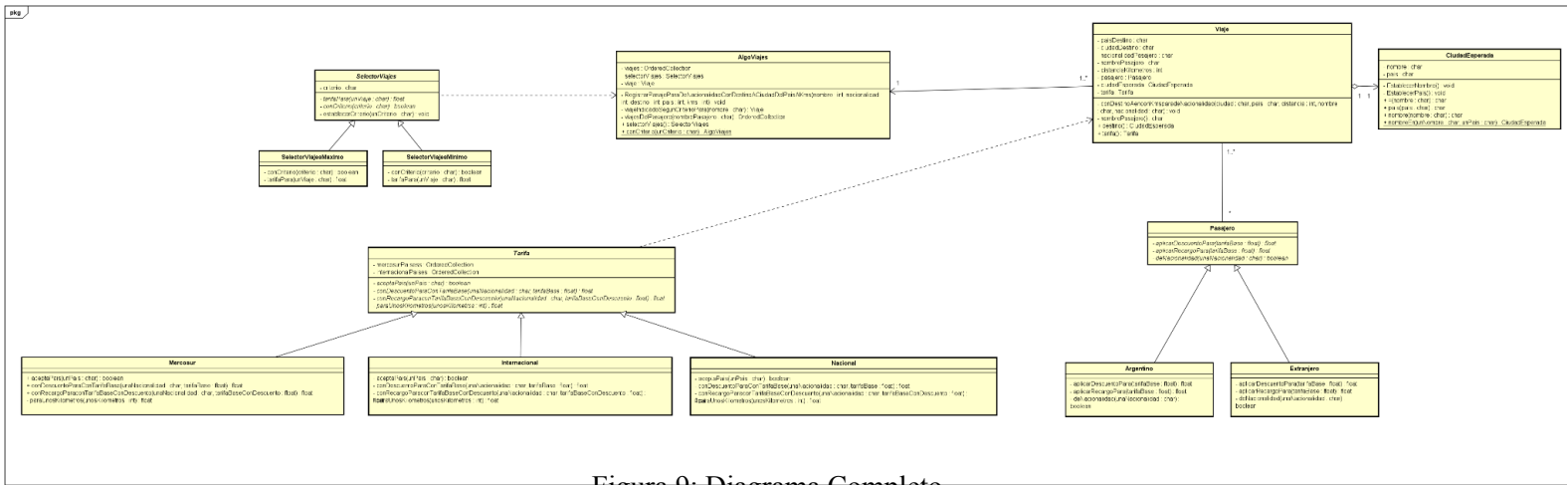

Figura 9: Diagrama Completo

Se intentó mostrar el diagrama completo pero no puedo achicarlo más porque sino los métodos no se entienden le dejo un link donde puede verlo https://photos.app.goo.gl/1BXBhJ2x2aasa1er5 si usted lo considera sino en todo caso tome las medidas correspondientes.+.

4. Detalles de implementación

4.1. AlgoViajes

Esta clase es la responsable general del trabajo, a través de ella se deben de realizar las diferentes opciones disponibles. Internamente almacena como datos una colección de tipo viajes. A continuación se detallaran algunos métodos importantes.

Metodo registrarPasajePara: unNombre deNacionalidad: unaNacionalidad conDestinoACiudad: unaCiudad delPais:

Este método registra un nuevo viaje y crea un objeto de tipo Viaje para luego ser añadido en un *orderedCollection* con el fin de obtener varios viajes y así mismo varias tarifas, pasajero y destinos del cual serán atributos de la clase Viaje.

Metodo viajeIndicadoSegunCriterioPara:

Este método invoca a un método de clase selector viajes con el fin de obtener la tarifa del pasajero. Se puede notar que se invoca TarifaPara: y se pasa viajesDelPasajero: de la clase AlgoViajes esto me parece más adecuado ya que viajes en un atributo de esa misma clase por tanto la responsabilidad de hacer tareas a ese *orderedCollection* es propia de la misma a pesar de que el lenguaje me permite hacerlo en SelectorViajes.

Metodo viajesDelPasajero:

Este método decidí no moverlo a otra clase porque *viajes* es un atributo de la clase e ir pasándolo entre clases no sería legible además de que en esa clase se inicializa así que en cierto modo toma la responsabilidad de dicho atributo. Para detectar el viaje indicado se usó un select del cual devuelve un orderedCollection de uno o más objetos Viaje el cual permitirá luego obtener sus atributos.

Metodo selectorViajes:

Este método se usa más que todo para inicializar el objeto Selector Viajes con el fin de poder inicializar el criterio (máximo o mínimo) del viaje en la clase *SelectorViajes.*

Metodo ConCriterio: unCriterio

Este método inicializa el criterio en la clase *SelectorViajes* y crea una nueva instancia de AlgoViajes.

4.2. Pasajero

Esta clase tiene como responsabilidad obtener el descuento, recargo de la tarifa del viaje respecto a la nacionalidad del pasajero del cual cuenta con dos clases hijas Argentino y Extranjero y tres métodos abstractos que realizarán las operaciones que correspondan se detallarán las clases más importantes.

Metodo aplicarDescuentoPara:

Este método abstracto recibe una tarifa base que es resultado del cálculo del viaje dependiendo si es nacional, Mercosur o Internacional más adelante se explicará con mayor detalle y retorna el resultado correspondiente respecto a la nacionalidad del pasajero Argentino o Extranjero.

Muestro el método implementado en la clase Argentino.

```
aplicarDescuentoPara: tarifaBase
	(tarifaBase <= 0) ifTrue:[LaTarifaBaseNoPuedeSerNegativoNiCero new signal].
	^(tarifaBase sqrt )
```

Metodo aplicarRecargoPara:

Este método abstracto recibe una tarifa base con el descuento respectivo calculado en el método aplicarDescuentoPara: y retorna el recargo correspondiente dependiendo si es Argentino o Extranjero.

Muestro el método implementado en la clase Argentino

```
aplicarRecargoPara: tarifaBase
	(tarifaBase <= 0) ifTrue:[LaTarifaBaseNoPuedeSerNegativoNiCero new signal].
	^(tarifaBase * 1.2)
```

Metodo deNacionalidad:

Este método abstracto tiene la responsabilidad de retorna un booleano dependiendo si es Argentino o extranjero para hacer esta operación recibe la nacionalidad.

Metodo descuentoCon:con:

Inicializar la nacionalidad nuevamente esto funciona como una ste metodo lo hice con el objetivo de que al llamar el método aplicarRecargoPara: o aplicarDescuentoPara: no tenga que inicializar el metodo desde otra clase ya que esto violaría la ley de demeter. Desde mi punto de vista esto reduce la dependencia entre los objetos y hace que el código sea más fácil de mantener y modificar en el futuro.

Sin este método

```
conRecargoPara: unaNacionalidad con: tarifaBaseConDescuento
	(tarifaBaseConDescuento <= 0) ifTrue:[LaTarifaBaseConDescuentoNoPuedeSerNegativoNiCero new
signal].
	^ (pasajero seleccionarTipoPara: unaNacionalidad) descuentoCon: tarifaBaseConDescuento)
```

Con Este Método

```
conRecargoPara: unaNacionalidad con: tarifaBaseConDescuento
	(tarifaBaseConDescuento <= 0) ifTrue:[LaTarifaBaseConDescuentoNoPuedeSerNegativoNiCero new signal].
	^ (pasajero descuentoCon: unaNacionalidad con: tarifaBaseConDescuento)
```

Metodo seleccionarTipoPara:

En este método, se crea una colección (en este caso, un array) que contiene instancias de ambas clases hijas de Pasajero. Luego, se utiliza el método "detect:" para buscar dentro de esta colección el primer objeto que cumple con la condición especificada en el bloque. En este caso, la condición es que el objeto debe tener la nacionalidad proporcionada.
Si se encuentra un objeto que cumple con esta condición, se devuelve a la instancia correspondiente caso contrario ‘nil’ que se omite por las condiciones proporcionadas en los métodos más adelante mostraré uno parecido del cual sí me pareció necesario ya que si pensamos esto como algo más profesional necesaria una base de datos o una tabla donde alojar las nacionalidades para evitar eso lo deje de ese modo.
Luego de muchos intentos fallidos para cumplir el poliformismo y evitar eso “if” del cual hacían que si cumpliera una condición me devolviera el objeto llegue a esta solución que si es polimórfica ya que partiendo de su definición, *capacidad de los objetos de diferentes clases de ser tratados como si fueran del mismo tipo o clase*, en este caso, las clases "Argentino" y "Extranjero" son subclases de una misma clase más general (llamado "Pasajero"), y ambos tienen un método "deNacionalidad:" que devuelve un booleano en función de la nacionalidad proporcionada. Al utilizar estas clases en una colección y luego buscar en esa colección utilizando un método polimórfico como "detect:", se permite que los objetos de diferentes clases sean tratados de la misma manera.

Además, si se agregara más subclases de la clase general (por ejemplo, "Joviano", "Marciano", etc.), el método "seleccionarTipoPara:" seguiría funcionando sin cambios, lo que demuestra la extensibilidad y flexibilidad del código.

```
seleccionarTipoPara: unaNacionalidad
	| nacionalidades pasajero|
	nacionalidades := { Argentino new. Extranjero new}.
    pasajero := nacionalidades detect: [ :f | f de Nacionalidad: unaNacionalidad].
	^pasajero
```

4.3. Tarifa

En este método se realiza el cálculo correspondiente a la tarifa del viaje para el pasajero para tal se implementó tres clases hijas Intencionalidad, Mercosur y Nacional que se detallaran con el fin de que cada una se haga responsable del cálculo dependiendo del país además de contar con métodos abstractos que hace a la clase abstracta en cierto punto. Detallaré algunos métodos del cual me parece importante explicar.

Metodo conDescuentoPara:con:

En este método se realiza el descuento correspondiente respecto a la nacionalidad para tal se pasa el mensaje conDescuentoPara:Con: a Pasajero y este devuelve el descuento correspondiente. En el caso de la clase internacional y Mercosur pude notar que no hay descuento así que solo devuelve la tarifa base.

Muestro un diagrama del razonamiento planteado

Figura 1. Nacionalidad > Destino > Descuento > Resultado

el código de este método en clase Nacional

```
conDescuentoPara: unaNacionalidad con: tarifaBase
	(tarifaBase <= 0) ifTrue:[LaTarifaBaseDeRecargoNoPuedeSerNegativoNiCero new signal].
	^ (pasajero conDescuentoPara: unaNacionalidad con: tarifaBase)
```

Metodo ConRecargoPara:Con

En este método si pude notar diferencias ya que dependiendo de su nacionalidad el cálculo era distinto. Recibe un descuento que como se explicó solo para la clase Nacional tendrá un resultado condicionado sino se le pasará la tarifa base.

Muestro un diagrama del razonamiento planteado

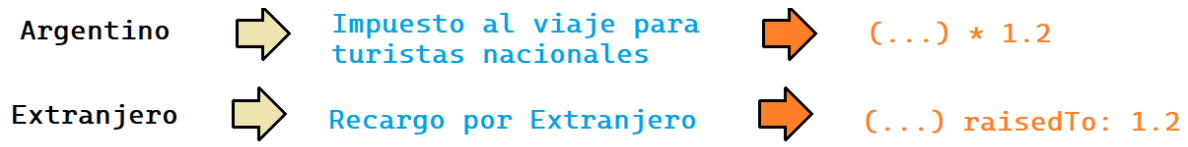

Figura 2. Nacionalidad > Recargo > Resultado

el código de este método en clase Nacional

```
conRecargoPara: unaNacionalidad con: tarifaBaseConDescuento
	(tarifaBaseConDescuento <= 0) ifTrue:[LaTarifaBaseConDescuentoNoPuedeSerNegativoNiCero new signal].
	^ (pasajero recargoCon: unaNacionalidad con: tarifaBaseConDescuento)
```

Metodo paraUnosKilometros

En este método se realiza el cálculo que denominó tarifa base ya que es aquel que se basa en la distancia de kilómetros por una tarifa respecto al viaje para luego pasar a su descuento y recargo.

Muestro un diagrama del razonamiento sobre la tarifa base

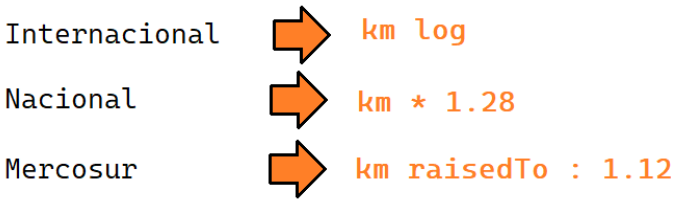

figura 3. Tipos de Destino > tarifa base

el código de este método en clase Internacional

```
paraUnosKilometros: kilometros
       (kilometros <= 0) ifTrue:[KmsNoPuedeSerNegativoNiCero new signal].
       ^ kilometros raisedTo: 1.12
```

Metodo para:aKms:paraPasajeroDeNacionalidad:

Si bien este método parece extenso pero traté de ser muy detallado con los nombres de cada variable lo pensé de este modo ya que en clase hablamos sobre dividir responsabilidades y de la coherencia de nuestro código.
Así mismo pensé que el mantenimiento o debuggear resultaría más sencillo ya que puedo saber con exactitud qué variable es la que posiblemente está teniendo un error.

el código es el siguiente

```
para: unPais aKms: unosKilometros paraPasajeroDeNacionalidad: unaNacionalidad
       |tarifaBase tarifaBaseConDescuento tarifaTotal|
       self seleccionarTipoDePara: unPais.
       tarifaBase := (tipoDeTarifa) paraUnosKilometros: unosKilometros.
       tarifaBaseConDescuento:= tipoDeTarifa conDescuentoPara: unaNacionalidad con: tarifaBase.
       tarifaTotal :=   tipoDeTarifa conRecargoPara: unaNacionalidad con: tarifaBaseConDescuento.

       ^tarifaTotal
```

Metodo seleccionarTipoDePara: unPais

Este método cuenta con la misma lógica del método SeleccionarTipoPara: .
El método "detect:" utilizado para encontrar el objeto adecuado en la colección devuelve el primer objeto que cumple con la condición dada en el bloque, o bien si no se encuentra ningún objeto en la colección que acepte el país proporcionado, el método "detect:" lanzará una excepción de error.

Para manejar esta situación, el método "seleccionarTipoDePara:" utiliza el método "ifNone:" en la llamada al método "detect:" para proporcionar una alternativa en caso de que no se encuentre ningún objeto en la colección que cumpla la condición. En este caso, la alternativa es crear y lanzar una excepción personalizada "NoExistePaisParaCalcularSuTarifa" utilizando el mensaje "new signal".

De esta manera, el uso de "ifNone:" permite manejar de manera adecuada la situación en la que no se encuentra ningún objeto en la colección que acepte el país proporcionado, lo que hace que el código sea más robusto y resistente a errores.

el código es el siguiente

```
seleccionarTipoDePara: unPais
       |tipoPais|
       tipoPais := { Nacional new. Mercosur new. Internacional new }.
    tipoDeTarifa := tipoPais detect: [ :f | f aceptaPais: unPais ] ifNone: [
NoExistePaisParaCalcularSuTarifa new signal].
```

4.4. Viaje

Esta clase recibe de AlgoViajes el registro del pasajero y se encarga de devolver el destino y la tarifa.Se detallan los mas importantes

Metodo conDestinoA:en:conKms:para:deNacionalidad:

El que inicia los valores es el constructor **.**

Metodo Destino

retorna un objeto CiudadEsperada con el destino de viaje

Metodo Tarifa

Este método le manda a un mensaje al método para:aKms:paraPasajeroDeNacionalidad: para luego retornar la tarifa del viaje.

4.5. SelectorViajes

En esta clase se retorna de la colección de viajes la tarifa respecto al criterio (máximo o mínimo) por eso mismo hereda dos clases hijas SelectorViajesMaximo, SelectorViajesMinimo. Se detallarán las clases más importantes.

Metodo tarifaPara:

Este método recibe una colección de viajes filtrado por el nombre del pasajero realizado por viajesDelPasajero: y retorna la tarifa.

Este código muestra el método en la clase SelectorViajesMinimo

```
tarifaPara: unViaje
	^(unViaje detectMin: [ :p | p tarifa ] )
```

5. Excepciones

Excepcion ElNombreDeLaCiudadNoPuedeSerVacio
Se pasa al método establecerNombre: de la clase CiudadEsperada una string vacío se espera una excepción

Excepción ElNombreDelPaisNoPuedeSerVacio
Se pasa al método establecerPais: de la clase CiudadEsperada una país vacío se espera una excepción

Excepción ElNombreDelPasajeroNoPuedeSerVacio
Se pasa al método conDestinoA:en:conKms: para:deNacionalidad de la clase Pasajero un nombre vacío se espera una excepción

Excepción KmsNoPuedeSerNegativoNiCero
Se pasa al método paraUnosKilometros: de la clase Tarifa unos kilómetros negativos se espera una excepción

Excepción LaNacionalidadNoPuedeSerVacia
Se pasa al método deNacionalidad: de la clase Pasajero una nacionalidad vacía se espera una excepción

Excepción LaTarifaBaseConDescuentoNoPuedeSerNegativoNiCero
Se pasa al método ConDescuento:Para: de la clase Tarifa una tarifa base negativa o cero se espera una excepción

Excepción LaTarifaBaseDeRecargoNoPuedeSerNegativoNiCero
Se pasa al método ConRecargo:Para: de la clase Tarifa una tarifa base negativa o cero se espera una excepción

Excepción LaTarifaBaseNoPuedeSerNegativoNiCero
Se pasa al método aplicarDescuentoPara: / aplicarRecargoPara: de la clase Pasajero una tarifa base negativa o cero se espera una excepción

Excepción NoExisteCriterioParaEseViaje
En la clase SelectorViajes no se encuentra la clase que cumpla con la condición del método conCriterio: por tanto "detect:" devuelve nil debería lanzar una excepción

Excepción NoExistePaisParaCalcularSuTarifa
En la clase Tarifa no se encuentra destino que cumpla con la condición aceptaPais: por tanto "detect:" devuelve nil debería lanzar una excepción

Excepcion NoSePuedeCalcularElDescuentoPorqueEsElMismoQueSuTarifa
Este error es bastante minucioso ya que se evalúa un límite al descuento que es cuando la tarifa base coincide con el valor a descontar esto en la clase Internacional por tanto la tarifa base devuelve cero debería lanzar una excepción

6. Diagramas de secuencia

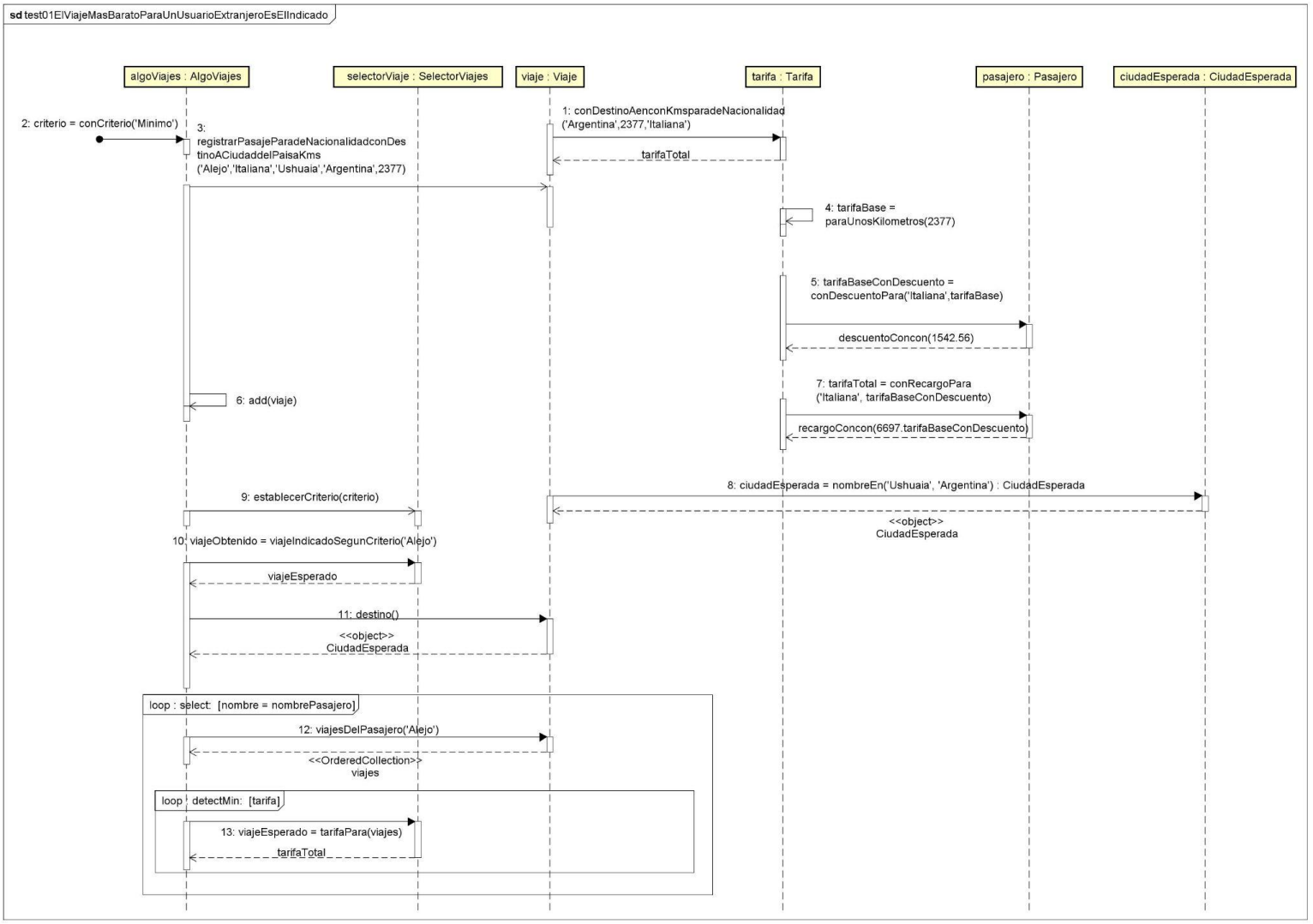

Figura 1: Diagrama de Secuencia del Test01

En el primer diagrama muestro las clases más importantes en la resolución del test01 para que el mismo no sea tan sobrecargado y la visualización sea adecuada. No coloco en el diagrama clases del propio pharo y algunas que podrían afectar la interpretación del resultado en un siguiente diagrama mostrare estos detalles que faltaron.

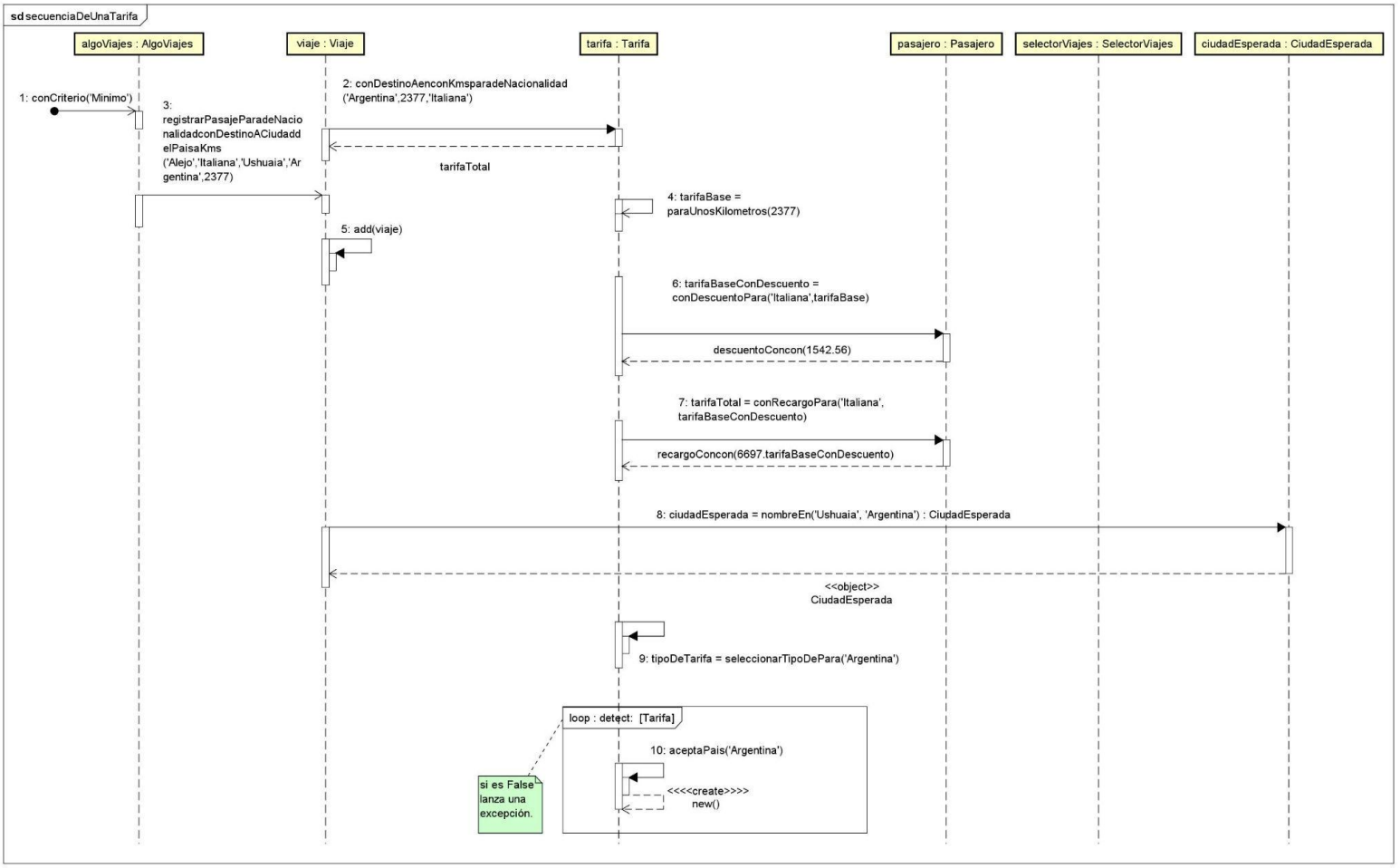

Figura 2. Secuencia de una Tarifa

En este diagrama muestro con detalle cómo se genera una tarifa y omito métodos en AlgoViajes del cual detalle en el anterior diagrama ya que mi objetivo es mostrar la secuencia de una tarifa así mismo se intentó representar el polimorfismo aplicado en la clase seleccionarTipoDePara:

7. Notas

Viendo mi código note mejoras como seleccionarTipoDePara: ya que aplicó la misma lógica en otra clases para esto se me ocurrió una interfaz pero la única limitación que encuentro es el tiempo de la entrega además del lenguaje que no cuenta con interfaces y tratar de representarlo es un poco costoso además hacerle muchos test para evitar mutantes. En el transcurso del desarrollo fuí haciéndome preguntas del cual no especifique en alguna clases/métodos así que decidí ponerle en esta sección.

1. ¿El método **seleccionarTipoPara:** cumple con la ley de demeter?

   Desde mi punto de vista si cumple ya que el método utiliza un objeto "tipoPais" que es una colección de objetos de diferentes subclases de "TipoDeTarifa" para seleccionar el tipo de tarifa correspondiente a un país dado. Sin embargo, el método no accede directamente a los objetos de la colección; en su lugar, utiliza el método "detect:" proporcionado por la colección para buscar un objeto que cumpla con la condición especificada.
   Por lo tanto, el método "seleccionarTipoDePara:" delega la responsabilidad de buscar el objeto adecuado a la colección, lo que hace que el código sea más mantenible y escalable.

2. ¿El método **para:aKms:paraPasajeroDeNacionalidad** es muy largo?
   Si es muy largo pero evitar el uso de variables que retorna cada método lo hace ilegible por otro lado podría evitar crear métodos para cada tarea pero aprendí que a través del principio divide y vencerás hago que un problema grande puede ser dividido en varios subproblemas más pequeños y manejables así pueda ser resuelto de manera independiente y luego integrado en una solución más grande.

3. ¿El método **para:aKms:paraPasajeroDeNacionalidad** que principio cumple?

Me seguí cuestionando por mi resolución y recordé lo que nos comentaron en las primeras clases de la materia un tema que no fue tomado con la importancia que se merece y es el Principio de responsabilidad única o SRP por sus siglas en inglés. Wikipedia menciona lo siguiente

- Una clase debería tener solo una razón para cambiar
- Cada responsabilidad es el eje del cambio
- Para contener la propagación del cambio, debemos separar las responsabilidades.
- Si una clase asume más de una responsabilidad, será más sensible al cambio.
- Si una clase asume más de una responsabilidad, las responsabilidades se acoplan.

En este caso, el método tiene como única responsabilidad calcular la tarifa para un viaje en función del país, los kilómetros y la nacionalidad del pasajero. El método delega el trabajo de seleccionar el tipo de tarifa y aplica los descuentos y recargos correspondientes a otras clases (por ejemplo, "seleccionarTipoDePara:") y utiliza. Al seguir el principio, el código es más fácil de entender, mantener y reutilizar en comparación con un método que realiza múltiples tareas no relacionadas.variables locales para almacenar los resultados intermedios.

4. ¿el método **conDestinoA:en:conKms:para:deNacionalidad:** está bien colocado en inst side?

Esto fue preguntado y respondido con poca claridad por el correcto así que justificó el porqué.
Tiene sentido colocarlos porque si lo coloco en class side luego para se comunique con inst es crear métodos como establecerNacionalidad.

Como comentarios finales Smalltalk me pareció interesante pero es un gran desafío poder desarrollar como es un lenguaje poco popular es mucho más difícil buscar ayuda y encontrar documentación de tercero aun más. Estuve investigando un poco más y a pesar de que es considerado como lenguaje “potente y elegante”tiene problemas de rendimiento haciéndolo lento y esto porque permite cambiar las estructuras del programa en tiempo de ejecución es decir que el sistema necesita realizar más verificaciones en tiempo de ejecución para garantizar que los cambios son válidos, lo que puede ralentizar la ejecución del programa.

⚠️ Importante

Si trae la imagen de casos de usos de la cátedra sobreescribe CiudadEsperada editada por mí así que dos de los test va a fallar. Le paso a detallar los métodos que pueden sufrir modificaciones en el proceso de importación de la imagen del proporcionado por la cátedra.

```
establecerPais: unPais
	(unPais isEmptyOrNil ) ifTrue:[ElNombreDelPaisNoPuedeSerVacio new signal].
	pais := unPais.
```

```
establecerNombre: unNombre
	(unNombre isEmptyOrNil )ifTrue: [ElNombreDeLaCiudadNoPuedeSerVacio new signal ].
	nombre := unNombre
```

Los test que pueden fallar debido a este conflicto.

```
test04ElNombreDeLaCiudadARegistrarEsVacio

	"arrange & should"

	self should: [(CiudadEsperada nombre: '' en: 'Peru')] raise:
ElNombreDeLaCiudadNoPuedeSerVacio.
```

```
test05ElNombreDelPaisARegistrarEsVacio

	"arrange & should"

	self should: [(CiudadEsperada nombre: 'Cuzco' en: '')] raise:
ElNombreDelPaisNoPuedeSerVacio.
```

Solución: importar los casos de uso proporcionados en el zip.