Universidad de Málaga

ETSI Informática

Modelado estructural y Dinámico de un sistema de coches

Modelado y Diseño del Software (2024-25)

Daniil Gumeniuk
Angel Bayon Pazos
Diego Sicre Cortizo
Pablo Ortega Serapio
Angel Nicolás Escaño López
Francisco Javier Jordá Garay
Janine Bernadeth Olegario Laguit

Grupo 1.1

Noviembre 2024

Índice

Índice de figuras

1 Introducción

El presente proyecto tiene como objetivo el modelado de un sistema de coches que puedan viajar entre ciudades y se sometan a revisiones utilizando la herramienta USE (UML-based Specification Environment), que permite la especificación y verificación formal de sistemas mediante el lenguaje OCL (Object Constraint Language). El sistema desarrollado gestiona la información relacionada con coches, ciudades, viajes, recorridos, revisiones, y talleres ya sean oficiales o no. A través de la implementación de este modelo, se puede asegurar el cumplimiento de reglas como cada cuanto tiempo a un coche le deben realizar una revisión o que no haya más de un taller oficial por ciudad.

El sistema está compuesto por varias clases fundamentales para el comportamiento estructural y dinámico de los coches. La nueva clase *Clock* discretiza el tiempo en el sistema. Su operación *tick()* avanza en el tiempo y los *tick()* asociados a cada *ActiveObject* representa la acción síncrona que cada subclase que herede realiza en cada llamada; esto implica que cada coche, el *ActiveObject* del modelo, implementará cambios en su estado actual. Además, el uso de restricciones e invariantes considerando estos nuevos elementos garantizan el comportamiento dinámico que se busca por la descripción proporcionada del sistema.

A lo largo de este documento, se presentarán las especificaciones detalladas de cada clase, junto con los atributos y las invariantes que aseguran la consistencia del sistema. También se describirá cómo se gestionan las relaciones entre las diferentes entidades del modelo, desde los recorridos que hay entre dos ciudades hasta saber si el coche esta en garantía o no.

Finalmente, se incluye un conjunto de pruebas definidas en el archivo `.soil` que demuestran el correcto funcionamiento del sistema, validando cada una de las restricciones especificadas en el modelo.

2 Modelado Estructural

2.1 Diagrama de Clases

En primer lugar veremos el diagrama de clase que hemos creado en VisualParadigm para la primera parte de la práctica, la cual pide el modelado conceptual de la estructura del sistema. Cuenta con una clase *Clock* de la cual hay una única instancia y las siguientes entidades con sus respectivas relaciones entre si:

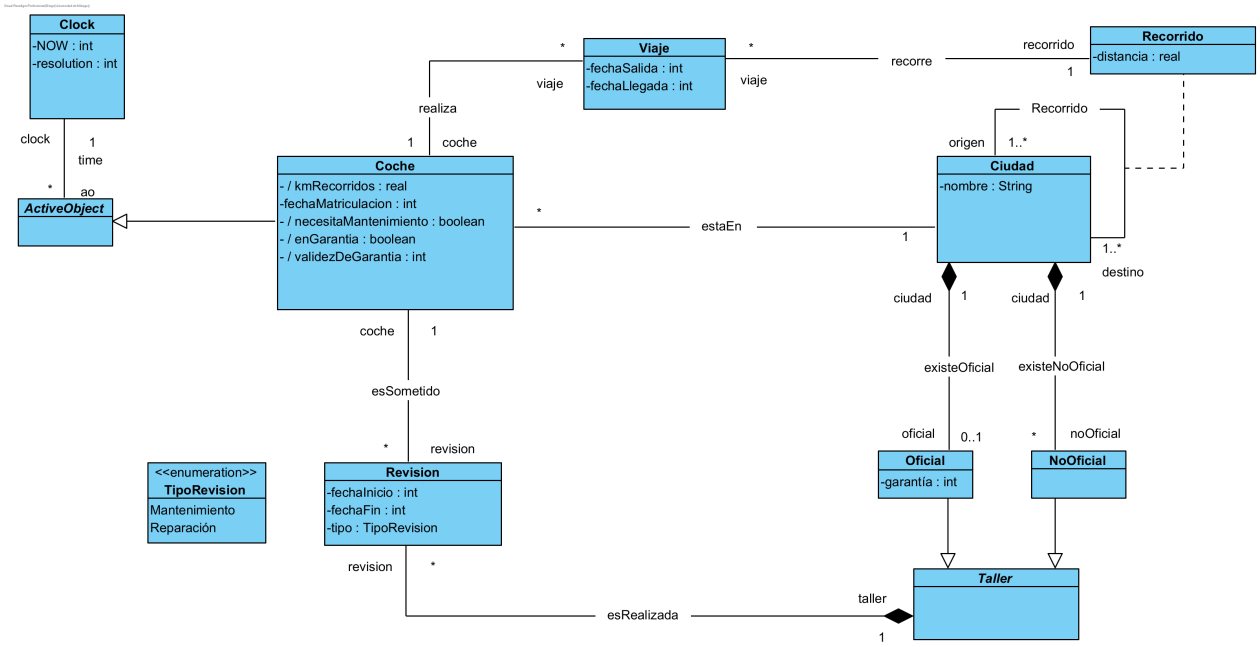

Figura 1: Diagrama del sistema de coches

2.2 Enumeraciones del Sistema

2.2.1 TipoRevision

Descripción: Distingue entre los tipos de revisión a los que un coche se puede someter.

- **Literales**:

  1. `Mantenimiento`: Revisión obligatoria que cada coche necesita después de 4 años desde que se matriculó o después de 1 año desde la última revisión de mantenimiento que tuvo algún coche.
  2. `Reparación`: Revisión necesaria para la reparación de alguna avería en el coche.

2.3 Clases del Sistema

2.3.1 Clock

Descripción: Representa el elemento que lleva el conteo del instante actual en el que se encuentre el sistema y el valor los saltos que cada tick avanza en el tiempo. Se definió que cada unidad de tiempo equivale a un día y un año se representa con 100 días.

- **Atributos**:
  - `NOW: Integer`: Contador del tiempo registrado. Representa el instante actual en el que se encuentre el sistema.
  - `resolution: Integer`: Valor de los saltos que hace el Clock en la llamada de cada tick.

2.3.2 ActiveObject

(Clase abstracta) **Descripción**: Representa la clase padre de cualquier entidad del sistema que necesite un paso del tiempo para cambiar su estado. En este sistema solo **Coche** es una especificación de ActiveObject.

2.3.3 Coche

(Subclase de ActiveObject) **Descripción**: Representa un coche en el sistema. Cada coche puede realizar viajes entre ciudades y se someten a revisiones cuando lo requieran.

- **Atributos**:
  - `kmRecorridos: Real (derive)`: Atributo derivado que registra todos los kilómetros que el coche ha viajado entre ciudades como la suma de *distancia* de cada *recorrido* entre un par de ciudades.
  - `fechaMatriculacion: Integer`: Marca el instante en el que el coche se ha registrado en el sistema.
  - `necesitaMantenimiento: Boolean (derive)`: Atributo derivado que indica que el coche necesita ir a revisión *Tipo: Mantenimiento* ya sea porque han pasado 4 años desde que se matriculó o pasó más de 1 año desde la última revisión de mantenimiento que tuvo.
  - `enGarantia: Boolean (derive)`: Atributo derivado que indica que el coche tiene una garantía activa ya sea porque no han pasado 4 años desde que se matriculó o la *validez de garantía* proporcionada por el *Taller Oficial* donde se revisó sigue vigente. Un coche en garantía puede necesitar alguna revisión si las condiciones lo dictan.
  - `validezDeGarantia: Integer (derive)`: Atributo derivado que indica los días por los que la garantía proporcionada por un *Taller Oficial* está activa. En caso que un coche realice una revisión *Tipo: Mantenimiento* en un *Taller Oficial*, este recibirá la garantía comenzará a contar a la par que el año de garantía que el coche recibe por revisión de mantenimiento.

2.3.4 Revision

Descripción: Representa la revisión que se le realizan a los coches en el sistema. Un coche puede realizar varias revisiones a lo largo del tiempo cuando esté obligado a hacerlo o lo necesite.

- **Atributos**:
  - `fechaInicio: Integer`: La fecha en la que se empezó la revisión del coche.
  - `fechaFin: Integer`: La fecha en la que se finalizó la revisión del coche.
  - `tipo: TipoRevisión`: Es el tipo de revisión que se le ha realizado al coche, que puede ser un mantenimiento o una reparación.

2.3.5 Taller

(Clase abstracta) **Descripción**: Representa un taller en el sistema y es la clase base para los diferentes tipos de taller. Las revisiones, independientemente del tipo, se realizan en un taller.

2.3.6 Oficial

(Subclase de `Taller`) **Descripción**: Hereda de la clase Taller. Al realizar una revisión en este tipo de taller, el coche obtiene una garantía. Solo existe un taller oficial por ciudad.

- **Atributos**:
  - `garantía: Integer`: Duración de la garantía que ofrece el taller oficial.

2.3.7 No_Oficial

(Subclase de `Taller`) **Descripción**: Hereda de la clase Taller. A diferencia del taller oficial, si se realiza un mantenimiento en ese taller, no se da una garantía al coche. Pueden existir varios talleres no oficiales en una ciudad.

2.3.8 Ciudad

(Clase abstracta) **Descripción**: Representa una ciudad en el sistema en el que un coche viaja hacia esta.

- **Atributos**:
  - `nombre: String`: Nombre de la ciudad.

2.3.9 Recorrido

(Clase de Asociación entre Ciudad y Viaje) **Descripción**: Representa el recorrido que se realiza durante un viaje para ir de una ciudad a otra en el sistema.

- **Atributos**:
  - `distancia: Double`: La distancia que hay entre la ciudad de origen y la de destino.

2.3.10 Viaje

Descripción: Representa los viajes en el sistema.

- **Atributos**:
  - `fechaSalida: Integer`: La fecha de salida de la ciudad de origen del coche.
  - `fechaLlegada: Integer`: La fecha de llegada a la ciudad de destino del coche.

2.4 Código USE

A continuación mostraremos todo el código desarrollado con el lenguaje USE, están divididos en varios sectores:

- **'Enumeraciones'**: Representa los distintos valores que un atributo puede tomar en un momento dado en sistema.
- **'Clases'**: Representa todas las entidades del sistema.

- **'Relaciones'**: Representa las interacciones existentes entre entidades.
- **'Invariantes'**: Representa todas las restricciones del sistema a manejar.

Todo el código suministrado ha sido realizado en VSCode y todas las secciones están separadas por sus respectivos títulos tabulados en mayúsculas.

Listing 1: Modelo de Sistema de Coches en USE

```
-- ================================================================
--                              MODELO
-- ================================================================

model P2_Toyota

-- ================================================================
--                           ENUMERATIONS
-- ================================================================

enum TipoRevision{Mantenimiento, Reparacion}


-- ================================================================
--                              CLASES
-- ================================================================

class Clock
attributes
    NOW : Integer
    resolution : Integer
end

abstract class ActiveObject
end

class Coche < ActiveObject
attributes
    kmRecorridos : Real derive :
        self.viaje.recorrido -> collect(dist | dist.distancia) -> sum()

    fechaMatriculacion : Integer

    necesitaMantenimiento : Boolean derive :
        if self.clock.NOW  - fechaMatriculacion < 400
            then
                false
            else
                let ultimaRevision : Revision = self.revision -> select(rev |
                    rev.tipo = TipoRevision::Mantenimiento) -> sortedBy(r | r.
                    fechaInicio) -> last() in
                if ultimaRevision <> null
                then
                    (self.clock.NOW - ultimaRevision.fechaFin) > 100
                else
```

```
                    true
                endif
        endif


    enGarantia : Boolean derive :

        ((self.clock.NOW  - fechaMatriculacion) < 400) or
        (validezDeGarantia > 0)

    validezDeGarantia : Integer derive :
        if (self.clock.NOW - fechaMatriculacion) < 400
            then
                400 - (self.clock.NOW  - fechaMatriculacion)
            else
                let revisionesOficiales : OrderedSet(Revision) = self.revision
                    -> select(rev | rev.taller.oclIsKindOf(Oficial)) ->
                    sortedBy(rev | rev.fechaInicio) in
                    if (self.clock.NOW - (revisionesOficiales -> last()).
                        fechaFin) < (revisionesOficiales -> last()).taller.
                        oclAsType(Oficial).garantia
                        then
                            (revisionesOficiales -> last()).taller.oclAsType(
                                Oficial).garantia - (self.clock.NOW - (
                                revisionesOficiales -> last()).fechaFin)
                        else
                            0
                    endif
        endif
end

class Viaje
attributes
    fechaSalida : Integer
    fechaLlegada : Integer

end

class Revision
attributes
    tipo : TipoRevision
    fechaInicio : Integer
    fechaFin : Integer
end

abstract class Taller
end

class Oficial < Taller
attributes
    garantia : Integer
end

class NoOficial < Taller
end
```

```
class Ciudad
attributes
    nombre : String
end


-- ========================================================================
--                          CLASES DE ASOCIACION
-- ========================================================================

associationclass Recorrido between
    Ciudad [1] role destino Ciudad [1] role origen
attributes
    distancia : Real
end

-- ========================================================================
--                            ASOCIACIONES
-- ========================================================================

association time between
Clock [1] role clock
ActiveObject [*] role ao
end

association realiza between
Coche [1] role coche
Viaje [*] role viaje
end

association recorre between
Viaje [*] role viaje
Recorrido [1] role recorrido
end

association estaEn between
Coche [*] role coche
Ciudad [1] role ciudad
end

association esSometido between
Coche [1] role coche
Revision [*] role revision
end

composition esRealizada between
Taller [1] role taller
Revision [*] role revision
end

composition existeOficial between
Ciudad [1] role ciudad
Oficial [0..1] role oficial
end
```

```
composition existeNoOficial between
Ciudad [1] role ciudad
NoOficial [*] role  NoOficial
end


-- ============================================================================
--                            INVARIANTES
-- ============================================================================

constraints

context Recorrido
    inv minimo5Km:
    self.distancia >= 5


context Coche
    inv revisionDespuesdeMatriculacion:
    self.revision -> forAll(rev | rev.fechaInicio > self.fechaMatriculacion)


    inv revisadoUnaVez:
    self.revision -> forAll(rev1, rev2 | not(rev1 <> rev2) or rev1.fechaInicio
        <> rev2.fechaInicio and (rev1.fechaInicio >= rev2.fechaFin or rev2.
        fechaInicio >= rev1.fechaFin))


context Revision
    inv mismaCiudadqueTallerEnRevision :
    let currentTime : Integer = self.coche.clock.NOW in   -- Accede al tiempo
        actual
    (currentTime - self.fechaFin) < 0 implies
    (self.taller.oclIsKindOf(Oficial) and self.taller.oclAsType(Oficial).ciudad
         = self.coche.ciudad) --
    or
    (self.taller.oclIsKindOf(NoOficial) and self.taller.oclAsType(NoOficial).
        ciudad = self.coche.ciudad) --


context Viaje
    inv viajandoOenCiudad:
    let currentTime : Integer = self.coche.clock.NOW in  -- Accede al tiempo
        actual
    (currentTime - self.fechaLlegada) < 0 implies
    self.coche.ciudad.oclIsUndefined() -- and self.coche.estaEn->notEmpty() --
        El coche existe en una ubicacion valida


context Coche
    inv enCiudadDestino:
    let ultimoViaje : Viaje = self.viaje -> asOrderedSet() -> sortedBy(
        fechaLlegada) -> last() in
    ultimoViaje.oclIsUndefined() or ultimoViaje.recorrido.destino = self.ciudad
```

```
context Coche
    inv viajeUnico:
    self.viaje -> forAll(v1,v2 | not(v1 <> v2) or v1.fechaSalida <> v2.
        fechaSalida and (v1.fechaLlegada <= v2.fechaSalida or v2.fechaLlegada
        <= v1.fechaSalida))


    inv destinoComoOrigen:
    let viajesOrdenados : OrderedSet(Viaje)  = self.viaje -> sortedBy(v | v.
        fechaSalida) in
    viajesOrdenados -> forAll(v1, v2 | viajesOrdenados->indexOf(v1) =
        viajesOrdenados->indexOf(v2) + 1 implies v1.recorrido.destino = v2.
        recorrido.origen)

context Ciudad
    inv hayRecorridos:
    self.origen -> notEmpty() and self.destino -> notEmpty()


context Coche
    inv empiezaViajeEnDestino:
    (self.viaje -> asOrderedSet() -> sortedBy(fechaSalida) -> last()).recorrido
        .destino = self.ciudad


    inv viajeDespuesdeMatriculacion:
    self.viaje -> forAll(viaje | viaje.fechaSalida > self.fechaMatriculacion)


    inv noViajeEnMantenimiento:
    self.necesitaMantenimiento implies self.viaje -> forAll(v | v.fechaLlegada
        < self.clock.NOW)


context Viaje
    inv fechaViajeBienDefinida:
    self.fechaLlegada > self.fechaSalida
```

2.5 Diagrama de Clases en USE

La herramienta USE nos permite llevar a cabo un diagrama de clases, el cual se muestra a continuación:

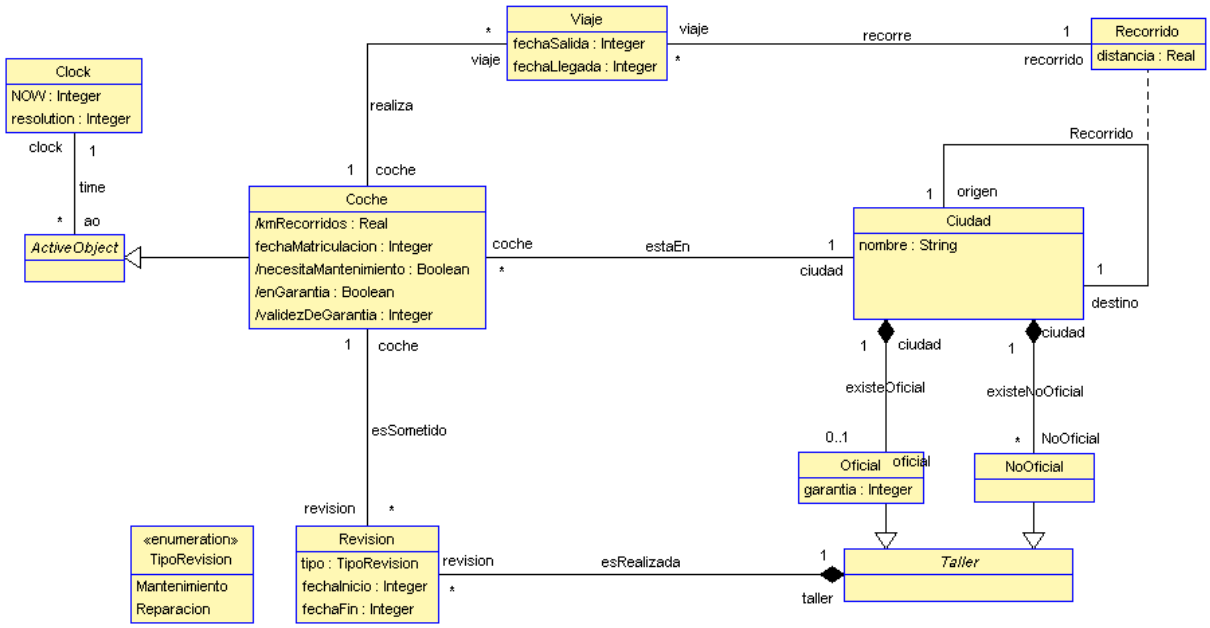

Figura 2: Diagrama del sistema de coches en USE

Todas las relaciones entre las entidades del modelo tienen su significado y están definidas de la siguiente manera:

- **esSometido (Coche-Revisión)**: Esta relación indica que cada coche en el sistema puede estar asociado a múltiples revisiones a lo largo de su vida útil. Cada revisión se registra cuando el coche necesita mantenimiento o reparación.
- **realiza (Coche-Viaje)**: Relación que representa el hecho de que un coche puede realizar varios viajes. Un viaje es un desplazamiento que inicia en una ciudad y concluye en otra, y se registra en el sistema para hacer seguimiento del historial de movimientos del coche.
- **estaEn (Coche-Ciudad)**: Esta relación representa la ubicación actual del coche en una ciudad. Cada coche estará siempre en una ciudad específica, ya sea de origen o destino de algún viaje.
- **esRealizada (Revisión-Taller)**: Define que toda revisión de un coche debe realizarse en un taller. Esto significa que las revisiones de mantenimiento y reparación se registran con el taller en el que se realizan.
- **existeOficial (Oficial-Ciudad)**: Relación que establece la existencia de un único taller oficial en cada ciudad. El taller oficial ofrece revisiones con garantía, lo cual beneficia a los coches que pasan por él.
- **existeNoOficial (NoOficial-Ciudad)**: Representa la existencia de uno o varios talleres no oficiales en cada ciudad. Estos talleres pueden realizar revisiones de mantenimiento y reparación, pero no otorgan garantía a los coches.
- **Recorrido (Ciudad-Ciudad)**: Relación de asociación entre dos ciudades, que define la distancia entre ellas. Esto permite modelar la distancia que un coche debe recorrer al viajar de una ciudad a otra.

- **time (Clock-ActiveObject)**: Esta relación conecta la instancia única de Clock con cada objeto activo (ActiveObject) del sistema, como los coches. A través de esta relación, el sistema puede realizar el seguimiento del paso del tiempo y cómo afecta a cada coche o entidad que depende del tiempo para su funcionamiento o estado.

2.6 Invariantes y .SOIL

Para poder comprobar las restricciones de nuestro sistema, primero es necesario definir las reglas que aseguran su correcto funcionamiento. Las restricciones o invariantes que se han definido son las siguientes:

- **1.**: Cada ciudad debe tener una distancia minima de 5 km para cada recorrido.

```
context Recorrido
  inv minimo5Km :
  self . distancia >= 5
```

Asegura que cada recorrido tenga una distancia mínima de 5 km. Esto evita que haya ciudades a menos de 5km. Además hemos decidido que si hay un recorrido entre una misma ciudad dicho recorrido debe ser mayor a 5 kilometros también.

- **2.**: Cada coche debe de pasar revision despues de matricularse y no antes.

```
context Coche
  inv revisionDespuesdeMatriculacion:
  self.revision -> forAll(rev | rev.fechaInicio > self.fechaMatriculacion)
```

Tiene que pasar obligatoriamente después porque antes de la fecha de matriculación no existe el dicho coche en nuestro modelo. Compara la fecha de matriculación del coche con la fecha de inicio de cada revisión asociada a él, y verifica que todas las revisiones cumplan con esta condición.

- **3.**: Todos los coches han de ser revisados como máximo en un momento dado.

```
inv revisadoUnaVez:
self.revision -> forAll(rev1, rev2 | not(rev1 <> rev2) or rev1.fechaInicio
     <> rev2.fechaInicio and (rev1.fechaInicio >= rev2.fechaFin or rev2.
    fechaInicio >= rev1.fechaFin))
```

Restringe a cada coche a ser revisado solo una vez en un tiempo dado, evitando revisiones superpuestas en el tiempo. Lo que hace es revisar las fechas de inicio y fin de cada revisión para asegurarse de que ninguna de ellas se superponga.

- **4.**: Si un coche esta en revision, debe de estar en la misma ciudad que el taller.

```
context Revision
inv mismaCiudadqueTallerEnRevision :
let currentTime : Integer = self.coche.clock.NOW in
(currentTime - self.fechaFin) < 0 implies
(self.taller.oclIsKindOf(Oficial) and self.taller.oclAsType(Oficial).
    ciudad = self.coche.ciudad)
or
(self.taller.oclIsKindOf(NoOficial) and self.taller.oclAsType(NoOficial).
    ciudad = self.coche.ciudad)
```

Asegura que un coche en revisión se encuentre en la misma ciudad que el taller donde se realiza la revisión. Esto garantiza que el coche y el taller están físicamente en el mismo lugar cuando se realiza el mantenimiento. Verifica la ciudad asociada al taller y la compara con la ciudad en la que se encuentra el coche en ese momento.

- **5.**: Un coche tiene que estar o viajando o en una ciudad.

```
context Viaje
  inv viajandoOenCiudad:
  let currentTime : Integer = self.coche.clock.NOW in
  (currentTime - self.fechaLlegada) < 0 implies
  self.coche.ciudad.oclIsUndefined()
```

Asegura que un coche se encuentre en una ciudad o esté realizando un viaje, pero no ambas cosas a la vez. Esto previene conflictos de ubicación, indicando que un coche en movimiento no está en una ciudad específica hasta que llegue a su destino. Evalúa si el coche está en un viaje actualmente (determinando que su fecha de llegada es futura) y, si es así, asegura que el coche no tenga una ciudad asignada hasta que termine el viaje.

- **6.**: Un coche después de su viaje tiene que encontrarse en su ciudad destino.

```
context Coche
  inv enCiudadDestino:
  let ultimoViaje : Viaje = self.viaje -> asOrderedSet() -> sortedBy(
      fechaLlegada) -> last() in
  ultimoViaje.recorrido.destino = self.ciudad
```

Establece que, al finalizar un viaje, el coche se encuentra en la ciudad de destino del recorrido, asegurando que los viajes finalicen en el lugar esperado. Verifica la última ciudad de destino registrada y la compara con la ubicación actual del coche, garantizando coherencia en la posición final.

- **7.**: No pueden haber dos viajes solapados.

```
context Coche
  inv viajeUnico:
  self.viaje -> forAll(v1,v2 | not(v1 <> v2) or v1.fechaSalida <> v2.
      fechaSalida and (v1.fechaLlegada <= v2.fechaSalida or v2.fechaLlegada
      <= v1.fechaSalida))
```

Impide que un coche tenga dos viajes concurrentes, asegurando que un coche solo pueda realizar un viaje a la vez. Revisa las fechas de inicio y fin de cada viaje asociado al coche y se asegura de que no existan superposiciones, es decir, que los períodos de cada viaje sean únicos.

- **8.**: Un coche debe de empezar el siguiente viaje en la ciudad destino del anterior.

```
  inv destinoComoOrigen:
  let viajesOrdenados : OrderedSet(Viaje)  = self.viaje -> sortedBy(v | v.
      fechaSalida) in
  viajesOrdenados -> forAll(v1, v2 | viajesOrdenados->indexOf(v1) =
      viajesOrdenados->indexOf(v2) + 1 implies v1.recorrido.destino = v2.
      recorrido.origen)
```

Asegura que el siguiente viaje de un coche comience en la ciudad de destino de su viaje anterior, lo que garantiza una secuencia lógica de viajes. Para lograr esto, ordena los viajes en secuencia temporal y verifica que cada destino sea el punto de partida del siguiente viaje.

- **9.**: Una ciudad tiene que tener un recorrido con otra ciudad de manera obligatoria

```
context Ciudad
  inv hayRecorridos:
  self.origen -> notEmpty() and self.destino -> notEmpty()
```

Asegura que cada ciudad tenga recorridos que la conecten con otras ciudades, promoviendo la conectividad en el sistema y evitando que existan ciudades aisladas. Verifica que existan recorridos que incluyan la ciudad como punto de partida o de llegada.

- **10.**: Un coche debe de estar en la ciudad destino de su último viaje antes de iniciar un viaje.

```
context Coche
  inv empiezaViajeEnDestino:
  (self.viaje -> asOrderedSet() -> sortedBy(fechaSalida) -> last()).
      recorrido.destino = self.ciudad
```

Define que un coche debe estar en la ciudad de destino de su último viaje antes de comenzar otro viaje, garantizando consistencia en la ubicación del coche entre viajes. Ordena los viajes de un coche en secuencia y compara el destino del último viaje con el origen del siguiente, asegurando que el coche no "salte"de una ciudad a otra sin haber llegado previamente a esa ubicación. Esto mantiene la coherencia en los desplazamientos del coche.

- **11.**: Cada coche debe de pasar revision despues de matricularse y no antes.

```
    inv viajeDespuesdeMatriculacion:
    self.viaje -> forAll(viaje | viaje.fechaSalida > self.fechaMatriculacion)
```

Restringe a los coches a realizar viajes solo después de su fecha de matriculación, lo que previene errores lógicos de viajes previos al registro oficial del vehículo. Compara la fecha de matriculación con la fecha de salida de cada viaje y asegura que todos los viajes comiencen después de dicha fecha.

- **12.**: Un coche no debe poder comenzar un viaje con la alerta de mantenimiento en true.

```
    inv noViajeEnMantenimiento:
    self.necesitaMantenimiento implies self.viaje -> forAll(v | v.fechaLlegada
        < self.clock.NOW)
```

Asegura que un coche con alerta de mantenimiento no inicie ningún viaje, promoviendo la seguridad y mantenimiento adecuado. Esto se logra comprobando que si necesitaMantenimiento es true, todos los viajes del coche ya hayan finalizado antes del tiempo actual

- **13.**: Un viaje no puede tener una fecha de inicio superior a su fecha de llegada.

```
context Viaje
    inv fechaViajeBienDefinida:
    self.fechaLlegada > self.fechaSalida
```

Asegura que la fecha de salida de un viaje sea anterior o igual a su fecha de llegada, previniendo datos inconsistentes en los registros de viaje.

2.7 Diagramas de objetos para cada restricción

A partir de aqui podemos desarrollar los archivos de prueba que respaldaran el funcionamiento de nuestro sistema. Vamos a comprobar restricción por restricción, comprobando cada caso con un ejemplo y un contraejemplo.

Hemos considerado, por las peticiones de concisión y brevedad, que la inclusión del código soil de cada uno de los modelos de objetos que comprueban invariantes y derivados enturbia y entorpece la lectura y comprensión del documento. A continuación se muestran los diagramas resultantes de dichos códigos, junto a sus comprobaciones. Los códigos fuente se han adjuntado con la entrega.

2.7.1 Invariante 1: mínimo5Km

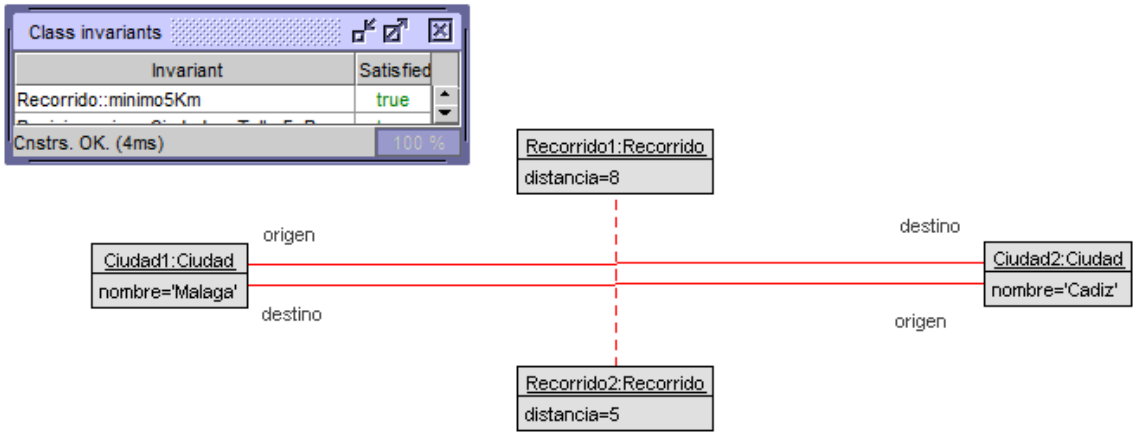

Figura 3: Diagrama de objetos para la invariante al cumplirse

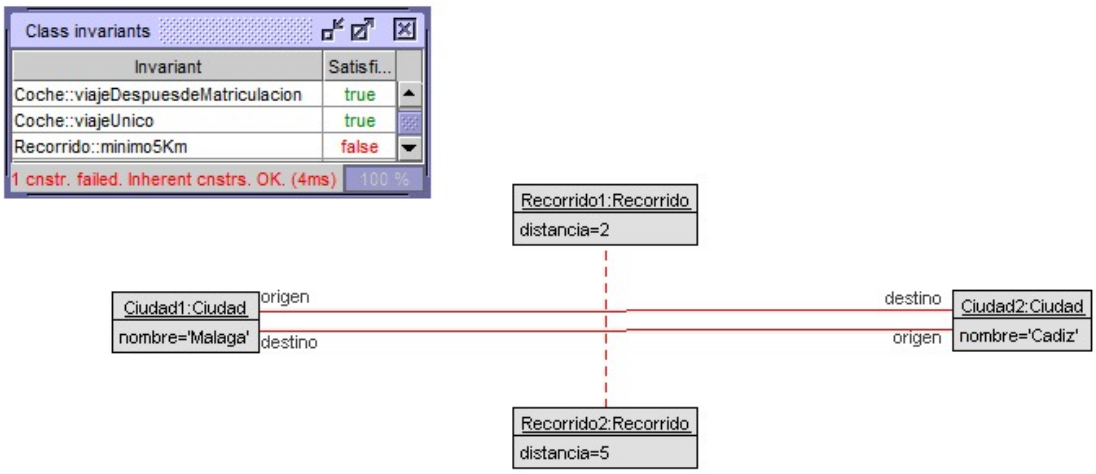

Figura 4: Diagrama de objetos para la invariante al incumplirse

La invariante 1 garantiza que cada recorrido entre ciudades tenga una distancia mínima de 5 km. Se considera cumplida si la distancia del recorrido es igual o superior a 5 km, y no cumplida si el recorrido es menor a esa distancia.

2.7.2 Invariante 2: revisionDespuesdeMatriculacion

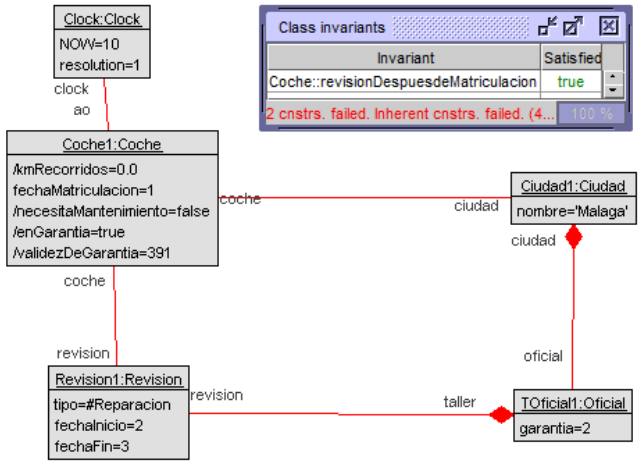

Figura 5: Diagrama de objetos para la invariante al cumplirse

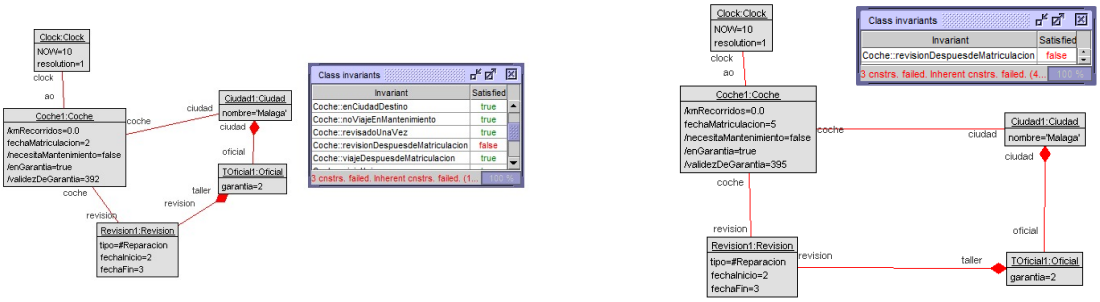

Figura 6: Diagramas de objetos para la invariante al incumplirse

La invariante 2 garantiza que cada coche pase su revisión después de la matriculación. Se considera cumplida si las revisiones ocurren después de esta fecha y no cumplida si la revisión se realiza el mismo día o antes de la matriculación.

2.7.3 Invariante 3: revisadoUnaVez

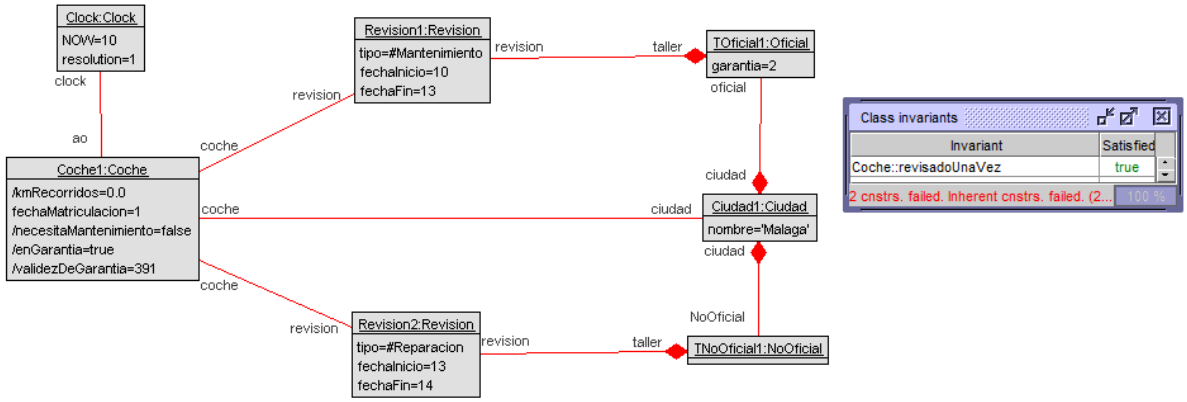

Figura 7: Diagrama de objetos para la invariante al cumplirse

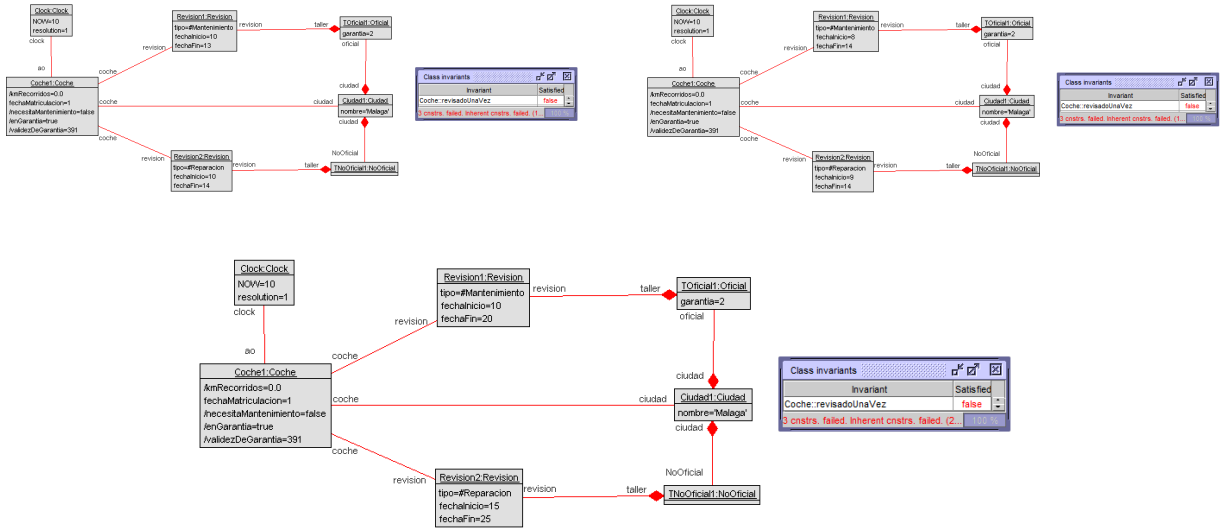

Figura 8: Diagramas de objetos para la invariante al incumplirse

La invariante 3 garantiza que cada coche sea revisado como máximo en un único momento dado. Se considera cumplida si no existen revisiones que se solapen, ya sea por fecha de inicio, fecha de finalización o si una revisión empieza antes de que termine otra. No se cumple cuando dos revisiones a un coche coinciden en sus fechas de inicio, finalización o si una comienza antes de que termine la otra.

2.7.4 Invariante 4: mismaCiudadqueTallerEnRevision

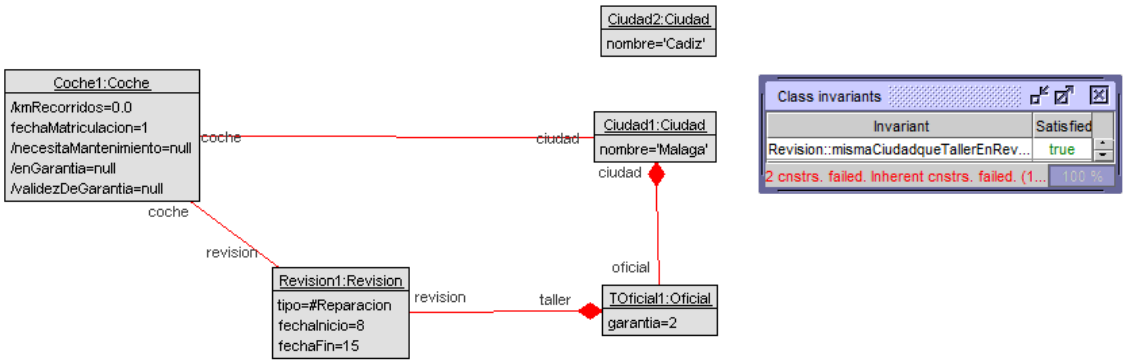

Figura 9: Diagrama de objetos para la invariante al cumplirse

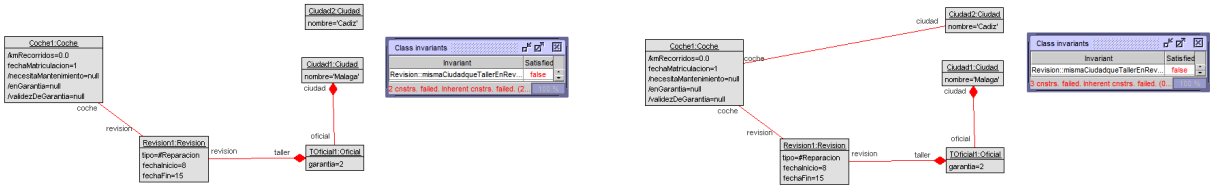

Figura 10: Diagramas de objetos para la invariante al incumplirse

La invariante 4 establece que un coche en revisión debe encontrarse en la misma ciudad que el taller donde se realiza la revisión. Se considera cumplida si el coche está en la misma ciudad que el taller. No se cumple cuando el coche no está en ninguna ciudad o si se encuentra en una ciudad distinta a la del taller.

2.7.5 Invariante 5: viajandoOenCiudad

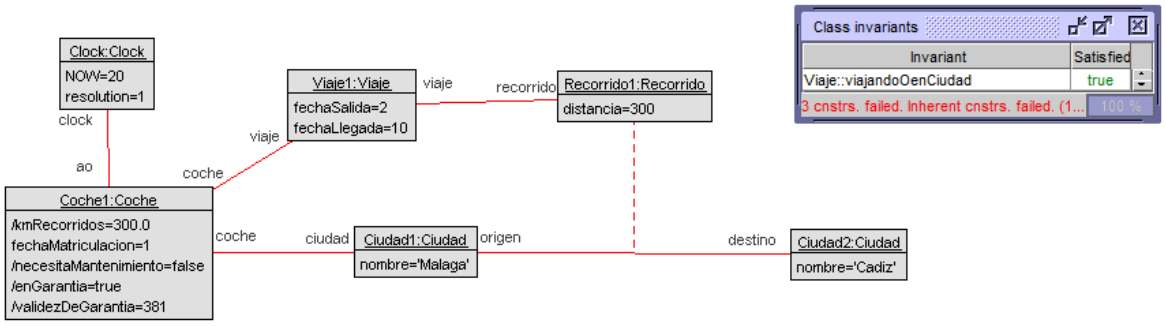

Figura 11: Diagrama de objetos para la invariante al cumplirse

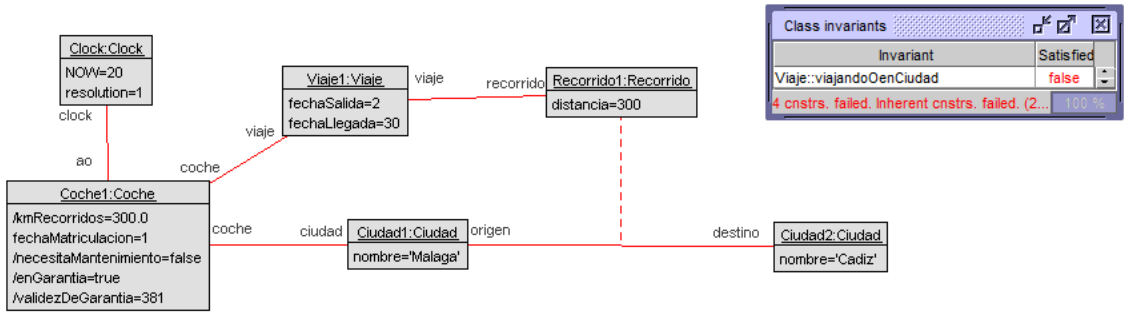

Figura 12: Diagrama de objetos para la invariante al incumplirse

La invariante 5 establece que un coche debe estar exclusivamente viajando o en una ciudad en un momento dado. Se considera cumplida si el coche se encuentra en una de estas dos condiciones. No se cumple cuando el coche está simultáneamente viajando y en una ciudad.

2.7.6 Invariante 6: enCiudadDestino

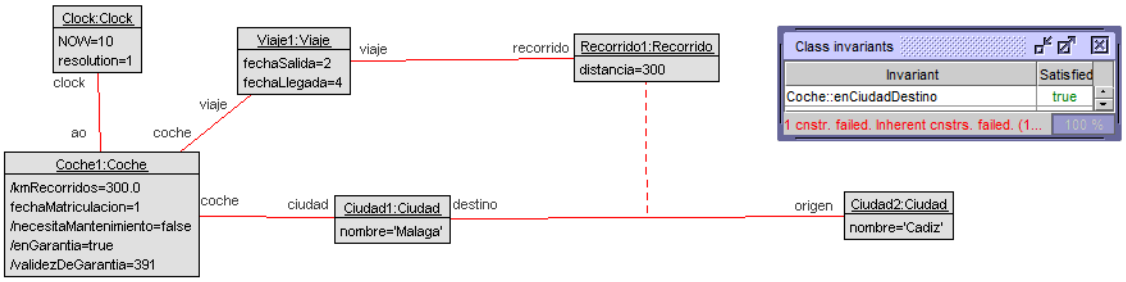

Figura 13: Diagrama de objetos para la invariante al cumplirse

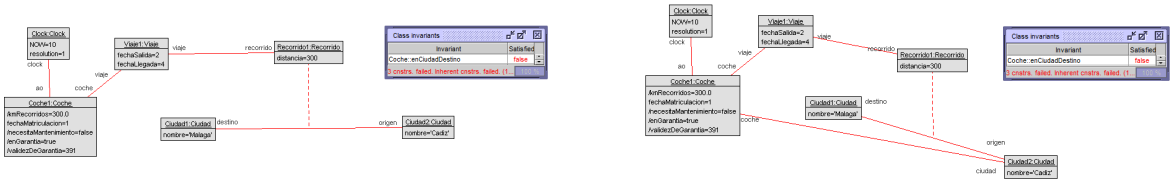

Figura 14: Diagramas de objetos para la invariante al incumplirse

La invariante 6 establece que un coche, después de completar su viaje, debe encontrarse en su ciudad destino. Se cumple cuando el coche está en la ciudad de destino al finalizar el viaje. No se cumple si, al finalizar el viaje, el coche no se encuentra en ninguna ciudad o si está en una ciudad diferente a la de destino.

2.7.7 Invariante 7: viajeUnico

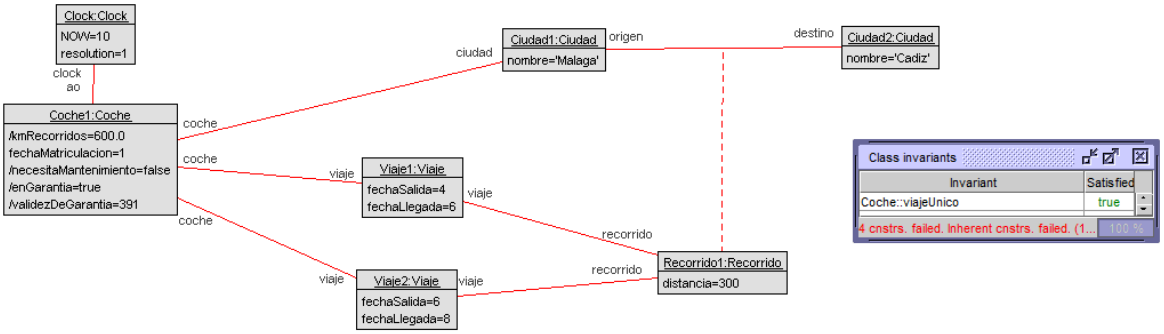

Figura 15: Diagrama de objetos para la invariante al cumplirse

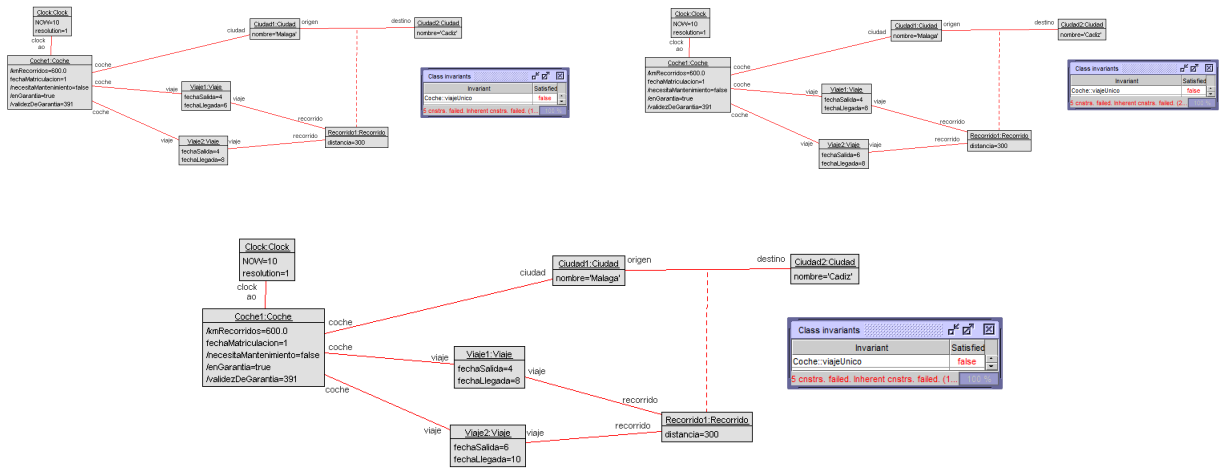

Figura 16: Diagramas de objetos para la invariante al incumplirse

La invariante 7 establece que no pueden existir dos viajes solapados. Se considera cumplida cuando las fechas de inicio y finalización de los viajes no coinciden. No se cumple si las fechas de inicio o finalización de dos viajes coinciden o si un viaje comienza antes de que termine otro viaje del mismo coche.

2.7.8 Invariante 8: destinoComoOrigen

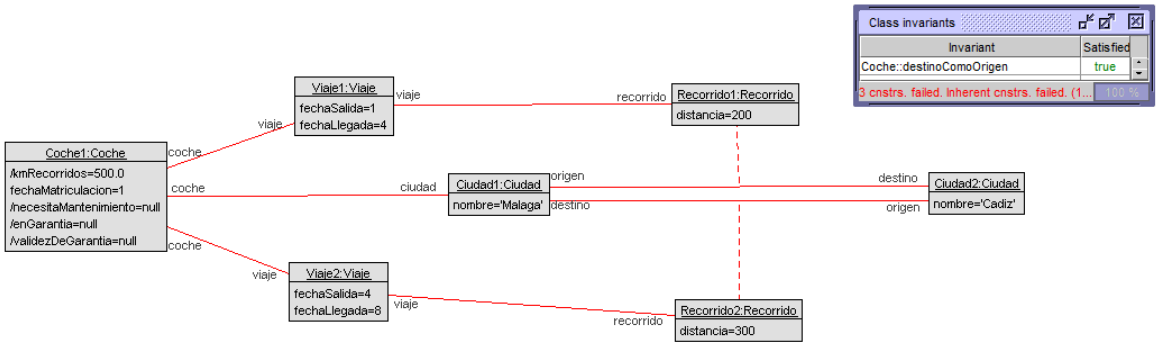

Figura 17: Diagrama de objetos para la invariante al cumplirse

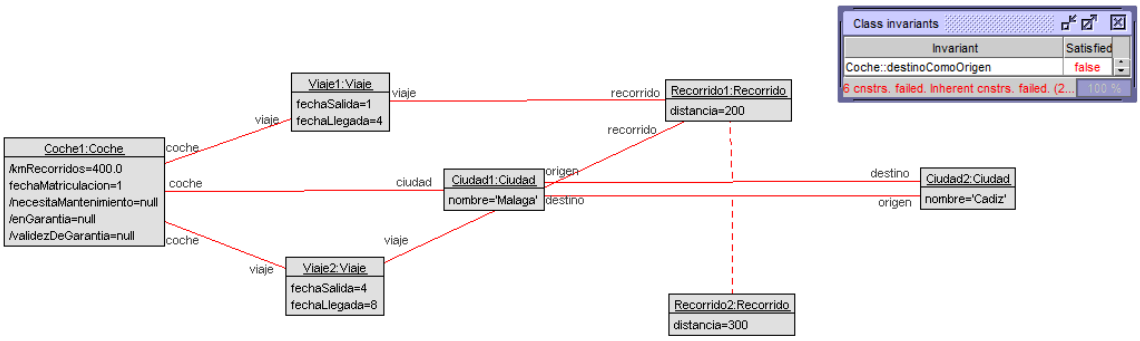

Figura 18: Diagrama de objetos para la invariante al incumplirse

La invariante 8 exige que un coche comience su siguiente viaje en la ciudad de destino del anterior. Se cumple cuando el coche inicia su viaje en la ciudad de destino del viaje anterior. No se cumple si el coche comienza el siguiente viaje en una ciudad diferente.

2.7.9 Invariante 9: hayRecorridos

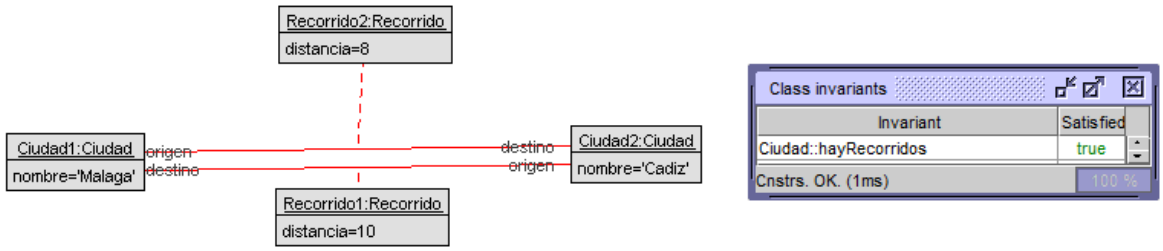

Figura 19: Diagrama de objetos para la invariante al cumplirse

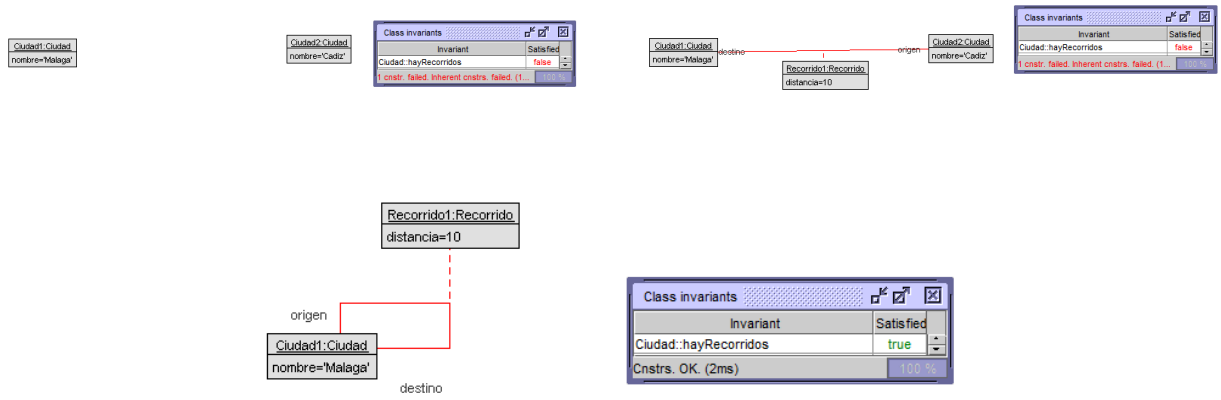

Figura 20: Diagramas de objetos para la invariante al incumplirse

La invariante 9 establece que cada ciudad debe tener al menos un recorrido con otra ciudad. Se cumple si una ciudad tiene al menos un recorrido hacia otra ciudad. También se cumple si una ciudad tiene un recorrido consigo misma. No se cumple si una ciudad no tiene ningún recorrido asociado o si tiene recorridos pero no es el origen de ninguno.

2.7.10 Invariante 10: empiezaViajeEnDestino

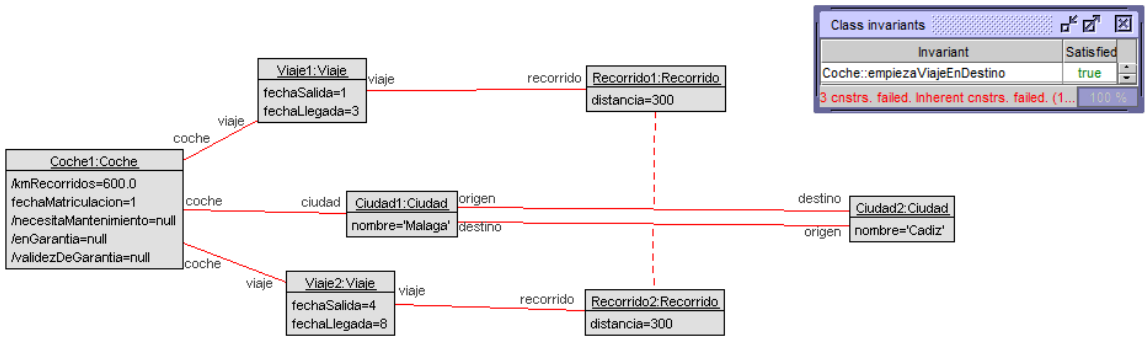

Figura 21: Diagrama de objetos para la invariante al cumplirse

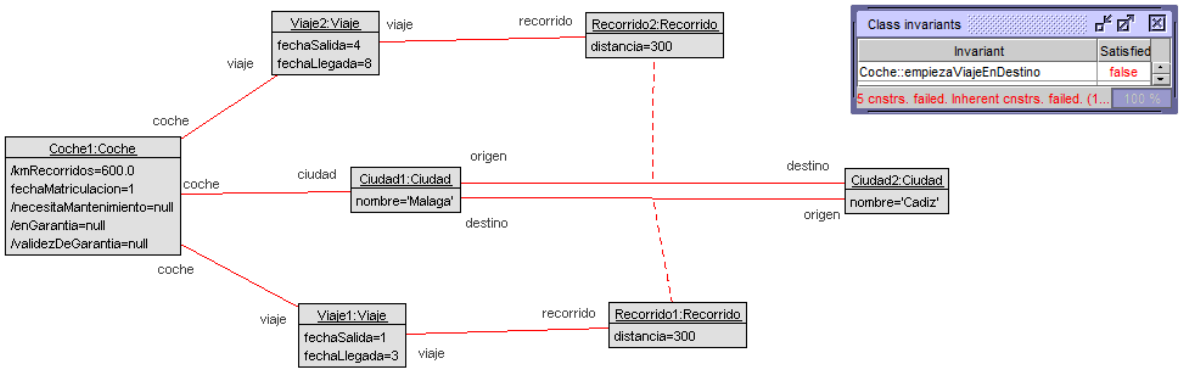

Figura 22: Diagrama de objetos para la invariante al incumplirse

La invariante 10 establece que un coche debe encontrarse en la ciudad destino de su último viaje antes de iniciar uno nuevo. Se cumple cuando el coche está en la ciudad de destino de su último viaje. No se cumple si el coche no está en esta ciudad antes de iniciar un nuevo viaje.

2.7.11 Invariante 11: viajeDespuesdeMatriculacion

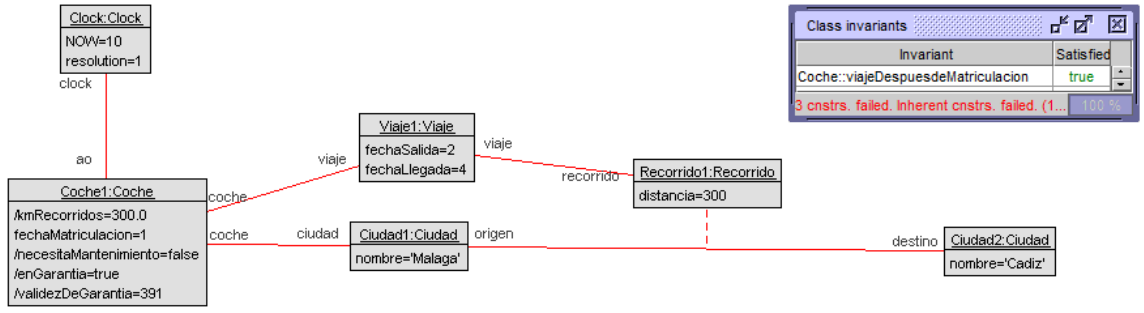

Figura 23: Diagrama de objetos para la invariante al cumplirse

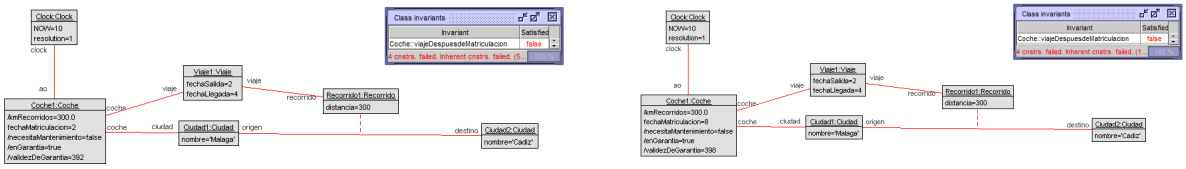

Figura 24: Diagramas de objetos para la invariante al incumplirse

La invariante 11 prohíbe que un coche viaje antes de su fecha de matriculación. Se cumple cuando el coche viaja después de la fecha de matriculación. No se cumple si el viaje se realiza antes de esa fecha.

2.7.12 Invariante 12: noViajeEnMantenimiento

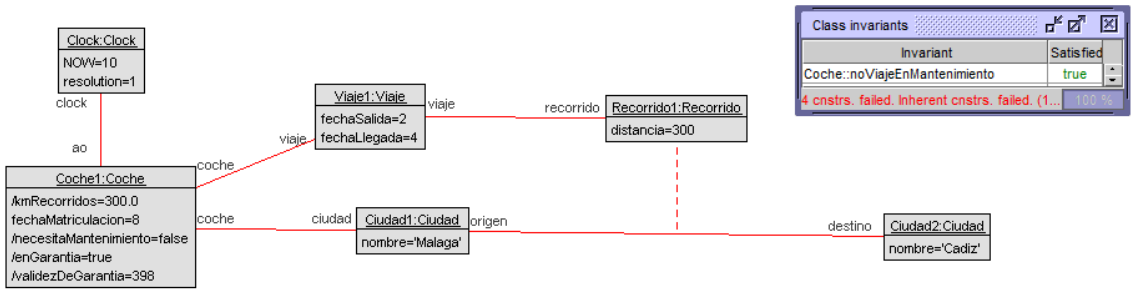

Figura 25: Diagrama de objetos para la invariante al cumplirse

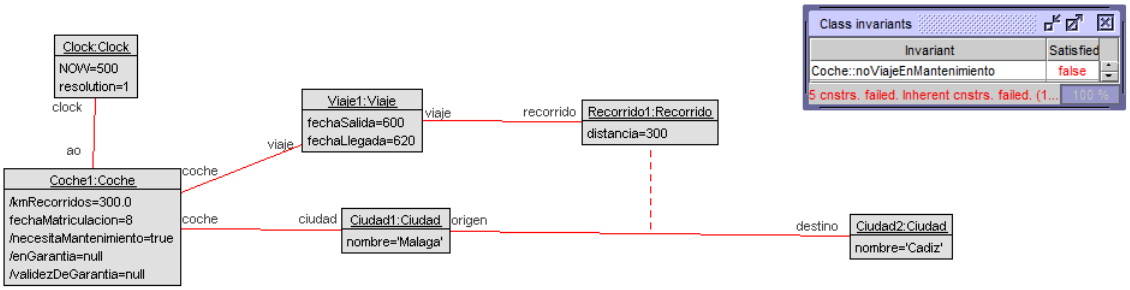

Figura 26: Diagrama de objetos para la invariante al incumplirse

La invariante 12 asegura que un coche no puede comenzar un viaje con la flag de necesitaMantenimiento activa. Se cumple cuando el coche no tiene la alerta de mantenimiento activa al iniciar un viaje. No se cumple si el coche inicia un viaje con la alerta de mantenimiento en true.

2.7.13 Invariante 13: fechaViajeBienDefinida

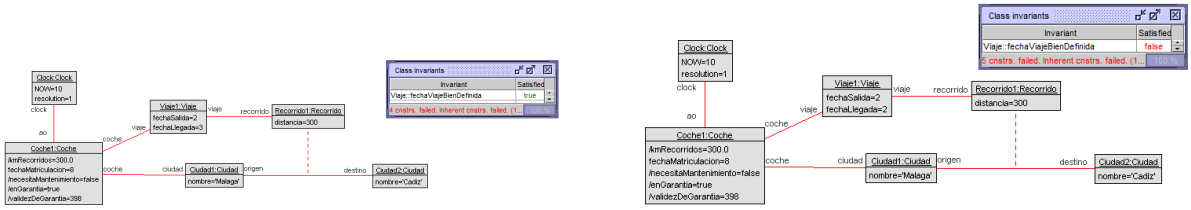

Figura 27: Diagrama de objetos para la invariante al cumplirse

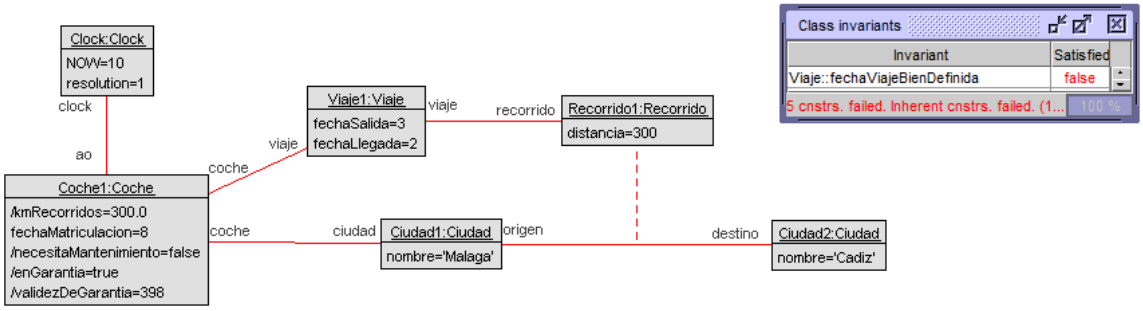

Figura 28: Diagrama de objetos para la invariante al incumplirse

La invariante 13 establece que un viaje no puede tener una fecha de inicio posterior a su fecha de llegada. Se cumple cuando la fecha de inicio es anterior o igual a la fecha de llegada. No se cumple si la fecha de inicio es posterior a la de llegada.

De esta manera podemos comprobar que todas las restricciones se cumplen.

3 Modelado Dinámico

En este apartado, se debe entregar la imagen del diagrama de clases y el código USE desarrollado (al ser este apartado un incremento respecto del anterior, hay que entregar únicamente el código USE nuevo). En las operaciones añadidas, especificar las pre- y post- condiciones.

3.1 Diagrama de clases

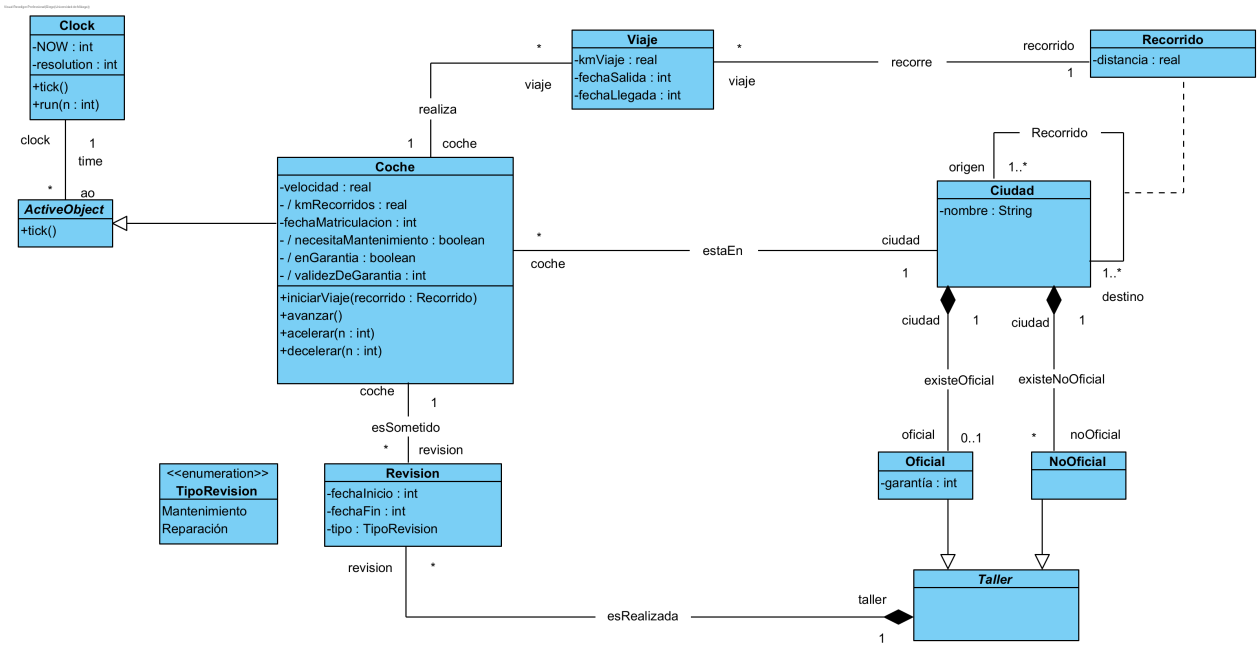

Figura 29: Diagrama del sistema de coches para el modelo dinámico

A diferencia del diagrama del modelo estructural, en este modelo hemos añadido dos atributos nuevos en las clases Coche y Viaje. Con estos atributos adicionales, el modelo puede ofrecer una representación más completa y precisa de los trayectos y los coches involucrados.

3.2 Novedades y modificaciones en las clases del sistema dinámico

3.2.1 Clock

Descripción: Hemos añadido las operaciones tick() y run(n:Integer) para simular el paso del tiempo.

- **Operaciones:**
  - `tick()`: Usada para avanzar en el tiempo la cantidad definida en *resolution*.
  - `run(n : Integer)`: Permite avanzar en el tiempo n pasos en un solo tick.

3.2.2 ActiveObject

Descripción: Hemos añadido la operación tick() para que los objetos que hereden de la clase ActiveObject se vean afectados por el paso del tiempo.

- **Operaciones:**

- `tick()`: Invoca el método tick() de la instancia de Clock. Esto significa que cada objeto activo tiene acceso a la funcionalidad del reloj y puede actualizar el tiempo del sistema llamando al método tick() de Clock.

3.2.3 Coche

Descripción: Hemos añadido el atributo "velocidad"que en un inicio vale 0 como también varias operaciones como iniciarViaje(recorrido : Recorrido), avanzar(), acelerar(n : Integer), decelerar(n : Integer) y tick().

- **Atributos:**
  - `velocidad: Real init : 0` : Velocidad del coche. Inicialmente está en 0, es decir, que está parado.
  - `kmRecorridos: Real (derive)`: En el dinámico consideramos que si existen viajes previos, el cálculo se realiza sumando las distancias de los recorridos de todos los viajes anteriores, excluyendo el último viaje. A este valor se le añade el kilometraje del último viaje (kmViaje) para obtener el total de kilómetros recorridos por el coche. Esto permite tener un registro dinámico y actualizado de la distancia total recorrida por el coche a medida que se realizan nuevos viajes.
- **Operaciones:**
  - `iniciarViaje(recorrido : Recorrido)`: Esta operación inicia un nuevo viaje para el coche. Se crea un objeto Viaje, se establece la fecha de salida como el tiempo actual y se relaciona al coche con el viaje y el recorrido correspondiente. También actualiza la ciudad en la que se encuentra el coche, eliminándola de la relación estaEn.
  - `avanzar()`: Esta operación actualiza el kilometraje del coche durante su viaje, sumando su velocidad actual al kilometraje recorrido. Si el coche ha alcanzado la distancia total del recorrido, la operación finaliza el viaje, marca la fecha de llegada a la actual, establece la velocidad a 0 y actualiza la ciudad en la que el coche termina su viaje.
  - `acelerar(n : Integer)`: Permite incrementar la velocidad del coche en n unidades. La operación ajusta la velocidad del coche sumando el valor n a la velocidad actual.
  - `decelerar(n : Integer)`: Reduce la velocidad del coche en n unidades, siempre que la velocidad actual sea mayor o igual al valor n. Esto disminuye la velocidad del coche de acuerdo al valor especificado.
  - `tick()`: Actualiza el estado del coche en función del tiempo. Si el coche está en un viaje en curso, avanza en su trayecto según la velocidad y distancia recorrida. Esta operación depende del reloj del sistema para actualizar el tiempo y garantizar el progreso del coche en su viaje.

3.2.4 Viaje

Descripción: Hemos añadido el atributo "kmViaje"que en un inicio vale 0.

- **Atributos:**
  - `kmViaje : Real init : 0` : Los kilómetros que ha realizado el coche durante el viaje.

3.3 Diagrama de Clases del modelo dinámico en USE

La herramienta USE nos permite llevar a cabo un diagrama de clases para el modelo dinámico, el cual se muestra a continuación:

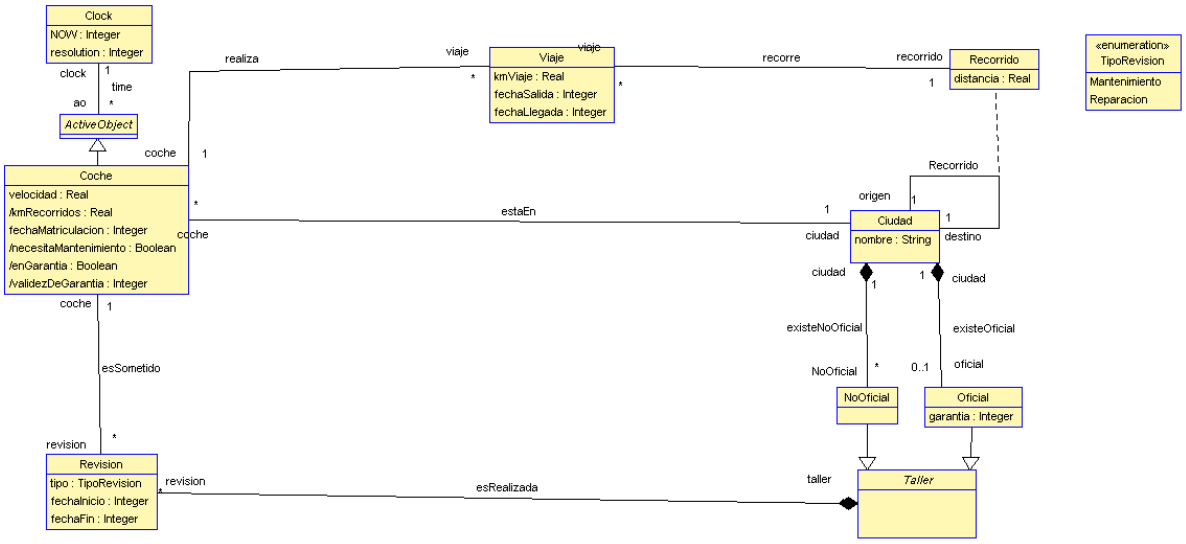

Figura 30: Diagrama del sistema de coches en USE

Las relaciones en el modelo dinámico se mantienen igual que en el modelo estructural que ya se han descrito anteriormente.

3.4 Codigo Use del apartado B

Listing 2: Codigo Use del apartado B

```
-- ================================================================
--                            MODELO
-- ================================================================

model P2_Toyota

-- ================================================================
--                          ENUMERATIONS
-- ================================================================

enum TipoRevision{Mantenimiento, Reparacion}


-- ================================================================
--                            CLASES
-- ================================================================

class Clock
attributes
    NOW : Integer init : 0                    -- Nos dice el dia actual en el
        que se encuentra el sistema
    resolution : Integer init : 1
operations
    tick()
```

```
    begin
        self.NOW := self.NOW + self.resolution;
        for o in self.ao do
            o.tick()
        end;
    end
    post sumaTiempo: self.NOW = self.NOW@pre + self.resolution

    run(n : Integer)
    begin
        for i in Sequence{1..n} do
            self.tick()
        end;
    end
    pre esPositivo: n > 0
    post sumaCorrecta: self.NOW = self.NOW@pre + n
end

abstract class ActiveObject
operations
    tick()
    begin
        self.clock.tick()
    end
end

class Coche < ActiveObject
    attributes
        velocidad : Real init : 0
        --Derivado 1:
        --Para comprobar que el derivado kmRecorridos funciona correctamente:
        --Se han desarrollado 3 archivos .soil:
        ---El primero comprueba
        kmRecorridos : Real derive :              -- Registro de todos los km
            que ha recorrido el coche entre ciudades en sus viajes

        if self.viaje -> size() = 0 then 0 else
        let ultimoViaje : Viaje = self.viaje -> sortedBy(v|v.fechaSalida) ->
            last() in
                self.viaje -> excluding(ultimoViaje).recorrido -> collect(dist
                    | dist.distancia) -> sum() + ultimoViaje.kmViaje
        endif

        fechaMatriculacion : Integer              -- Fecha en la que el coche
            se puso en funcionamiento en nuestro sistema

        necesitaMantenimiento : Boolean derive :  -- Flag que indica que el
            coche necesita ir a revision Tipo: Mantenimiento
            -- razones de necesitaMantenimiento
           if  self.clock.NOW  - fechaMatriculacion < 400
            then
                false
            else
                let ultimaRevision : Revision = self.revision -> select(rev |
                    rev.tipo = TipoRevision::Mantenimiento) -> sortedBy(r | r.
                    fechaInicio) -> last() in
```

```
            if ultimaRevision <> null
            then
                (self.clock.NOW - ultimaRevision.fechaFin) > 100
            else
                true
            endif
        endif

    enGarantia : Boolean derive :              -- Flag que indica que el
        coche tiene una garantia activa
        -- razones por estar enGarantia
        ((self.clock.NOW  - fechaMatriculacion) < 400) or
        (validezDeGarantia > 0)

    validezDeGarantia : Integer derive :       -- Dias por los que la
        garantia proporcionada por un tallerOficial esta activa
    if (self.clock.NOW - fechaMatriculacion) < 400
        then
            400 - (self.clock.NOW  - fechaMatriculacion) -- Asignamos el
                valor de la validez restante
        else
            let revisionesOficiales : OrderedSet(Revision) = self.revision
                -> select(rev | rev.taller.oclIsKindOf(Oficial)) ->
                sortedBy(rev | rev.fechaInicio) in
                if revisionesOficiales->size() > 0 and (revisionesOficiales
                     -> last()).fechaFin <> null and (self.clock.NOW - (
                    revisionesOficiales -> last()).fechaFin) < (
                    revisionesOficiales -> last()).taller.oclAsType(Oficial
                    ).garantia
                    then
                        (revisionesOficiales -> last()).taller.oclAsType(
                            Oficial).garantia - (self.clock.NOW - (
                            revisionesOficiales -> last()).fechaFin)
                    else
                        0
                endif
    endif
operations
    iniciarViaje(recorrido : Recorrido)
    begin
        declare nuevoViaje : Viaje;
        nuevoViaje := new Viaje;
        nuevoViaje.fechaSalida := self.clock.NOW;
        insert(self, nuevoViaje) into realiza;
        insert(nuevoViaje, recorrido) into recorre;
        delete(self, self.ciudad) from estaEn;
    end
    pre estaEnCiudad : self.ciudad = recorrido.origen
    pre viajeNoEnCurso : self.viaje->isEmpty() or not (self.viaje->
        asOrderedSet()->sortedBy(v | v.fechaSalida)->last().fechaLlegada =
        null)
    pre noNecesitaMantenimiento : self.necesitaMantenimiento implies self.
        viaje -> forAll(v | v.fechaLlegada <> null)
    post noEstaEnCiudad : self.ciudad.oclIsUndefined()

    avanzar()
```

```
        begin
            declare viajeEnCurso : Viaje;
            viajeEnCurso := self.viaje -> asOrderedSet() -> sortedBy(v | v.
                fechaSalida) -> last();
            viajeEnCurso.kmViaje := viajeEnCurso.kmViaje + self.velocidad;
            if viajeEnCurso.kmViaje > viajeEnCurso.recorrido.distancia then
                viajeEnCurso.kmViaje := viajeEnCurso.recorrido.distancia;
                self.velocidad := 0;
                viajeEnCurso.fechaLlegada := self.clock.NOW;
                insert(self, viajeEnCurso.recorrido.destino) into estaEn;
            end;
        end
        pre estaEnViaje: (self.viaje -> asOrderedSet() -> sortedBy( v | v.
            fechaSalida) -> last()).fechaLlegada = null and self.ciudad.
            oclIsUndefined()

        acelerar(n : Integer)
        begin
            self.velocidad := self.velocidad + n;
        end
        pre sumandoValido : n > 0
        post sumaCorrecta : self.velocidad = self.velocidad@pre + n

        decelerar(n : Integer)
        begin
            self.velocidad := self.velocidad - n;
        end
        pre sumandoValido : n > 0 and self.velocidad >= n
        post restaCorrecta : self.velocidad = self.velocidad@pre - n

        tick()
        begin
            -- Verificamos que el coche esta realizando un viaje para llamar la
                 operacion de avanzar
            if (self.viaje -> asOrderedSet() -> sortedBy( v | v.fechaSalida) ->
                 last()).fechaLlegada = null and self.ciudad.oclIsUndefined()
            then
                self.avanzar();
            end;
        end
end

class Viaje
attributes
    kmViaje : Real init : 0
    fechaSalida : Integer                    -- Dia en el que el coche
        comienza en viaje
    fechaLlegada : Integer                   -- Dia en el que el coche
        finaliza el viaje
end

class Revision
attributes
    tipo : TipoRevision                      -- Selecciona el tipo de revision
         del enumeration definido
    fechaInicio : Integer                    -- Guarda el registro de cuando
```

```
        inicio la revision del coche
    fechaFin : Integer                          -- Guarda el registro de cuando
        el coche sale de revision
end

abstract class Taller                           -- Clase abstracta Taller. No
    debe tener instancias, solo sus subclases tienen
end

class Oficial < Taller                          -- Taller Oficial hereda de
    Taller
attributes
    garantia : Integer                          -- Total del dias en lo que el
        coche estara enGarantia
end

class NoOficial < Taller                       -- Taller No Oficial hereda de
    Taller
end

class Ciudad
attributes
    nombre : String                             -- Nombre de la ciudad
end


-- ================================================================
 --                        CLASES DE ASOCIACION
-- ================================================================

associationclass Recorrido between
    Ciudad [1] role destino
    Ciudad [1] role origen
attributes
    distancia : Real
end

-- ================================================================
--                          ASOCIACIONES
-- ================================================================

association time between
Clock [1] role clock
ActiveObject [*] role ao
end

association realiza between
Coche [1] role coche
Viaje [*] role viaje
end

association recorre between
Viaje [*] role viaje
Recorrido [1] role recorrido
end
```

```
association estaEn between
Coche [*] role coche
Ciudad [1] role ciudad
end

association esSometido between
Coche [1] role coche
Revision [*] role revision
end

composition esRealizada between
Taller [1] role taller
Revision [*] role revision
end

composition existeOficial between
Ciudad [1] role ciudad
Oficial [0..1] role oficial
end

composition existeNoOficial between
Ciudad [1] role ciudad
NoOficial [*] role  NoOficial
end


-- ============================================================================
--                            INVARIANTES
-- ============================================================================

constraints
-- Invariante 1: Cada cuidad debe tener una distancia minima de 5 km de la otra
    .
context Recorrido
    inv minimo5Km:
    self.distancia >= 5

-- Invariante 2: Cada coche debe de pasar revision despues de matricularse y no
     antes.
context Coche
    inv revisionDespuesdeMatriculacion:
    self.revision -> forAll(rev | rev.fechaInicio > self.fechaMatriculacion)


    inv revisadoUnaVez:
    self.revision -> forAll(rev1, rev2 | not(rev1 <> rev2) or rev1.fechaInicio
        <> rev2.fechaInicio and (rev1.fechaInicio >= rev2.fechaFin or rev2.
        fechaInicio >= rev1.fechaFin) and (not(rev1.fechaFin = null and rev2.
        fechaFin = null)))

-- Invariante 4: Si un coche esta en revision, debe de estar en la misma ciudad
     que el taller.
context Revision
    inv mismaCiudadqueTallerEnRevision :
    self.fechaFin = null implies
    (self.taller.oclIsKindOf(Oficial) and self.taller.oclAsType(Oficial).ciudad
```

```
        = self.coche.ciudad)
    or
    (self.taller.oclIsKindOf(NoOficial) and self.taller.oclAsType(NoOficial).
        ciudad = self.coche.ciudad)

-- Invariante 5: Un coche tiene que estar o viajando o en una ciudad.
context Viaje
    inv viajandoOenCiudad:
    self.fechaLlegada = null implies-- El coche no existe
    self.coche.ciudad.oclIsUndefined() -- and self.coche.estaEn->notEmpty() --
        El coche existe en una ubicacion valida

-- Invariante 6: Un coche despues de su viaje tiene que encontrarse en su
    ciudad destino.
context Coche
    inv enCiudadDestino:
    let ultimoViaje : Viaje = self.viaje -> asOrderedSet() -> sortedBy(
        fechaLlegada) -> last() in
    ultimoViaje.oclIsUndefined() or ultimoViaje.recorrido.destino = self.ciudad

-- Invariante 7: No pueden haber dos viajes solapados.
context Coche
    inv viajeUnico:
    self.viaje -> forAll(v1,v2 | not(v1 <> v2) or v1.fechaSalida <> v2.
        fechaSalida and (v1.fechaLlegada <= v2.fechaSalida or v2.fechaLlegada
        <= v1.fechaSalida) and (not(v1.fechaLlegada = null and v2.fechaLlegada
        = null)))

-- Invariante 8: Un coche debe de empezar el siguiente viaje en la ciudad
    destino del anterior.
    inv destinoComoOrigen:
    let viajesOrdenados : OrderedSet(Viaje)  = self.viaje -> sortedBy(v | v.
        fechaSalida) in
    viajesOrdenados -> forAll(v1, v2 | viajesOrdenados->indexOf(v1) =
        viajesOrdenados->indexOf(v2) + 1 implies v1.recorrido.destino = v2.
        recorrido.origen)

    -- Invariante 9 (antigua 13): Una ciudad tiene que tener un recorrido con
        otra ciudad de manera obligatoria
context Ciudad
    inv hayRecorridos:
    self.origen -> notEmpty() and self.destino -> notEmpty()


context Coche
    inv empiezaViajeEnDestino:
    (self.viaje -> asOrderedSet() -> sortedBy(fechaSalida) -> last()).recorrido
        .destino = self.ciudad

-- Invariante 11 (antigua 15): Cada coche debe de pasar revision despues de
    matricularse y no antes.
--Un coche no puede viajar antes de su fecha de matriculacion
    inv viajeDespuesdeMatriculacion:
    self.viaje -> forAll(viaje | viaje.fechaSalida > self.fechaMatriculacion)

-- Invariante 12 (antigua 16): Un coche no debe poder comenzar un viaje con la
```

```
        alerta de mantenimiento en true
        inv noViajeEnMantenimiento:
        self.necesitaMantenimiento implies self.viaje -> forAll(v | v.fechaLlegada
            <> null)

        -- Invariante 13 (antigua 17): Un viaje no puede tener una fecha de inicio
            superior a su fecha de llegada
    context Viaje
        inv fechaViajeBienDefinida:
        (self.fechaLlegada = null) xor (self.fechaLlegada > self.fechaSalida)
```

3.5 Modificaciones de las invariantes en el modelo dinámico

3.5.1 Inv4: Si un coche está en revisión, debe de estar en la misma ciudad que el taller

```
context Revision
    inv mismaCiudadqueTallerEnRevision :
    self.fechaFin = null implies
    (self.taller.oclIsKindOf(Oficial) and self.taller.oclAsType(Oficial).ciudad =
        self.coche.ciudad)
    or
    (self.taller.oclIsKindOf(NoOficial) and self.taller.oclAsType(NoOficial).ciudad
         = self.coche.ciudad)
```

Es una modificacioón con respecto al modelo estático ya que en dicho modelo se comparaba el tiempo del clock con la fecha final ya que al ser estático estaba prefijada antes de empezar la revisión, y en el dinámico tenemos la fecha de final a null mientras este en la revisión con eso podemos comprobar si esta en un taller o no y ya con dicha información sabemos si se encuentran en la misma ciudad.

3.5.2 Inv5: Un coche tiene que estar o viajando o en una ciudad

```
context Viaje
    inv viajandoOenCiudad:
    self.fechaLlegada = null implies
    self.coche.ciudad.oclIsUndefined() and self.coche.estaEn->notEmpty()
```

Es una modificación con respecto al modelo estático ya que en dicho modelo se comparaba el tiempo del clock con la fecha final ya que al ser estático estaba prefijada antes de empezar el viaje, mientras que en el dinámico la fecha de llegada es null ya que esta de viaje, y esta es igual al tiempo del clock cuando ha terminado dicho recorrido.

3.5.3 Inv13: Un viaje no puede tener una fecha de inicio superior a su fecha de llegada

```
context Viaje
    inv fechaViajeBienDefinida:
    (self.fechaLlegada = null) xor (self.fechaLlegada > self.fechaSalida)
```

En esta invariante añadimos la primera parte del or con el null para ya que al estar en el viaje dicho atributo se encuentra en null por lo que si no lo añadieramos daría fallo el modelo.

3.6 Añadimos soils???

3.7 b3

Un coche comienza un viaje desde la ciudad en la que se encuentra. Esta operación debe recibir como parámetro el recorrido entre dos ciudades que debe realizar en su viaje. Una operación avanzar que se ejecuta sobre los coches, y que no recibe ningún parámetro. Esta operación debe hacer avanzar el coche el número de kilómetros indicados en su velocidad si el coche está realizando algún viaje. Se debe modelar el paso del tiempo, de modo que un tic del reloj representa el paso de un día, lo cual se debe tener en cuenta a la hora de que los coches puedan avanzar en el viaje que estén realizando.

4 Modelo de Objetos

Ahora se debe desarrollar un modelo de objetos y simularlo.

4.1 Instante 0

Primero se generan las ciudades Málaga, Sevilla y Granada como se indica en el enunciado. Con recorridos entre Málaga y Sevilla de 210 km y entre Sevilla y Granada de 250 km. Se genera un coche con fecha de matriculación en el día 0 que se encuentra en Málaga con una velocidad de 27 km/h.

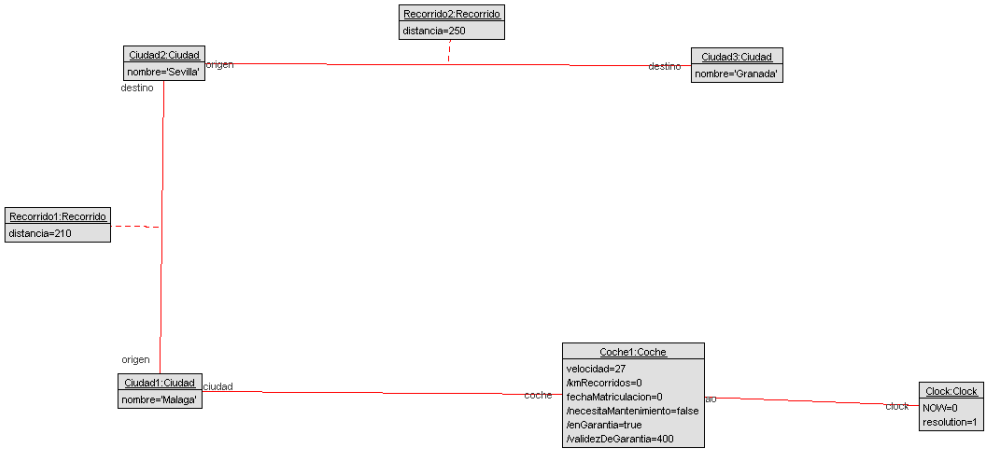

Figura 31: Diagrama en el instante 0

4.2 Instante 1

Tras realizar la foto, se hacen pasar 5 días con el comando `!Clock.run(5)` como indica el enunciado, donde se iniciará un viaje para el coche desde Málaga hasta Sevilla con `!Coche1.inciarViaje(Recorrido1)`.

La siguiente foto se toma cuando el coche llega a Sevilla, en el día 13, ya que si empieza el viaje en el día 5 y tiene que recorrer 210 km a 27 km por día, tarda 7,7 días. Luego, hay que hacer pasar 8 días usando `!Clock.run(7)` seguido de un `!Clock.tick()`.

En nuestro sistema, se ha considerado que el coche tenga velocidad 0 una vez termine un viaje.

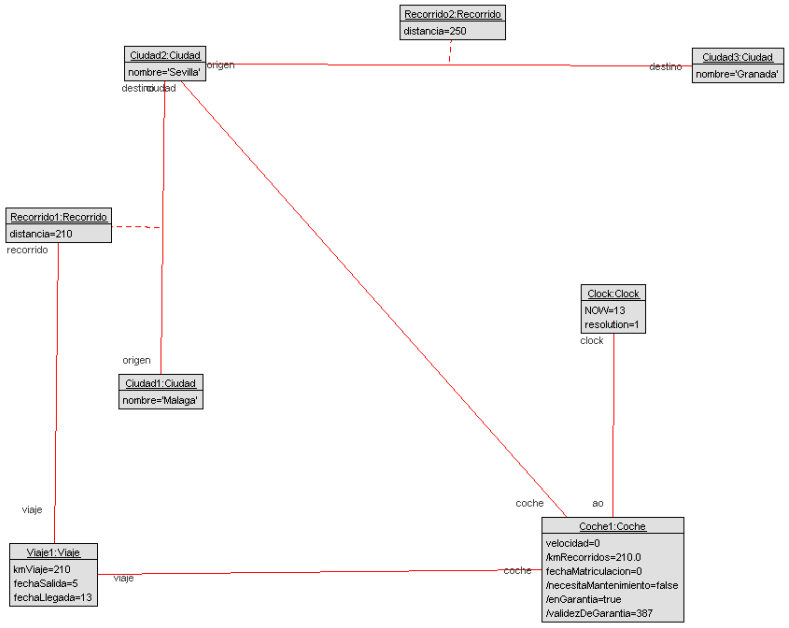

Figura 32: Diagrama en el instante 1 (llegada a Sevilla)

4.3 Instante 2

Se ha restablecido el atributo de velocidad nuevamente a 27, con el comando `!Coche1.acelerar(27)`, para que pueda continuar su nuevo viaje a Granada.

Para llegar a Granada, el coche tiene que recorrer 250 km. A 27 km por día, esto toma 9,26 días. Así que se hacen pasar 10 días con `!Clock.run(10)`.

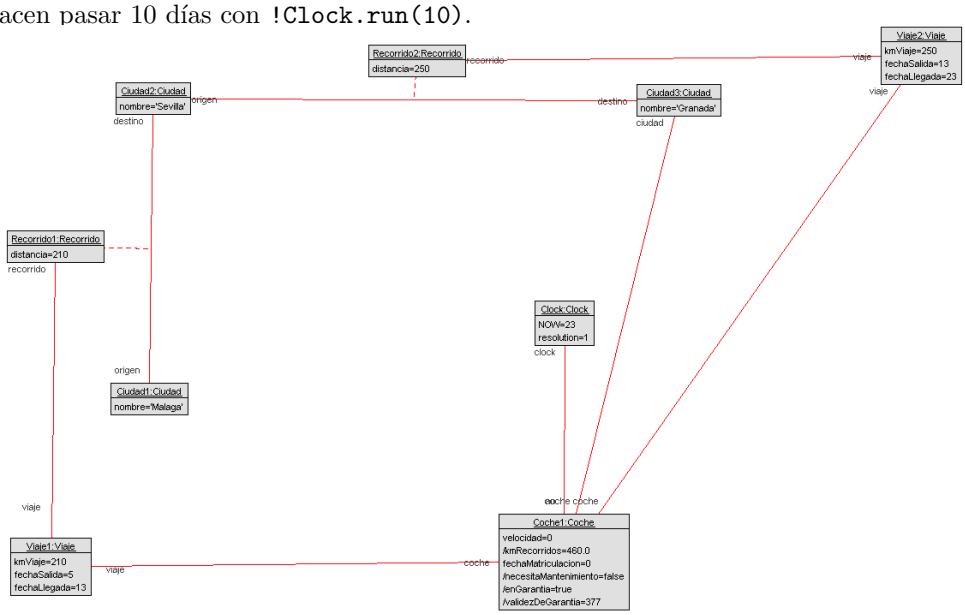

Figura 33: Diagrama en el instante 2 (llegada a Granada)

4.4 Código SOIL para la Simulación

A continuación se presenta el código SOIL necesario para reproducir el modelo conceptual y la simulación descrita:

Listing 3: Codigo SOIL Modelo Objetos

```
!new Clock('Clock')
!Clock.resolution := 1

!new Ciudad('Ciudad1')
!Ciudad1.nombre := 'Malaga'
!new Ciudad('Ciudad2')
!Ciudad2.nombre := 'Sevilla'
!new Ciudad('Ciudad3')
!Ciudad3.nombre := 'Granada'

!new Recorrido('Recorrido1')
!Recorrido1.origen := Ciudad1
!Recorrido1.destino := Ciudad2
!Recorrido1.distancia := 210

!new Recorrido('Recorrido2')
!Recorrido2.origen := Ciudad2
!Recorrido2.destino := Ciudad3
!Recorrido2.distancia := 250

!new Coche('Coche1')
!Coche1.fechaMatriculacion := 0
!Coche1.velocidad := 27
!Coche1.ciudad := Ciudad1

-- Instante 0

!Clock.run(5)
!Coche1.iniciarViaje(Recorrido1)

-- Instante 1 (Sevilla)
-- !Clock.run(7)
-- !Clock.tick()

-- Se reestablece la velocidad
-- !Coche1.acelerar(27)
-- !Clock.run(10)

-- Instante 2 (Granada)
```