SASE 2015 SIMPOSIO ARGENTINO DE SISTEMAS EMBEBIDOS

UML minimalista y el desarrollo evolutivo basado en modelos

Ing. Leandro Francucci (francuccilea@gmail.com)

Vortex

12 de Agosto de 2015

Introducción a MDD

MDD (Modelo-Driven Development) en Real-Time Embedded Systems (RTES)

¿Qué hay de especial en los RTES?

- Su software interactúa directamente con dispositivos electrónicos e indirectamente con algunos mecánicos, y químicos, entre otros. Esto se denomina *Codesarrollo*.
- Frecuentemente, se diseña y escribe el software dependiente del hardware, aún cuando este no existe.
- Alguno son de naturaleza reactiva, cuya respuesta a eventos externos es estricta en el tiempo. Estos se denominan ***Sistemas Reactivos***.
- Usualmente, se construyen con las computadoras más baratas.
- Muchos, son sensibles al costo de fabricación recurrente, con lo cual utilizan procesadores con menos recursos.
- Las herramientas para su depuración no son lo suficientemente sofisticadas y estables.
- Por lo general, es difícil visualizar mensajes de error y diagnóstico.

¿Qué hay de especial en los RTES?

- Algunos, deben lidiar con limitaciones en el consumo de energía.
- Otros no sólo deben cumplir con las restricciones temporales, sino también con la robustez y la fiabilidad. En ciertos casos, si fallan tienen el potencial de causar mucho daño.
- Usualmente, deben funcionar ininterrumpidamente, durante largos períodos de tiempo, y puede que se desempeñen en entornos adversos y hostiles.

Un sistema de "tiempo-real" no significa un sistema "rápido"
Real-time != Fast-time

Nuestro objetivo y desafío

Todas estas dificultades se traducen en mayor complejidad, lo que implica más tiempo, más esfuerzo y (a menos que tengamos cuidado) más defectos.

También le exigimos alta calidad, manteniendo la funcionalidad deseada a un costo y duración no muy elevado.

Por lo tanto, nuestro objetivo es mostrar herramientas y enfoques que mitiguen estos inconvenientes y así ***DISMINUIR*** los ***TIEMPOS***, el ***COSTO*** y la ***COMPLEJIDAD*** del desarrollo de embedded software de *alta calidad*, en todas las etapas de su ciclo productivo.

Principios para un desarrollo eficiente

- Nuestro objetivo principal: *desarrollar software que funcione*
- Siempre deberemos medir el progreso contra el objetivo, no contra la implementación
- Es decir, nuestra principal medición del progreso es el software que funciona
- La mejor manera de no tener defectos en el software es no incorporarlos desde el principio
- La realimentación continua es trascendental
- Planifique, siga y adapte
- La causa principal del fracaso del proyecto es ignorar el riesgo
- Es esencial la atención continua en la calidad
- *Modelar es fundamental*

Prácticas para un desarrollo eficiente

- Construir incrementalmente.
- Usar planificación dinámica.
- Minimizar la complejidad.
- Modelar con un propósito.
- Usar frameworks.
- Probar continuamente que el sistema bajo desarrollo es correcto.
- Crear el software y las pruebas al mismo tiempo.
- Aplicar patrones inteligentemente.
- Manejar interfaces para facilitar la integración.
- Usar la asociatividad modelo-código.

Modelar es fundamental
¿Porqué modelar? I

- Cuando hablamos de desarrollo basado en modelos (MDD), estamos comparándolo implícitamente con el desarrollo centrado en el código. ¿Cuáles son las razones de MDD?.
- El código fuente es una vista altamente detallada de la estructura del sistema.
- Esto hace extremadamente difícil observar el sistema desde diferentes puntos de vista, como los aspectos arquitectónicos, la funcionalidad, o el comportamiento.
- Los sistemas grandes son difíciles de comprender, y por lo tanto es caro y tedioso agregarle recursos humanos.
- Las partes arquitectónicas no se representan explícitamente, deben inferirse leyendo cientos de líneas de texto.
- El diseño está incorporado en el código, y por lo tanto es difícil asegurar que la intención del mismo es la representada.

Modelar es fundamental
¿Porqué modelar? II

- Los modelos permiten explícitamente y claramente mostrar la funcionalidad y la intención del diseño en un lenguaje de mayor nivel de abstracción que el código fuente.
- Con UML, un lenguaje de modelado, puede mostrarse como están estructuradas las partes, como se comportan, y como se interrelacionan.

Los ***modelos mejoran la productividad y la eficiencia***, permitiendo a los desarrolladores, enfocarse en las cosas correctas, en el nivel de abstracción adecuado y en el momento adecuado.

UML en Real-Time Embedded Systems

- UML (Unified Modeling Language) es un estándar de facto para el modelado de software.
- Permite crear modelos precisos, ejecutables y verificables.
- Permite la interoperabilidad de herramientas visuales de modelado.
- La notación es gráfica y simple.
- Los sistemas complejos pueden desarrollarse fácilmente con tres diagramas esenciales: **clases, máquinas de estados y secuencias**.
- Soporta todas las cuestiones necesarias para modelar RTES

Diagramas de estructuras

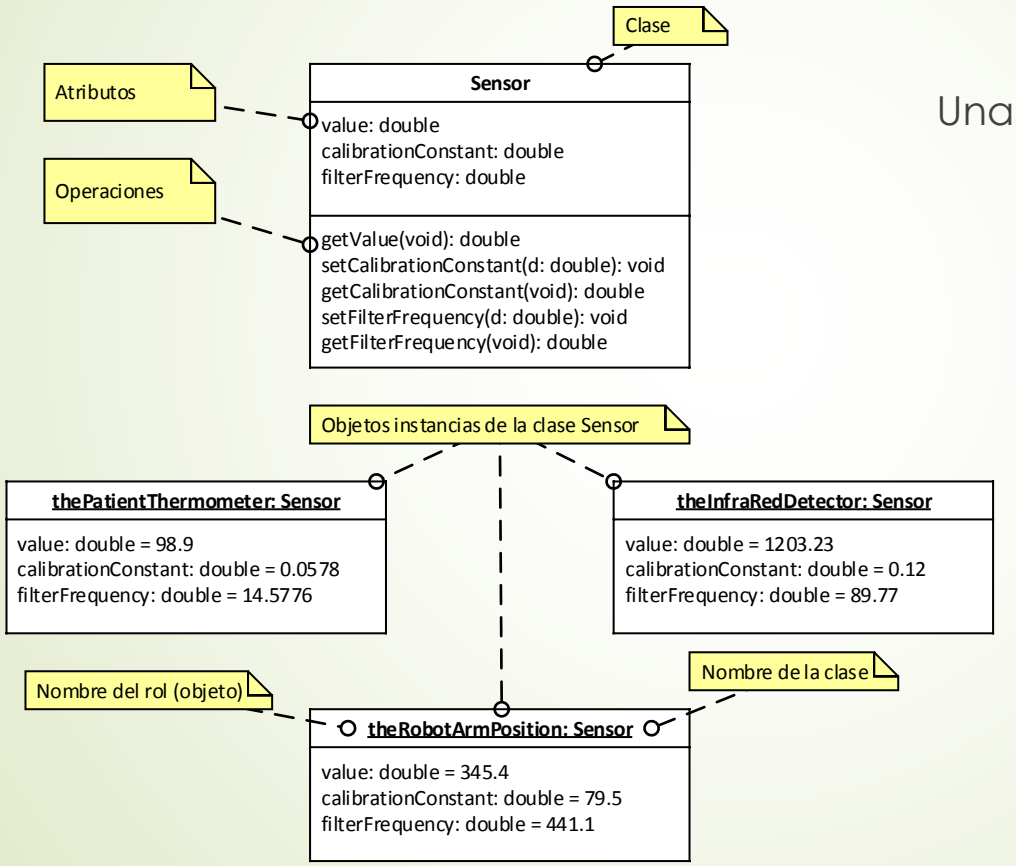
Objetos

- En su forma más simple, un *objeto* es una estructura de datos que también provee servicios que actúan sobre esos datos.
- Únicamente existe en tiempo de ejecución, es decir, mientras el sistema está en ejecución, ocupa alguna posición de memoria en algún momento específico.
- Los datos del objeto se almacenan en sus *atributos*.
- Los servicios que actúan sobre esos datos se denominan *métodos*, los cuales se invocan por clientes de ese objeto u otros métodos del objeto.
- Las máquinas de estados y los diagramas de actividad pueden imponer secuencias específicas de estos servicios.

Clases

- Una *clase* es una especificación de un conjunto de objetos, que comparten una estructura y un comportamiento específico.
- Los objetos se dicen ser *instancias* de una clase.
- Durante la ejecución del sistema, una clase puede tener muchas instancias pero cada objeto es una instancia de una única clase.
- También puede especificar una máquina de estados, la cual coordina y gestiona la ejecución de sus comportamientos primitivos (denominados *acciones*).
- Frecuentemente, las acciones son invocaciones de los métodos definidos por la clase, en un conjunto admisible de secuencias.
- Al conjunto de atributos y métodos se los denomina *características* de una clase
- Permiten el ***encapsulamiento***, la ***herencia*** y el ***polimorfismo***

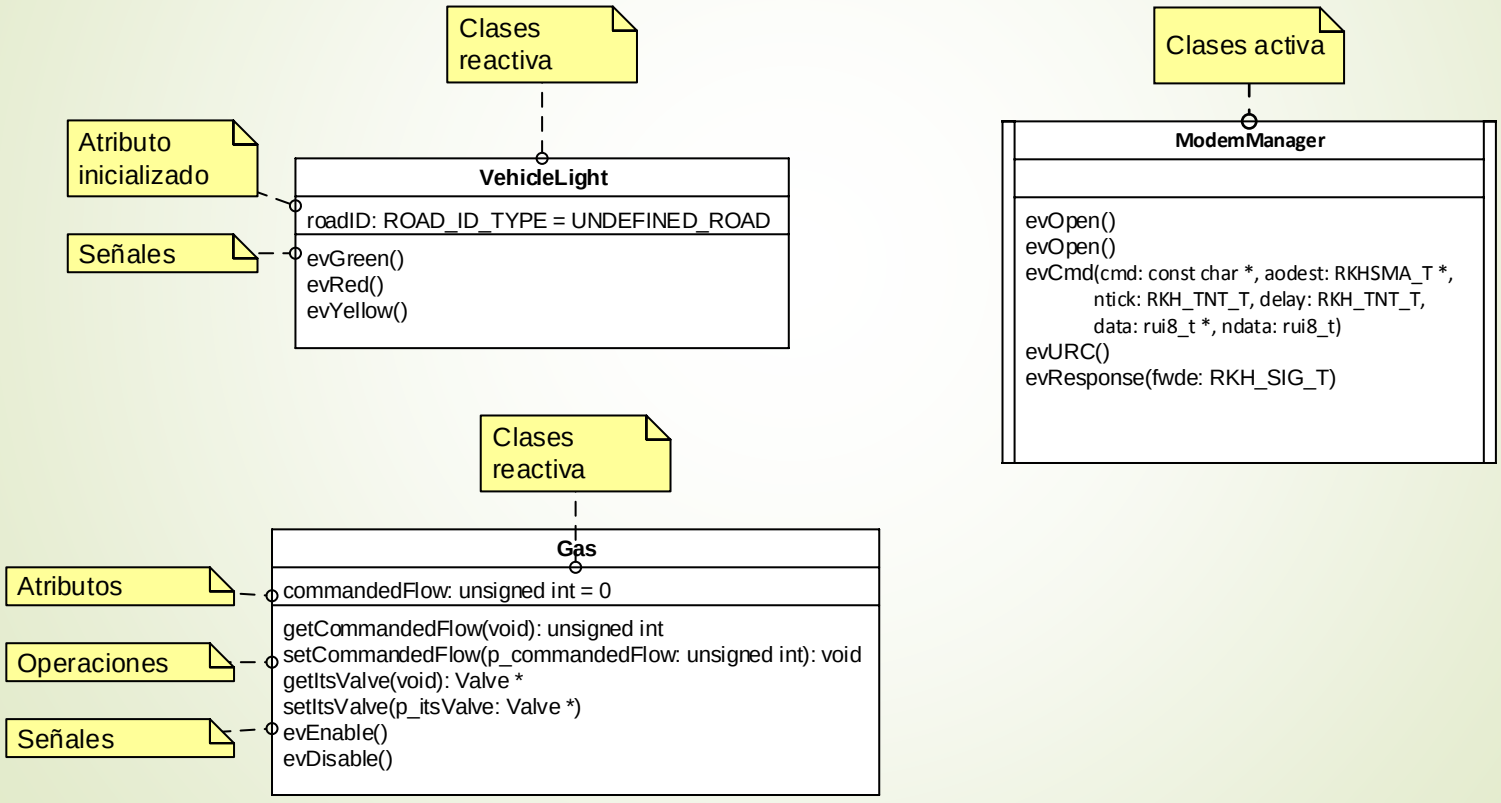
Clases y objetos en UML



Una clase especifica:

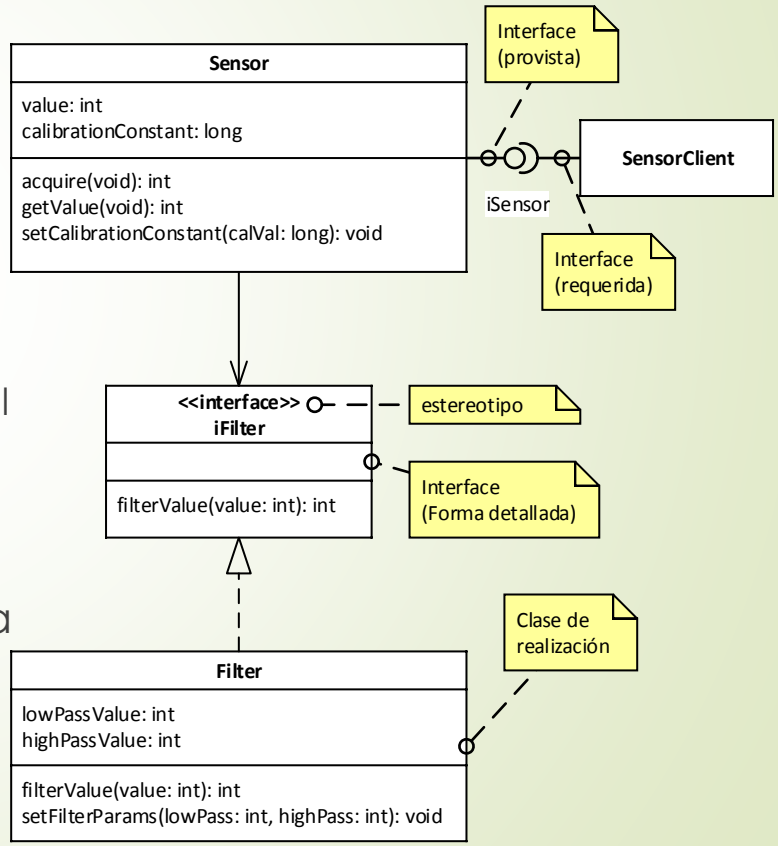
- **Métodos**
  - Implementación de operaciones
  - Típicamente manipulan los atributos de la clase
- **Atributos**
  - Valores tipificados conocidos por la clase
- **Máquina de estados**
  - Estados, transiciones y acciones
- **Interfaces**
  - Operaciones realizadas por los métodos de la clase

Clases y objetos en UML
Ejemplos



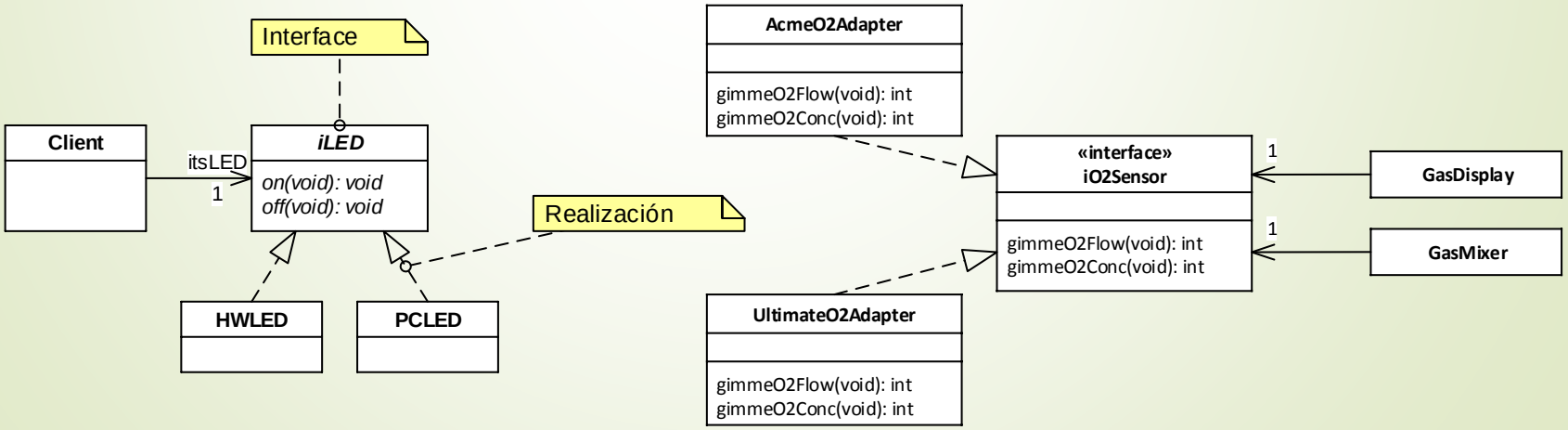
Interfaces

- Una *interface* es una colección de servicios.
- Los servicios puede ser:
  - *Operaciones*: servicios sincrónicos o asincrónicos, invocados por clientes.
  - *Recepción de señales*: servicios asincrónicos, invocados al enviar una señal asincrónica al objeto que acepta ese evento.
- El nombre del servicio, su valor de devolución y sus parámetros se denomina *firma*.
- Una clase que cumple con una *interface* provee un método para cada operación y una recepción de evento para cada señal. Se dice que la clase *realiza* la interface.



Interfaces

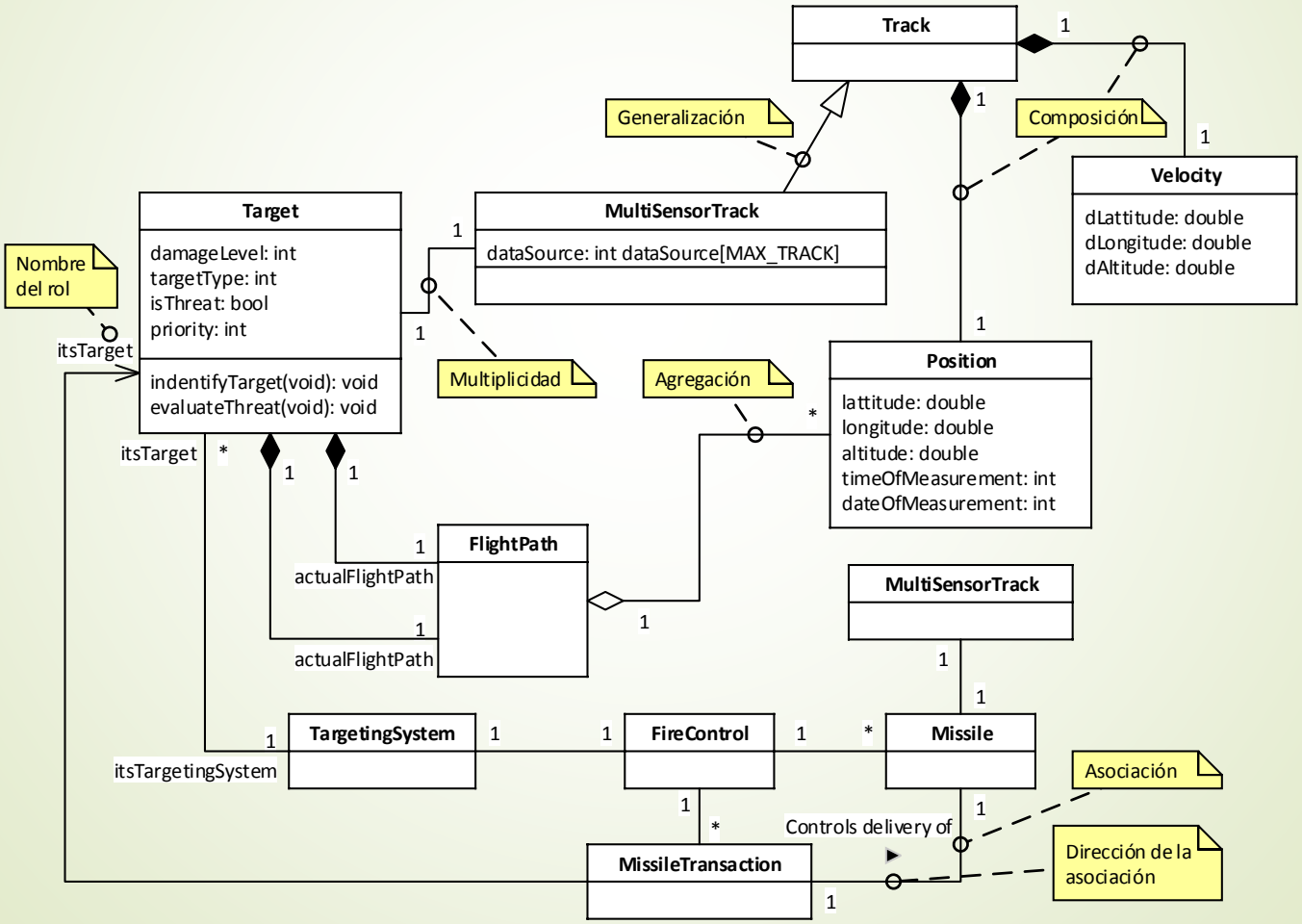
- Los métodos y las recepciones de eventos, incluyen las líneas de código que implementan dicho servicio.
- Así, las interfaces separan la especificación de los servicios de su implementación.
- No son *instanciables*.



Relaciones

- Para lograr el comportamiento deseado del sistema, los objetos trabajan de manera colectiva, y para ello deben relacionarse de alguna manera.
- UML define diferentes tipos de relaciones entre clases. Las más importantes son:
  - Asociación
  - Generalización
  - Dependencia

Relaciones
Ejemplo

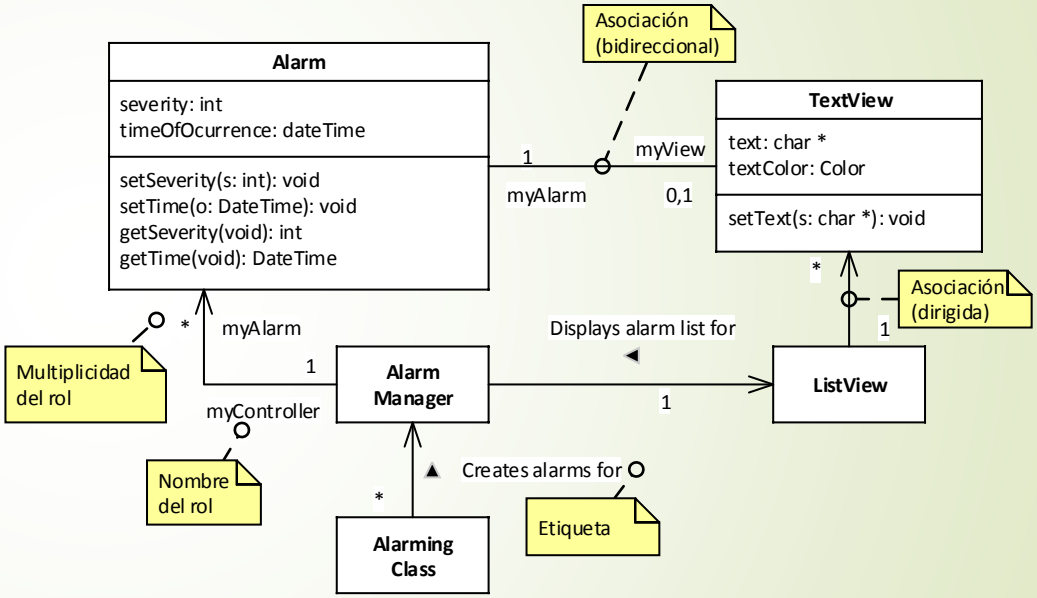


Relaciones
Asociación

- Una asociación es una relación entre clases en tiempo de diseño, que especifica que, en ejecución, las instancias de esas clases pueden tener un *enlace* (relación navegable entre objetos, a través de la cual pueden invocarse los servicios). Un enlace es una instancia de una asociación.
- UML identifica tres tipos de asociación:
  - Asociación: es una especificación de los conductos, que permiten a los objetos enviarse mensajes, en ejecución. Tienen diferentes aspectos a distinguir: *nombre de roles*, etiquetas, multiplicidad, navegabilidad.
  - Agregación
  - Composición

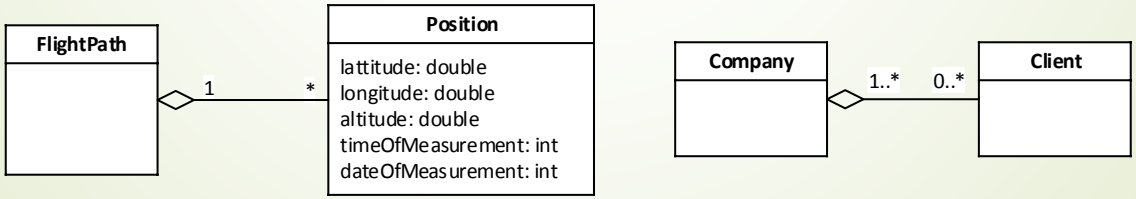
Relaciones
Asociación simple

- Asociación: es una especificación de los conductos, que permiten a los objetos enviarse mensajes en ejecución.
- Las distinguen diferentes aspectos como: *nombre de roles, etiquetas, multiplicidad, navegabilidad*.



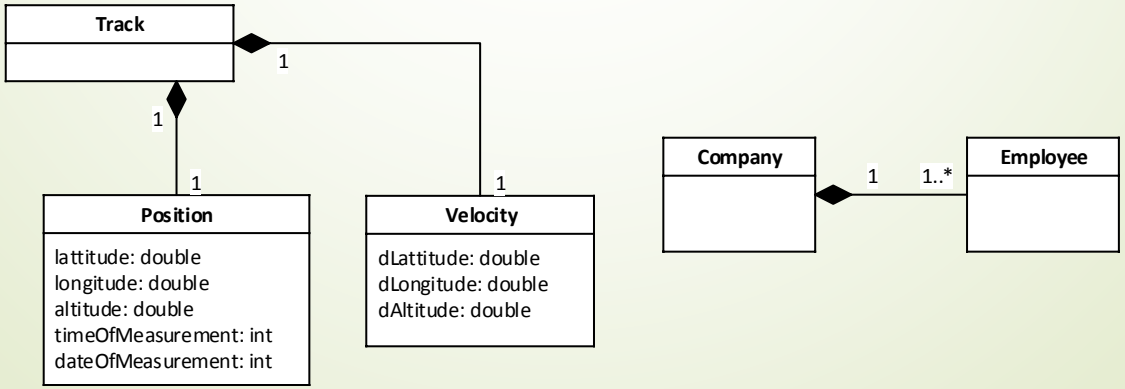
Relaciones
Agregación

- Es un tipo especializado de una asociación, que indica que existe una relación "whole-part" entre dos objetos.
- Esta relación "whole-part" es relativamente débil, ya que no se realiza ninguna declaración relacionada con la dependencia del ciclo de vida o la responsabilidad de la creación/destrucción.
- Generalmente, en diseño e implementación se la trata de igual manera que una asociación.



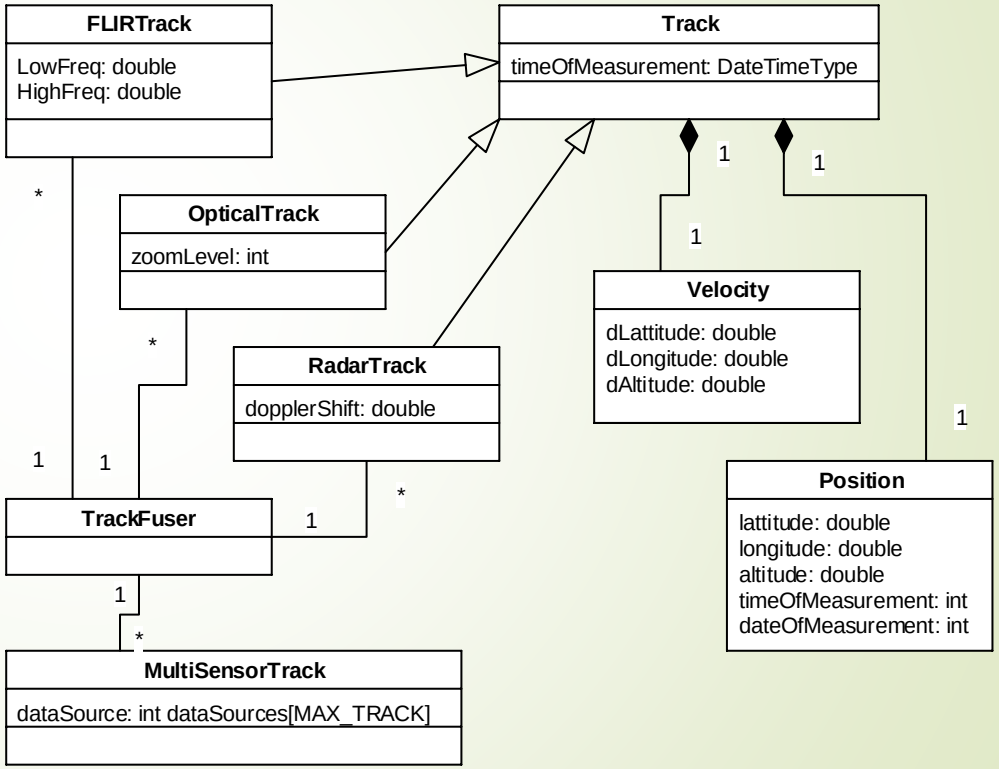
Relaciones
Composición

- Es una forma fuerte de agregación, en la cual "el todo" (también conocido como *composite* o "compuesto") posee la responsabilidad de la creación y destrucción de sus partes.
- Dado que la creación y destrucción de las partes está explícitamente asignada al compuesto, la multiplicidad que le corresponde a este, en la relación de composición, es *siempre* 1.
- En objeto puede tener varios propietarios de agregación, sin embargo puede tener a lo sumo un propietario compuesto.



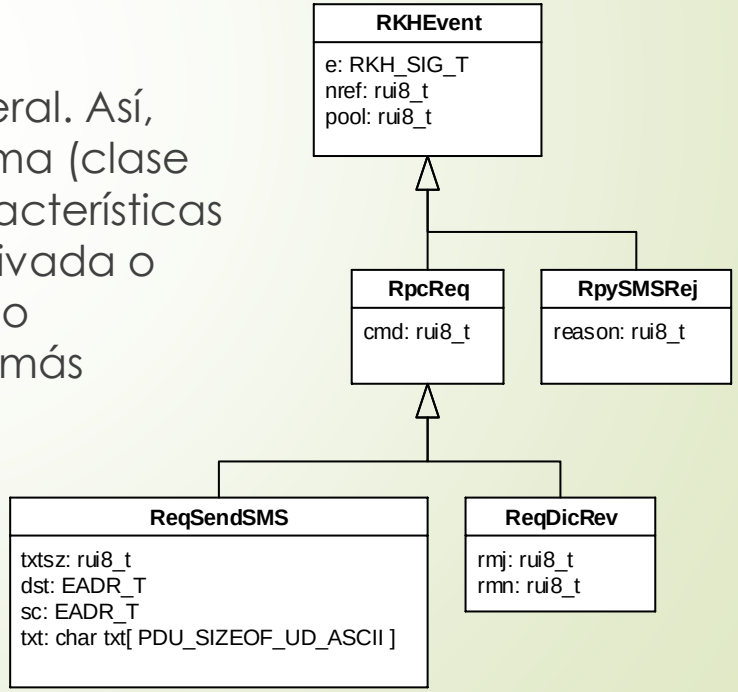
Relaciones
Generalización

- En UML la relación de generalización significa que una clase define un conjunto de características que son especializadas o extendidas en otra clase.
- Significa *herencia* y *sustituibilidad*.



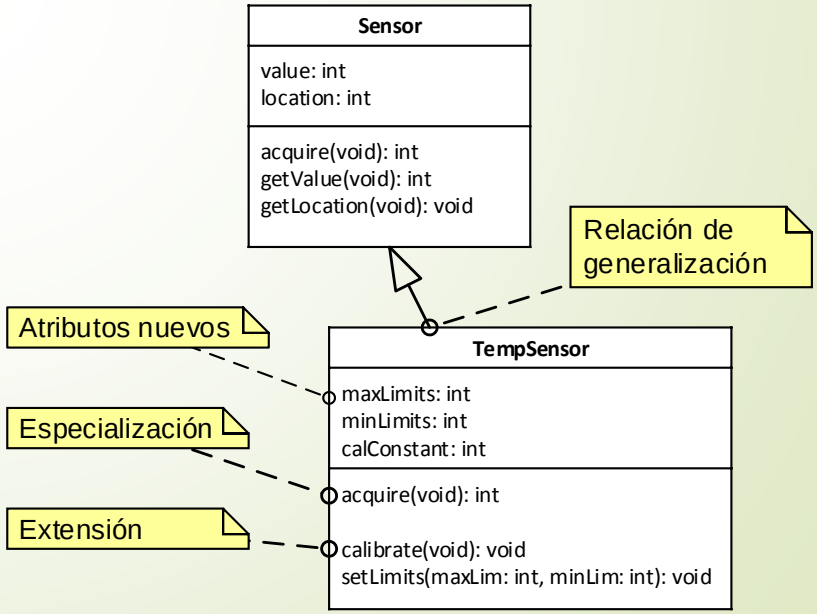
Relaciones
Generalización - Herencia

- **Herencia**
  - Es una manera de representar clases que son un "*tipo especializado de*" otra clase.
  - La clase especializada hereda las características de la clase más general. Así, todas las características de esta última (clase base o *superclase*) son también características de la clase especializada (clase derivada o subclase), no obstante está permitido *especializar* y/o *extender* una clase más general.
  - Por lo tanto, una de las principales ventajas de la herencia es la capacidad de reutilizar tanto el diseño como el código



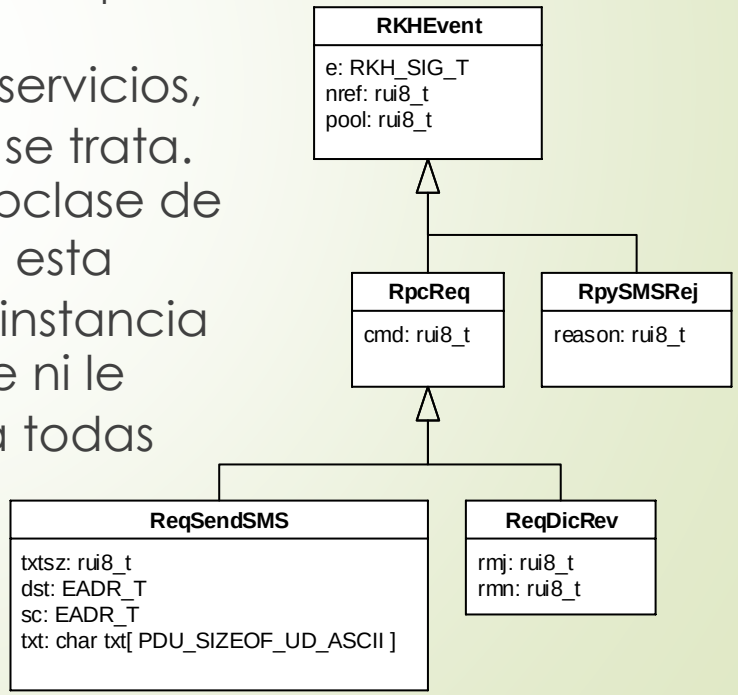
Relaciones
Generalización

- Especializar
  - Una subclase puede redefinir las operaciones provistas por la superclase. Esto se denomina *polimorfismo*, que permite a una misma operación o máquina de estados, representar diferentes comportamientos, de acuerdo al contexto.
- Extender
  - Una subclase define nuevas características respecto de su superclase.



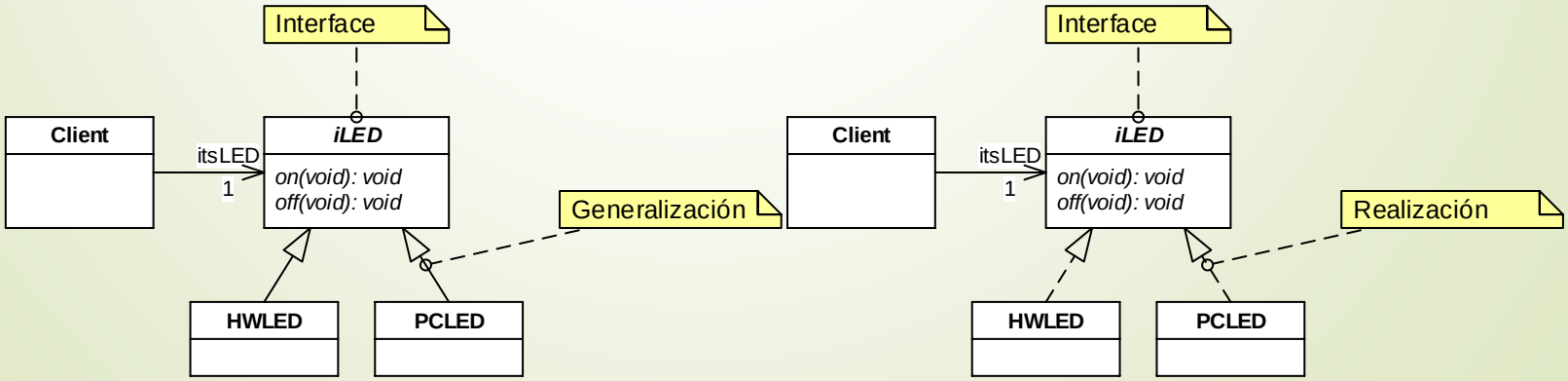
Relaciones
Generalización - Sustituibilidad

- **Sustituibilidad**
  - Significa que una instancia de una subclase es libremente sustituible siempre que haya una instancia de una superclase.
- Ej: la clase `RKHEvent` puede proveer servicios, como devolver de que tipo de señal se trata. Debido a que `ReqSendSMS` es una subclase de `RKHEvent`, en ejecución el cliente de esta última puede tener un enlace a una instancia de `ReqSendSMS`, sin embargo no sabe ni le importa, ya que ReqSendSMS hereda todas las capacidades de sus superclases.



Relaciones
Generalización

- La generalización se utiliza como un medio para asegurar el complimiento de una interfaz.
- Además, es la manera más común de implementar interfaces en lenguajes como C o C++.
- A su vez, simplifica el modelo de clases, ya que un conjunto de características comunes a un determinado número de clases, puede abstraerse en una única superclase, en lugar de redefinir la misma estructura.



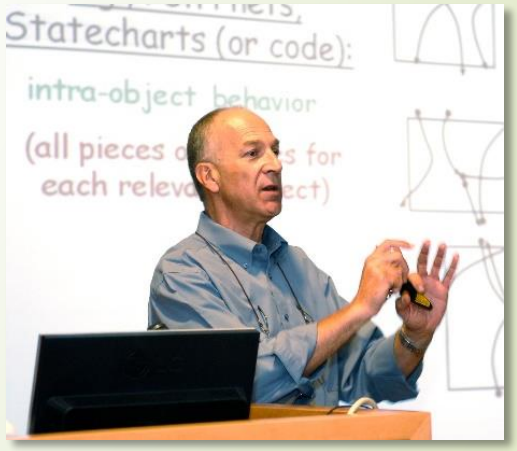
Diagramas de comportamiento

Máquina de estados
Elementos del modelo

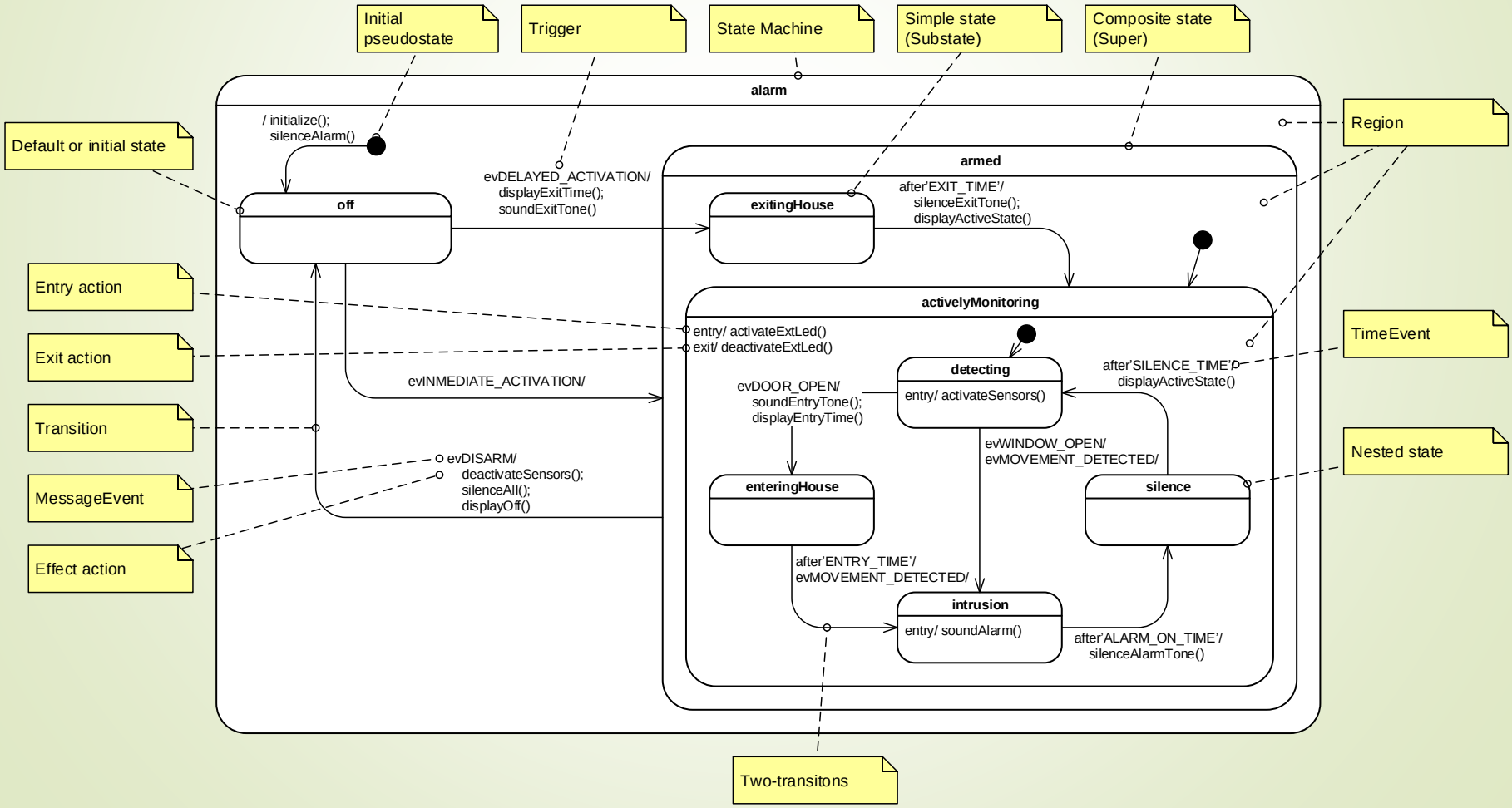
- Máquina de estados (basadas en Statecharts)
- Región
- Vértice
  - Estado: Simple, Compuesto, Submáquina
  - Referencia de punto de conexión
  - Estado Final
  - Pseudoestado: Initial, DeepHistory, ShallowHistory, Join, Fork, Junction, Choice, EntryPoint, ExitPoint, Terminate
- Transición
- Evento
  - Message Event: SignalEvent, CallEvent
  - TimeEvent
  - ChangeEvent

Statecharts

- Constituye un **formalismo visual** para describir los estados y transiciones de manera modular, permitiendo la agrupación y concurrencia, fomentando la capacidad de moverse fácilmente entre diferentes niveles de abstracción.
- Respecto de las tradicionales FSM, básicamente incorpora:
  - Anidamiento jerárquico de estados
  - Concurrencia
  - Transiciones condicionadas
  - Acciones de entrada y salida de estados
  - Actividades
  - Pseudoestados
- La notación y semántica fue definida por *Dr. David Harel*, luego adoptada por UML
- Surgió como una manera *clara* y *precisa* para describir el comportamiento dinámico de un sistema de vuelo de un avión de combate Israelí, específicamente IAI Lavi.

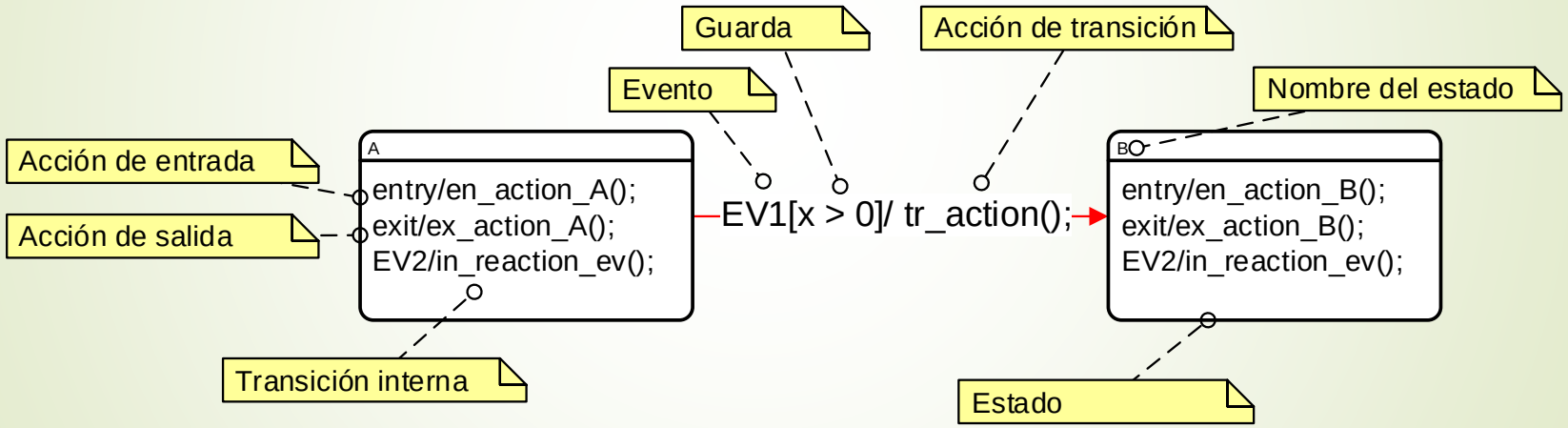


Máquina de estados



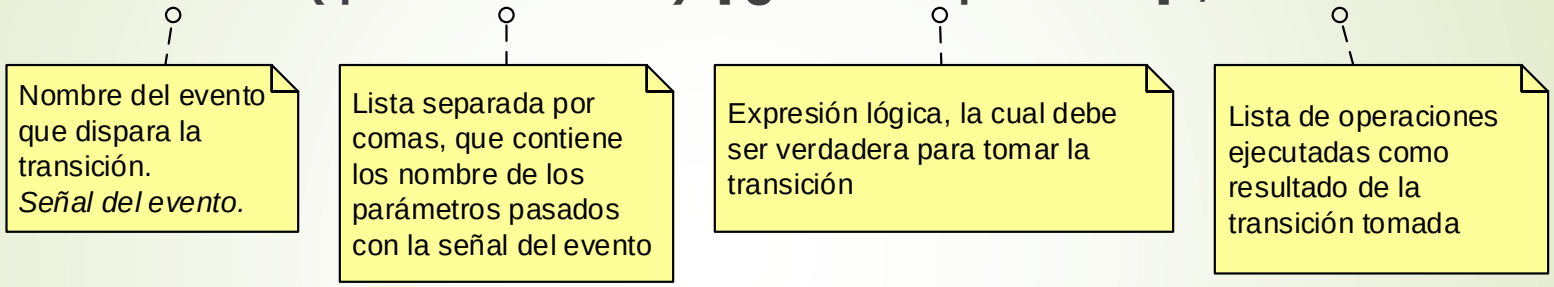
Máquina de estados
Elementos básicos

Estados, transiciones, eventos, guardas y acciones



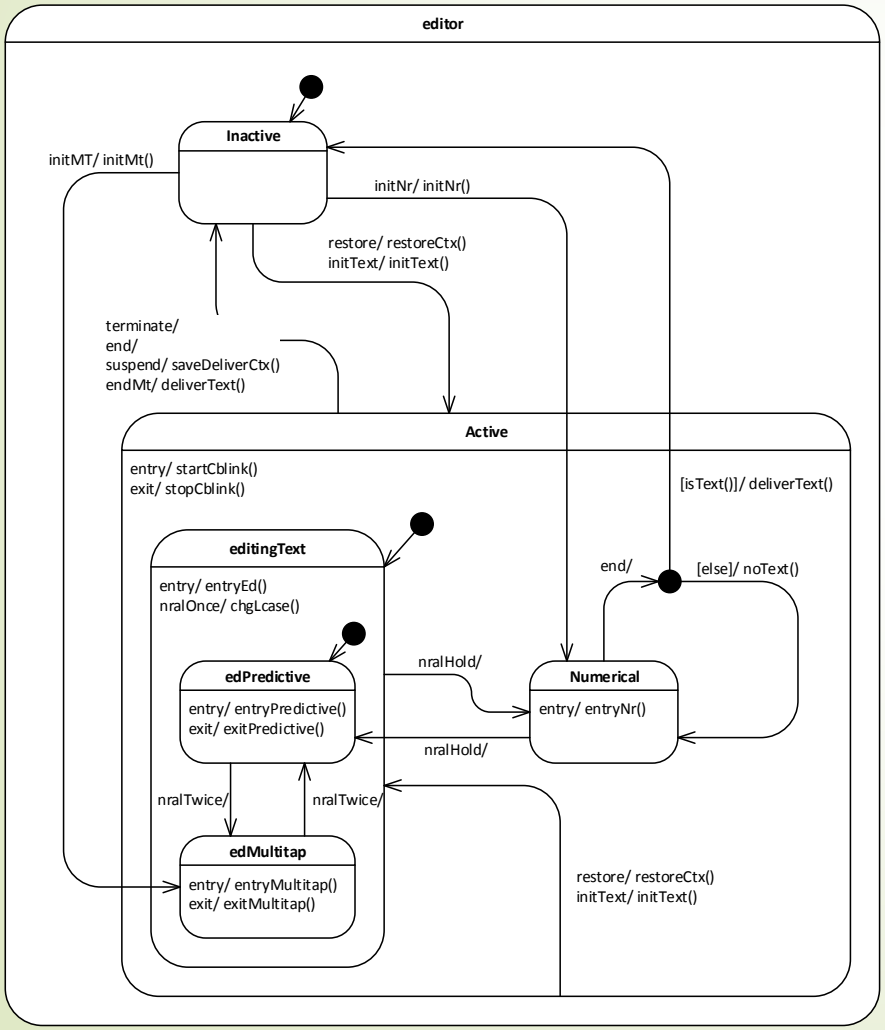
Notación de la transición

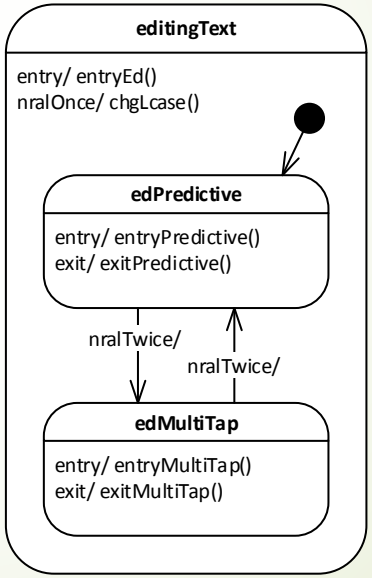
event-name’(‘parameter-list’)’’[‘guard-expression’]’’/’action-list

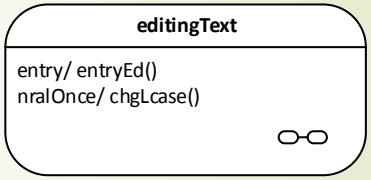


- Ante un evento recibido, la máquina recorre los estados, ascendiendo en jerarquía, desde el estado más interno hasta encontrar la transición que dispara, caso contrario se descarta.
- Algunos ejemplos de transiciones permitidas:
  1) E1
  2) E1[x < my_sensor >value]
  3) E1(x: float) / y = filter(x * 100);
  4) E1(x: float, y: long) [abs(x) > abs(y ^ 2) > 0] / result = x / y;
  5) E1 / y = sqrt(x^2 + y^2); z = 2*y – 3;

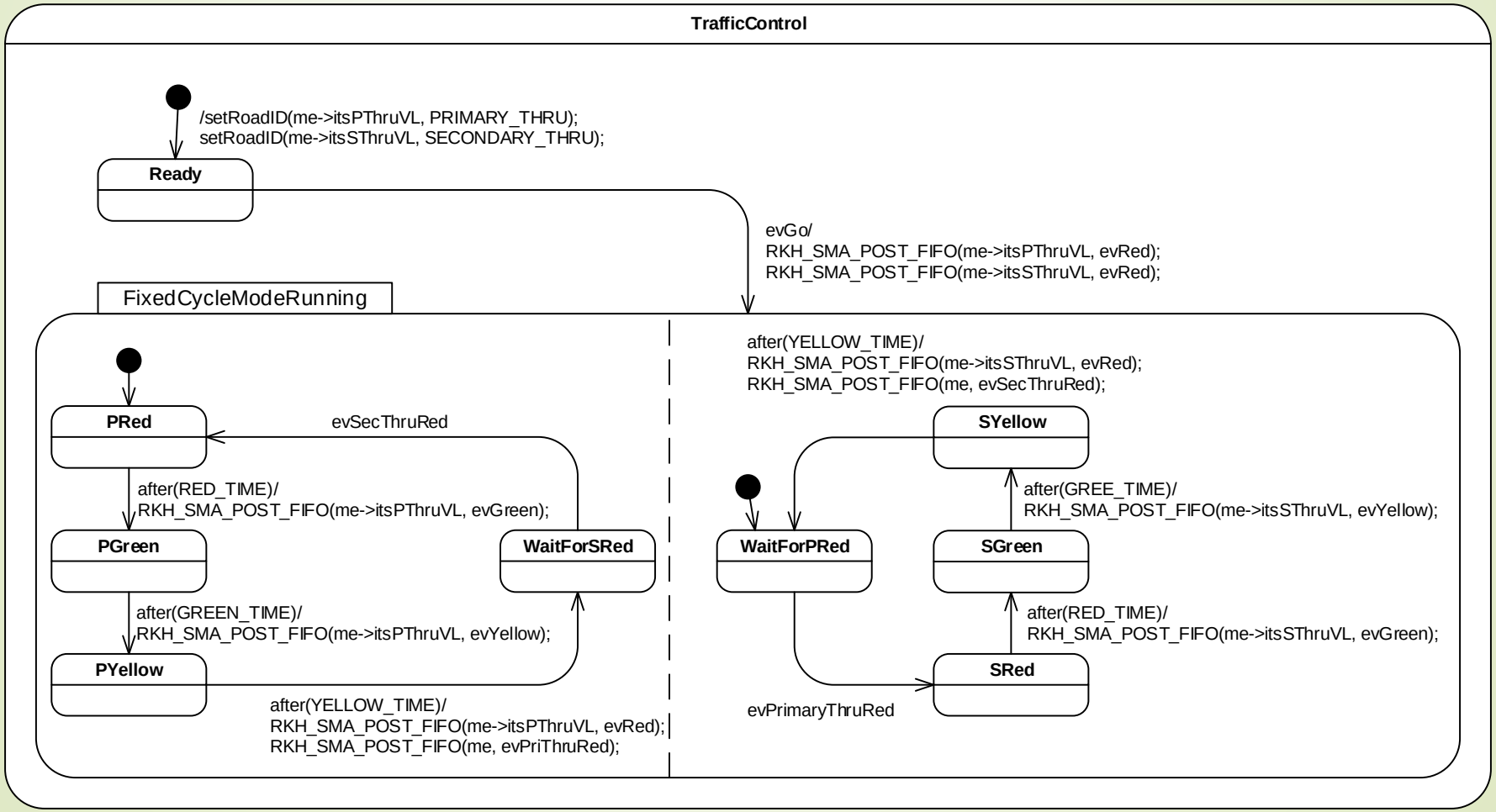
Estado
Ejemplo estado compuesto simple



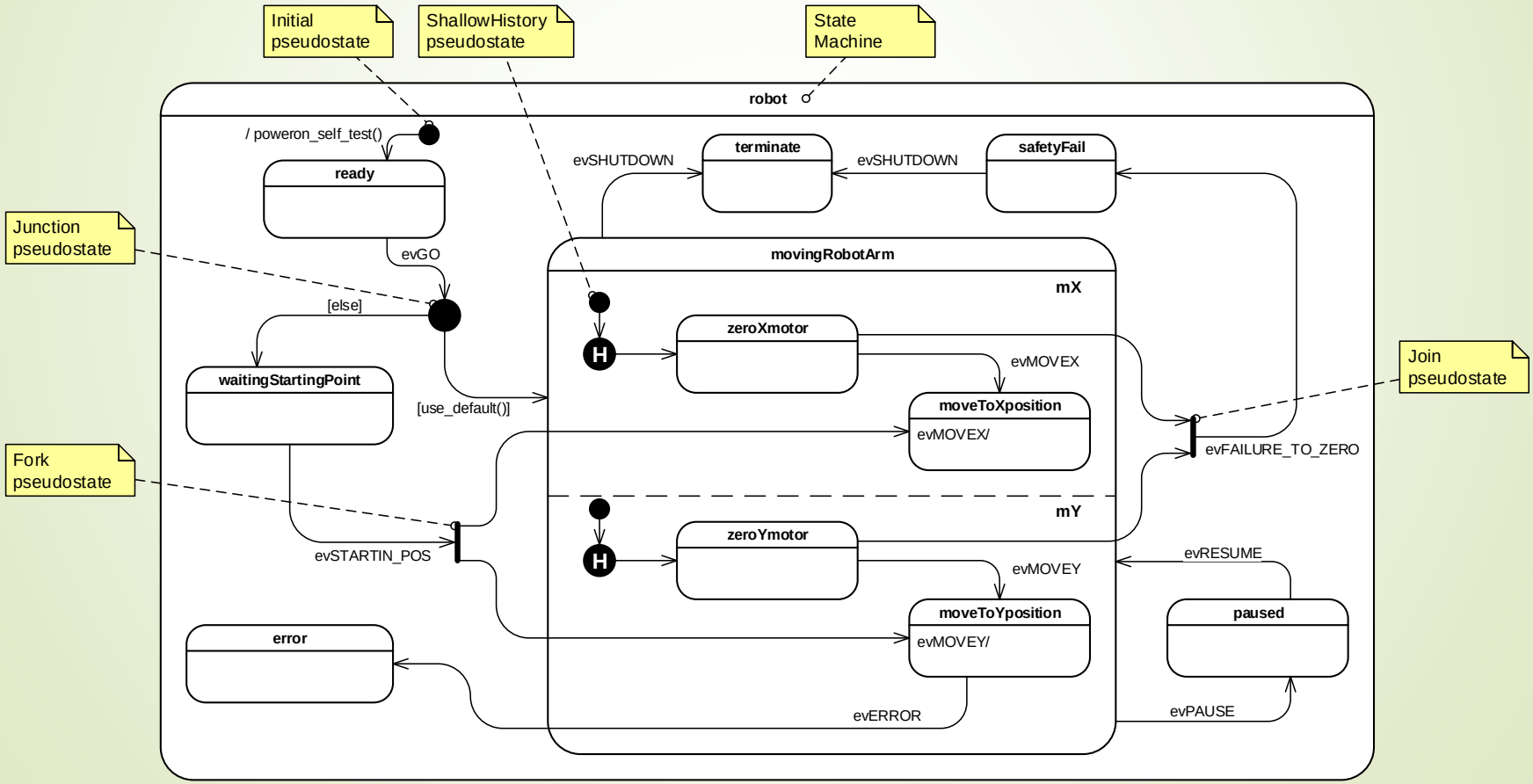




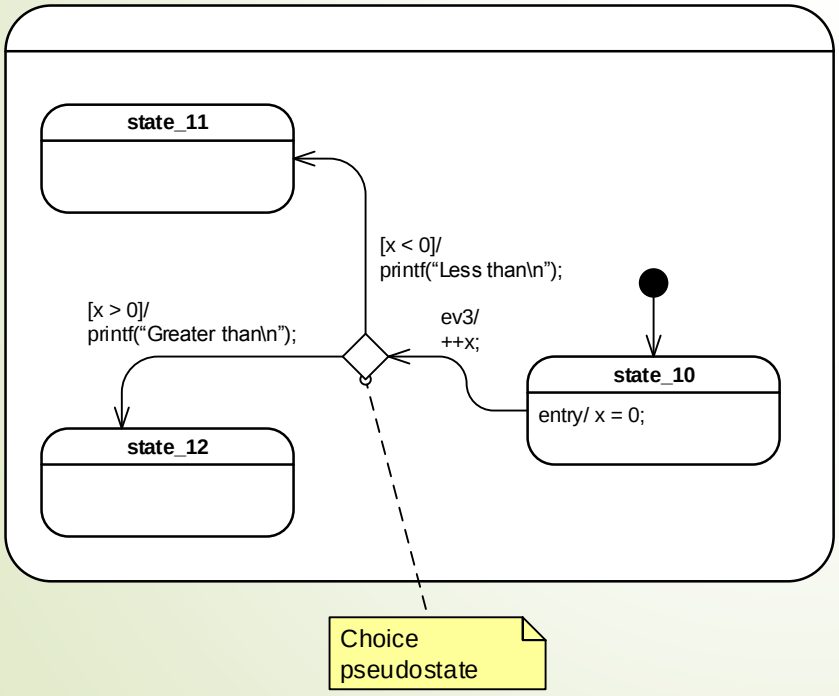
Estado
Ejemplo estado ortogonal

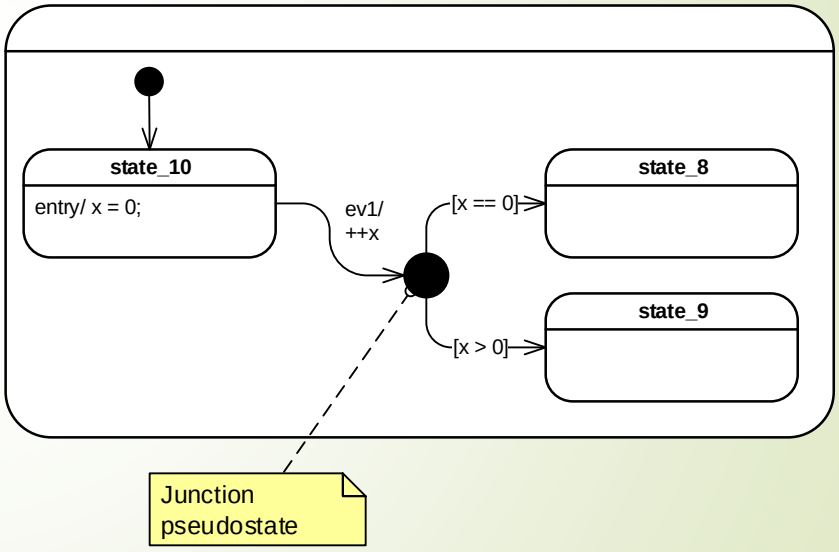


Pseudoestado

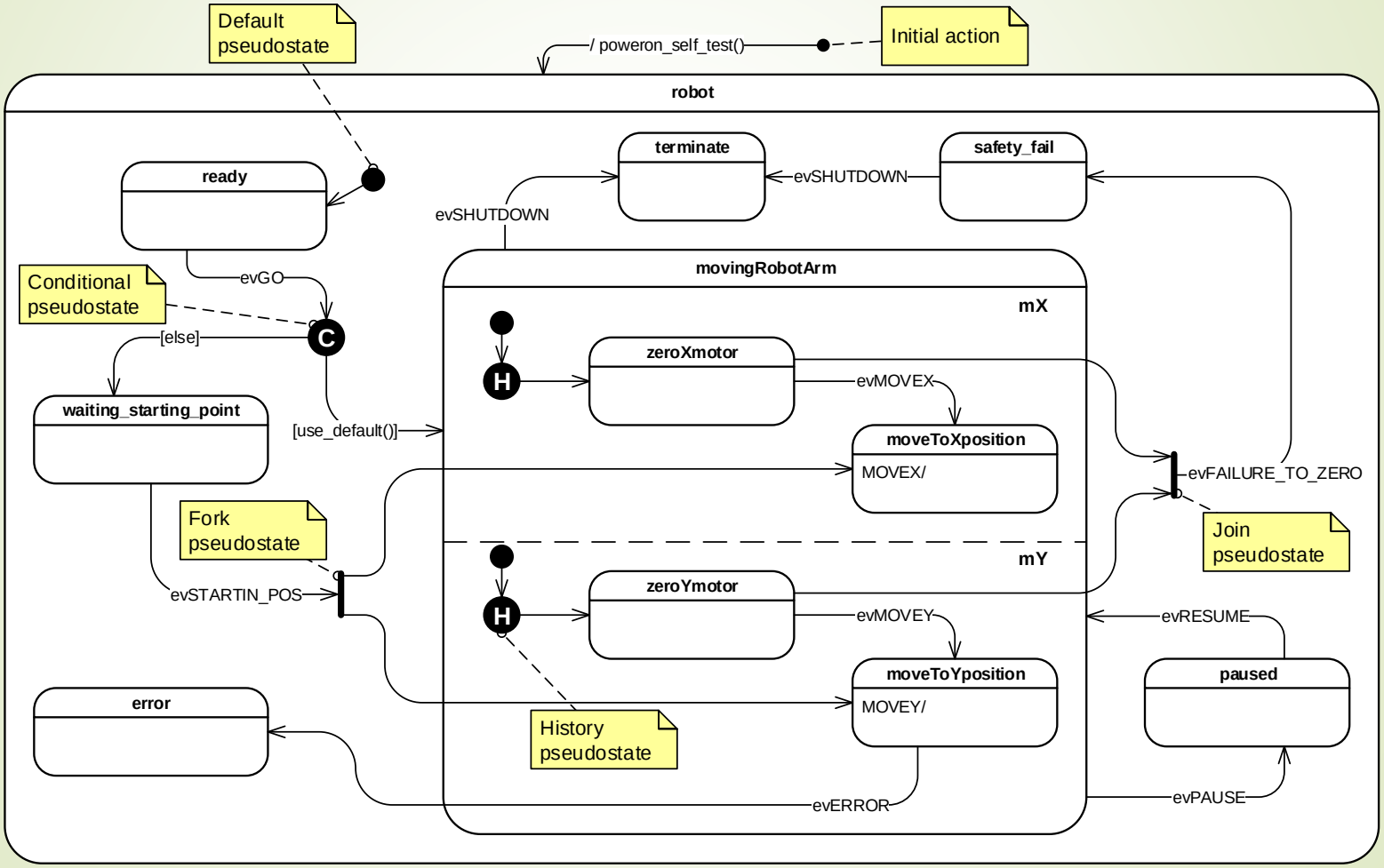


Pseudoestado
Junction & Choice





Pseudoestado
Join & Fork



Al igual que los estados ortogonales, son complejos de implementar

Diagramas de interacción

Interacciones

- UML modela como *interacciones* el comportamiento colaborativo de múltiples entidades trabajando en conjunto.
- Para representar estas interacciones UML básicamente define tres diagramas:
  - Diagrama de comunicación
  - Diagrama de secuencias
  - Diagrama temporal
- El diagrama de secuencias es por mucho el más utilizado.

Diagrama de secuencias

- Representa objetos interactuando en función del tiempo.
- Las líneas verticales se denominan *líneas de vida*, que representan al objeto en cuestión. Estos pueden recibir y transmitir mensajes, los cuales se muestran por las flechas que van de una línea de vida a otra.
- Muestra una determinada ejecución, bajo ciertas condiciones, de alguna porción del sistema, denominada *escenario*.

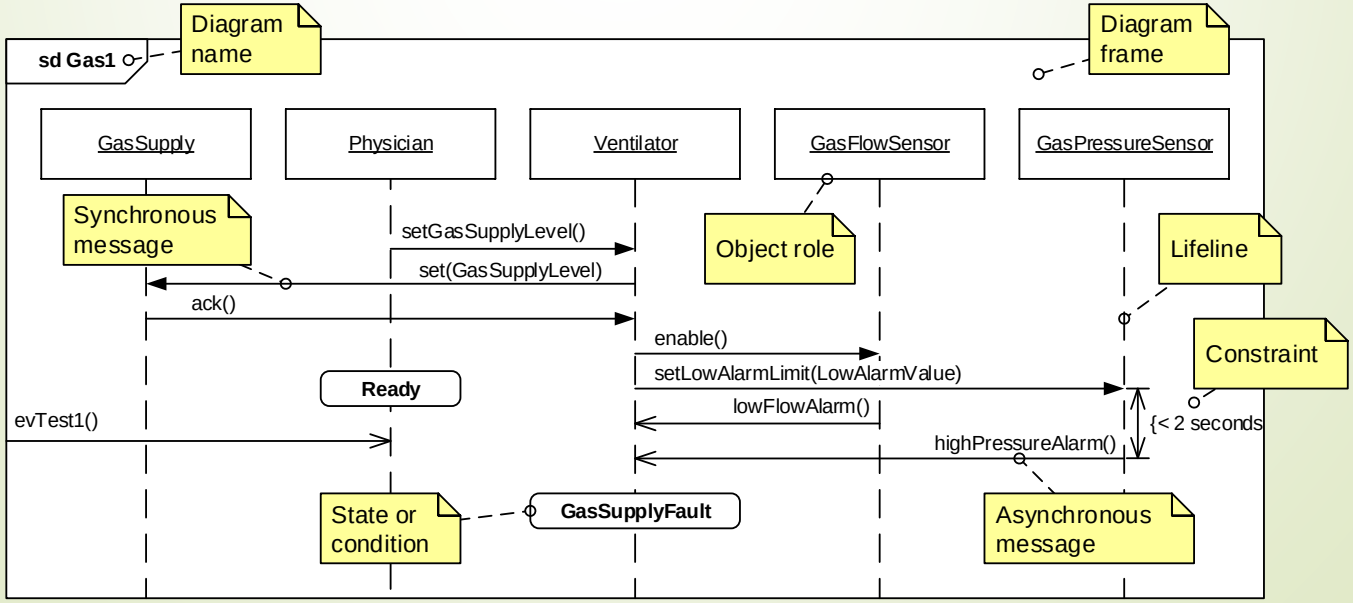


Diagrama de secuencias

- Los mensajes pueden ser sincrónicos o asincrónicos. Opcionalmente, los mensajes sincrónicos pueden devolver un mensaje.
- En las líneas de vida pueden mostrarse condiciones o estados del objeto. Como así también, restricciones.

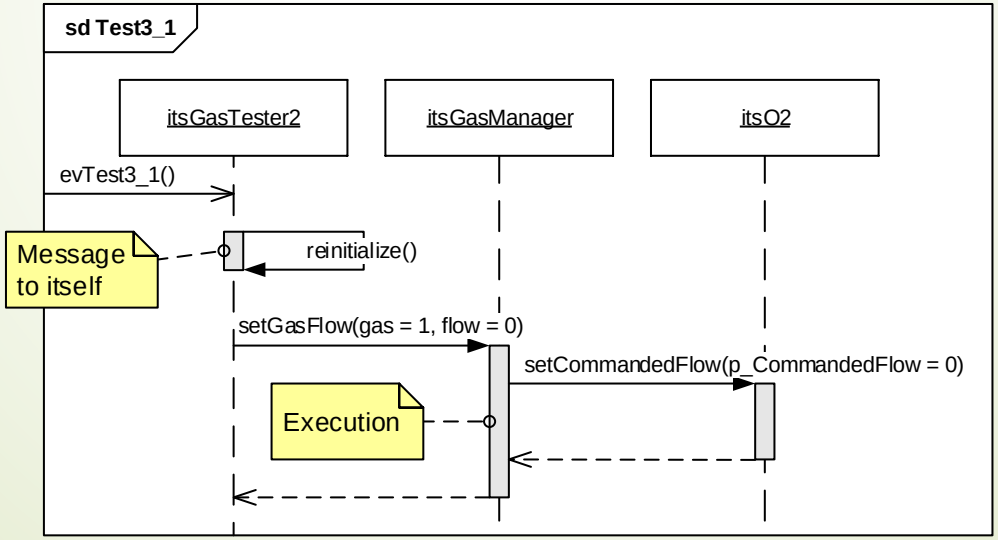


Diagrama de secuencias
Fragmentos combinados

- Un *fragmento combinado* se utiliza para agrupar mensajes de forma tal de mostrar flujo condicional en un diagrama de secuencias.
- UML define diferentes tipos de interacción para los fragmentos combinados, de acuerdo a su operador, como ser *Weak Sequencing* (seq), *Alternative* (alt), *Option* (opt), *Break* (break), *Parallel* (par), *Strict Sequencing* (strict), *Loop* (loop), *Critical Region* (critical), *Negative* (neg), *Assertion* (assert), *Ignore* (ignore), y *Consider* (consider).

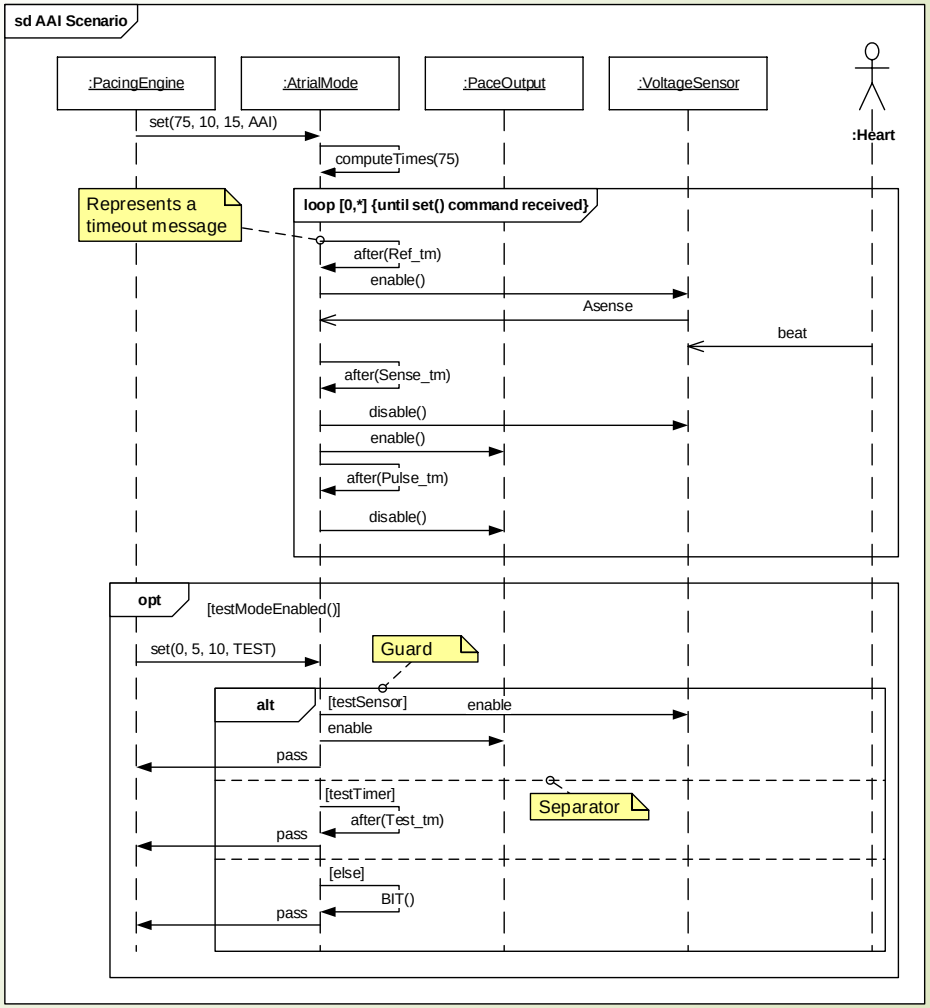


Diagrama de secuencias
Fragmentos combinados - Ejemplo

- par: se utiliza para indicar ejecución concurrente.
- loop: permite modelar una secuencia repetitiva un número de veces.
- critical: indica que debe tratarse como atómico
- assert: representa una aserción
- consider: especifica que mensajes deberían considerarse

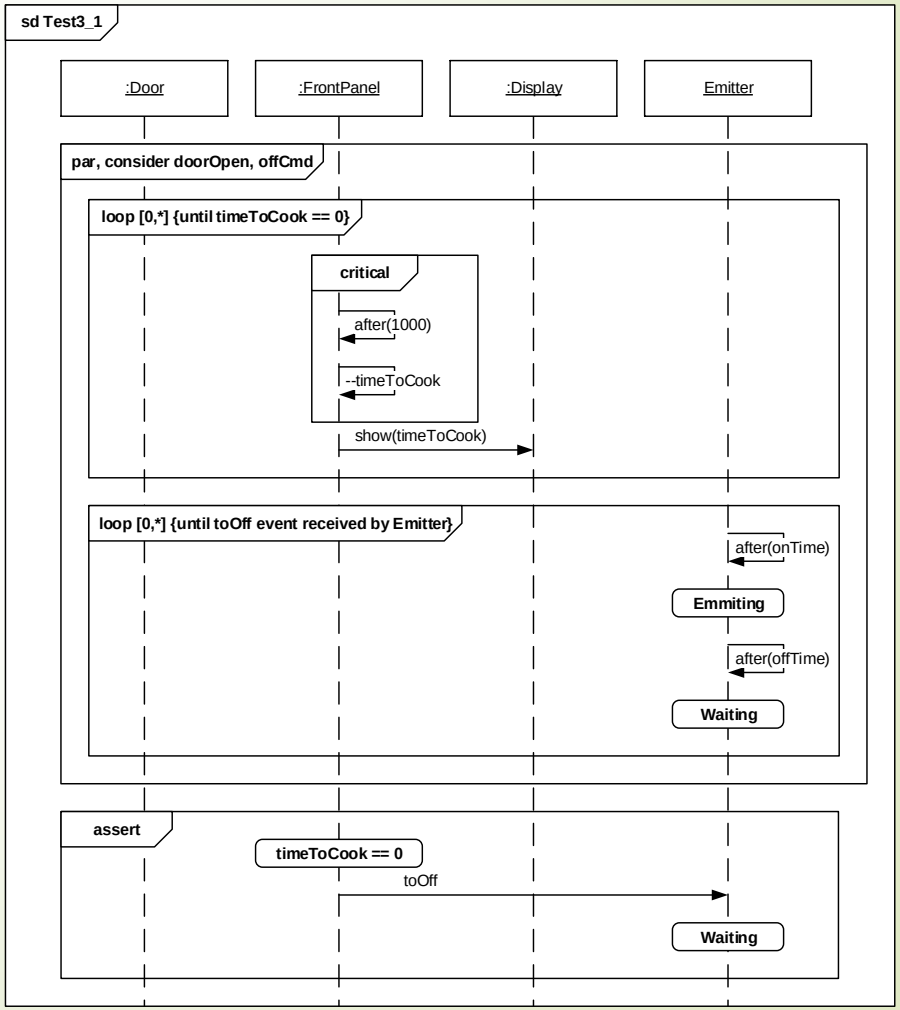


Diagrama de secuencias
Descomposición

- Un diagrama de secuencia puede descomponerse:
  - ***"horizontalmente"***, utilizando un fragmento de interacción con el operador *ref,* o
  - ***"verticalmente"***, estableciendo una referencia desde una línea de vida a otro diagrama de secuencias, que muestra el mismo escenario pero en un nivel de abstracción más detallado.

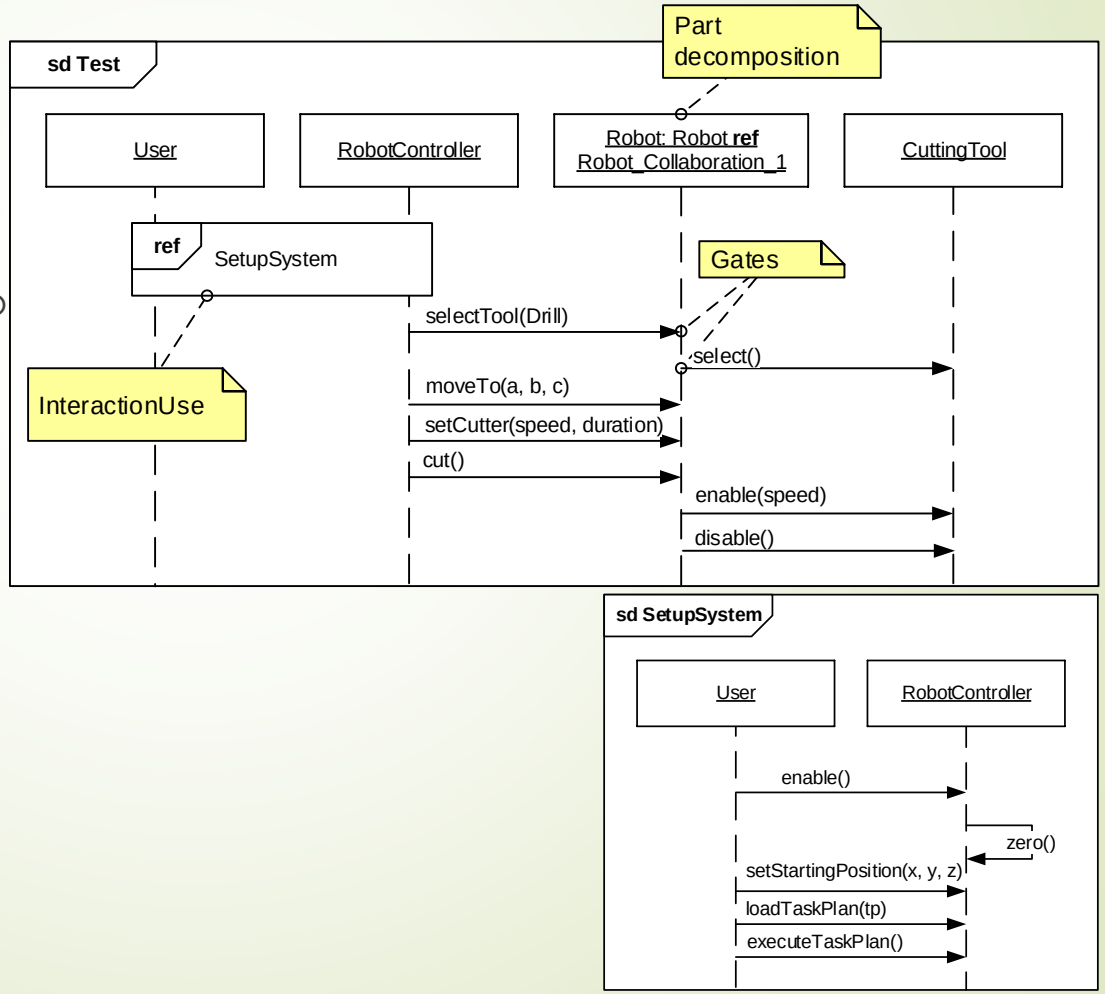
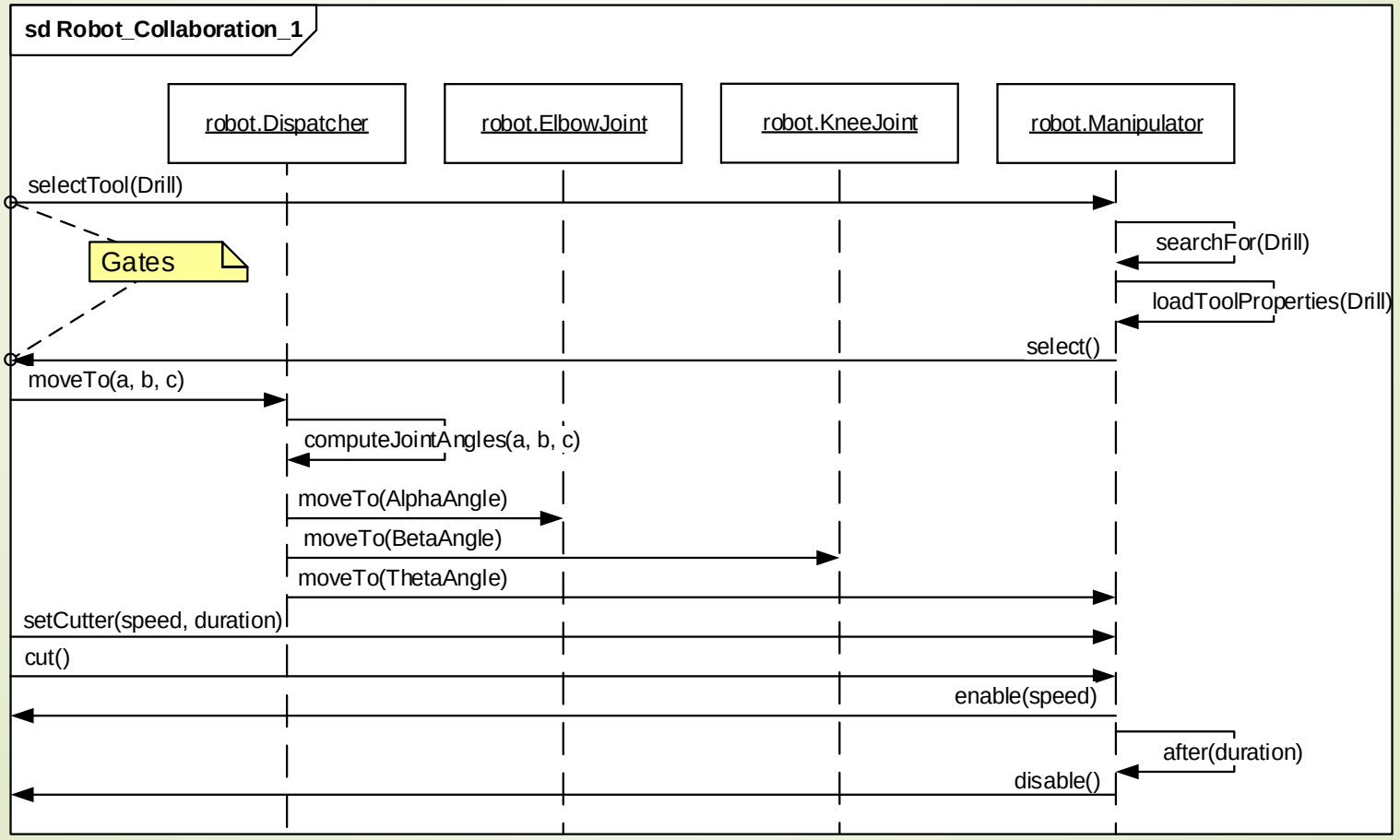


Diagrama de secuencias
Descomposición - Ejemplo

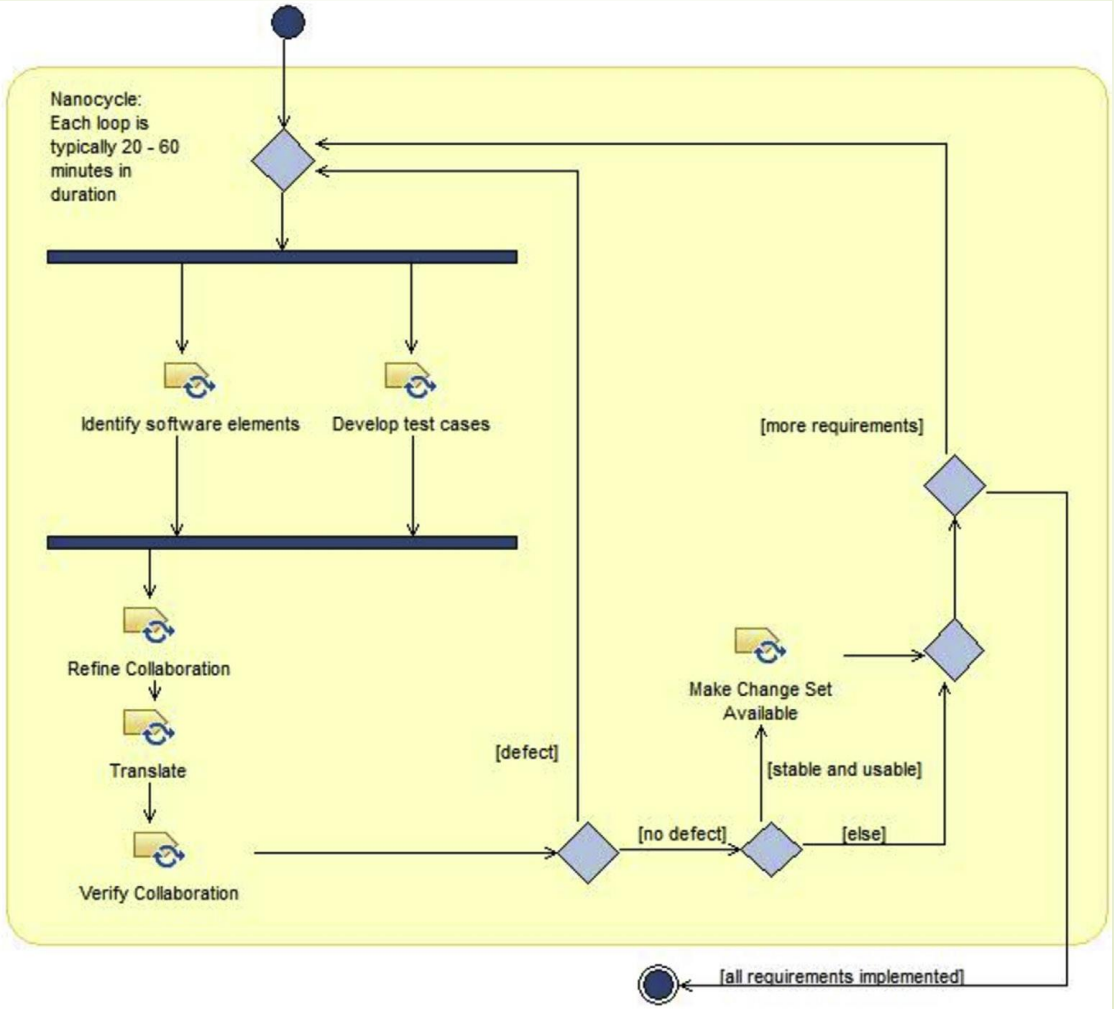


Desarrollo evolutivo basado en modelos

Evolutivo e incremental

- Básicamente el desarrollo se aplica desde la funcionalidad pretendida, organizando los requerimientos funcionales en un conjunto coherente de casos de uso.
- El desarrollo se divide en iteraciones, durante las cuales se realizan una cierta cantidad de casos de uso, de manera evolutiva e incremental.
- Durante una iteración, para cada uno de los casos de uso, identificamos los objetos y las clases y refinamos el modelo, agregando detalles como operaciones, atributos y relaciones (colaboración), logrando un prototipo validado y ejecutable.
- La clave para construir efectivamente la colaboración que realiza el caso de uso es verificar continuamente el software durante su construcción.

Evolutivo e incremental
Workflow



Enfoque TDD

1. Identificamos algunos elementos de software y definimos sus pruebas unitarias
2. Los agregamos en la colaboración
3. Generamos el código de la colaboración (manual o automáticamente)
4. Verificamos (aplicamos las pruebas unitarias)
5. Repetimos hasta lograr el caso de uso

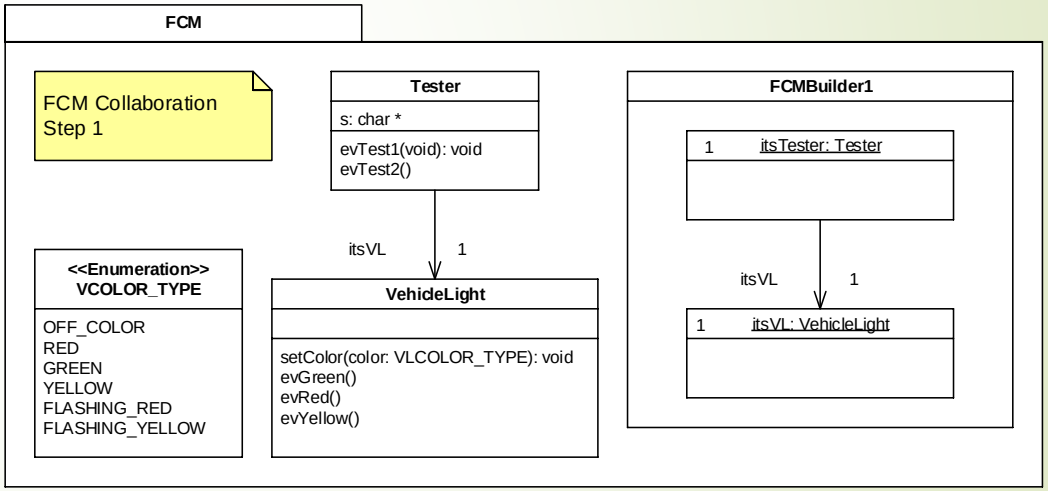
Refinar implica generar código, ejecutar, depurar y producir versiones ejecutables de la aplicación (prototipo incremental)

Ejemplo: Modo cíclico fijo de un semáforo (FCM)

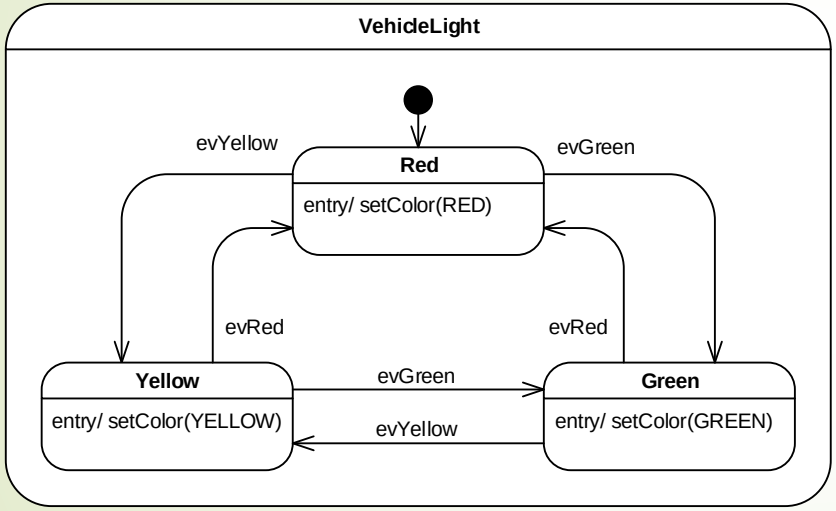
- Para realizarlo se propone realizar un ciclo mediante el siguiente plan:
  - **Paso 1**: hacer funcionar un luz
  - **Paso 2**: agregar un controlador y verificar que una única luz recorra sus estados posibles
  - **Paso 3**: agregar luces de paso (verde) para la calle principal y la secundaria. Lograr su control
  - Paso 3.1: modificar la máquina de estados del controlador tal que utilice un estado ortogonal
  - **Paso 4**: agregar sensores de automóviles para los carriles de giro
  - **Paso 5**: agregar el segundo carril de giro
  - **Paso 6**: agregar los botones de peatones e identificar donde se procesarán sus solicitudes.
  - **Paso 7**: manejar las solicitudes de los peatones

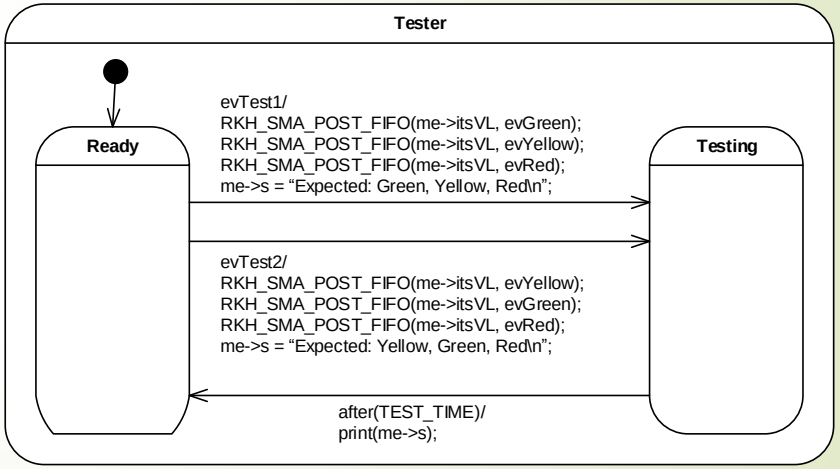
Ejemplo: FCM
Primer paso - Colaboración

- Paso 1: hacer funcionar una luz
- Crear una clase que gestione la máquina de estados de una única luz (señal)
  - Mediante un evento, esta luz puede ir a cualquiera de sus estados
  - Agregar una clase ("buddy class") que permita incorporar las *pruebas unitarias*
  - Demostrar su funcionamiento mediante ejecución

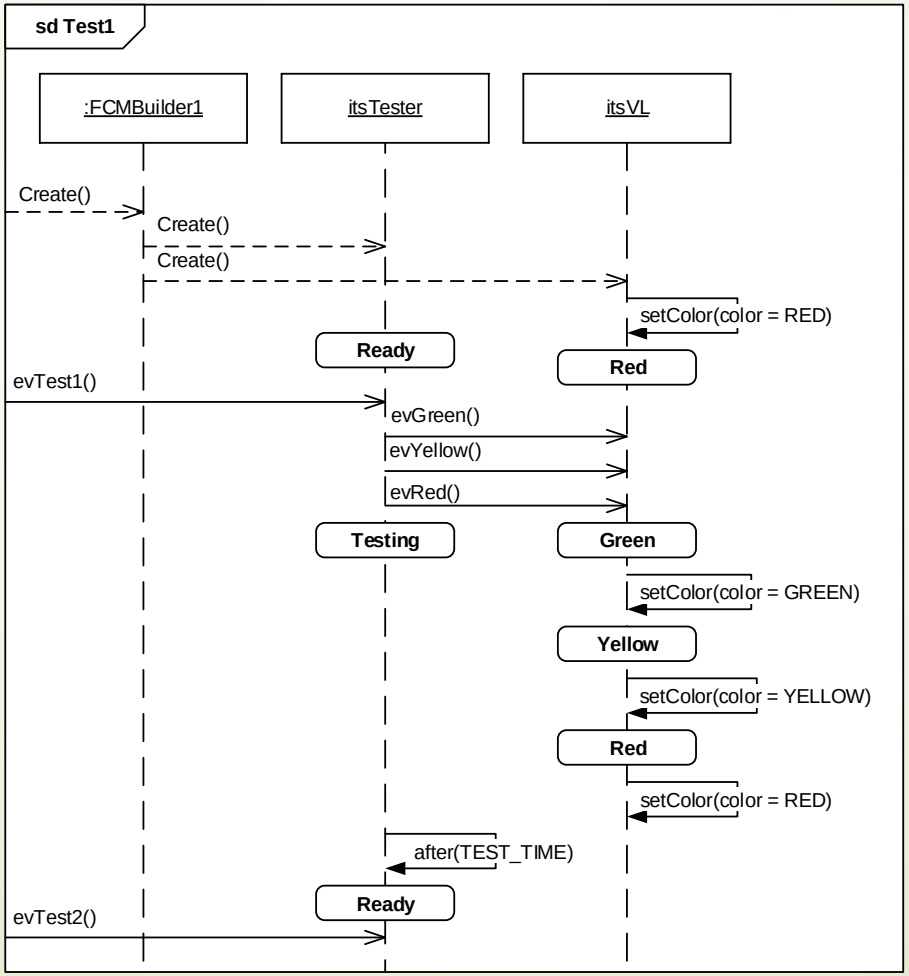


Ejemplo: FCM
Primer paso - Comportamiento



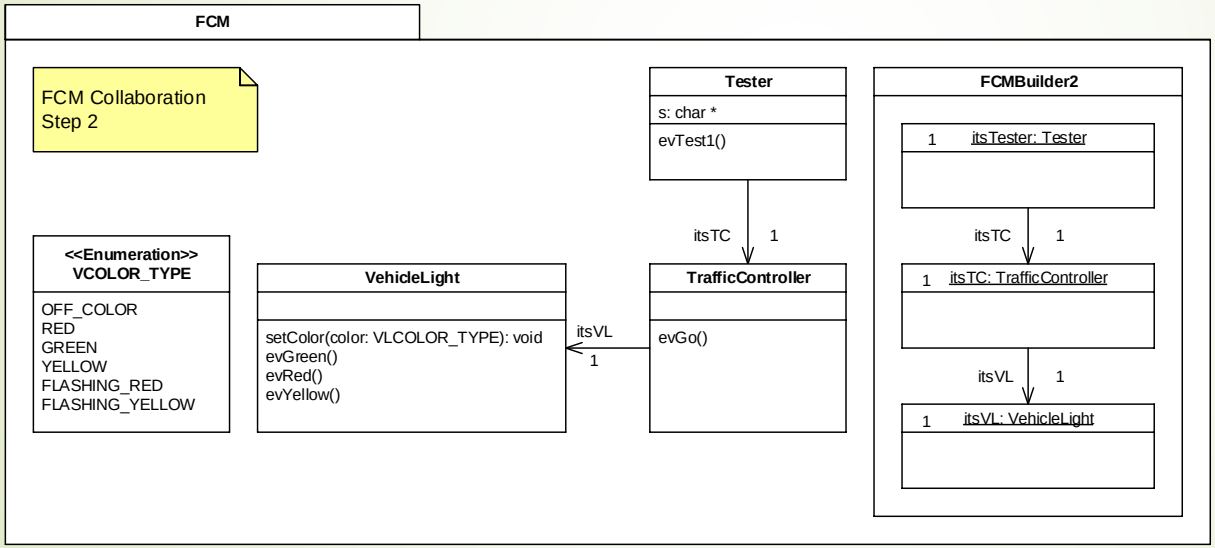


Ejemplo: FCM
Primer paso – Ejecución



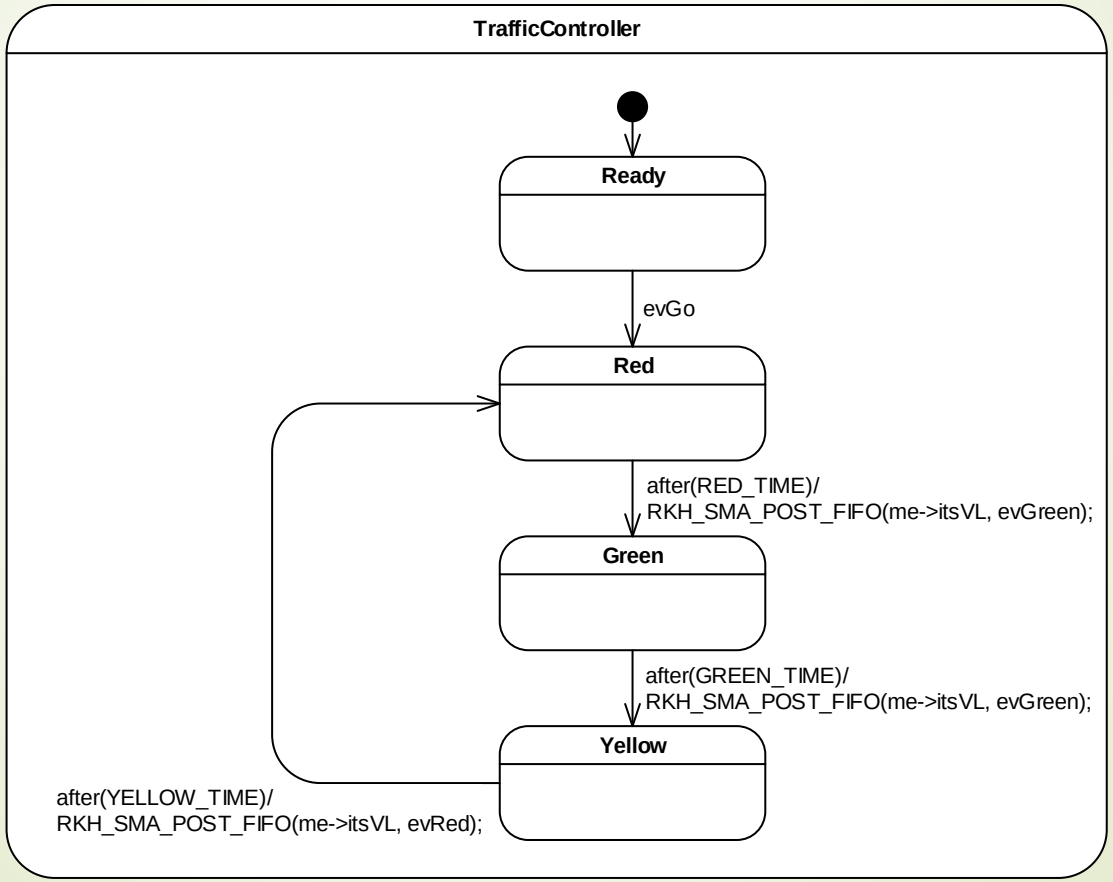
Ejemplo: FCM
Segundo paso - Colaboración

- Paso 2: agregar un controlador y verificar que una única luz recorra sus estados posibles



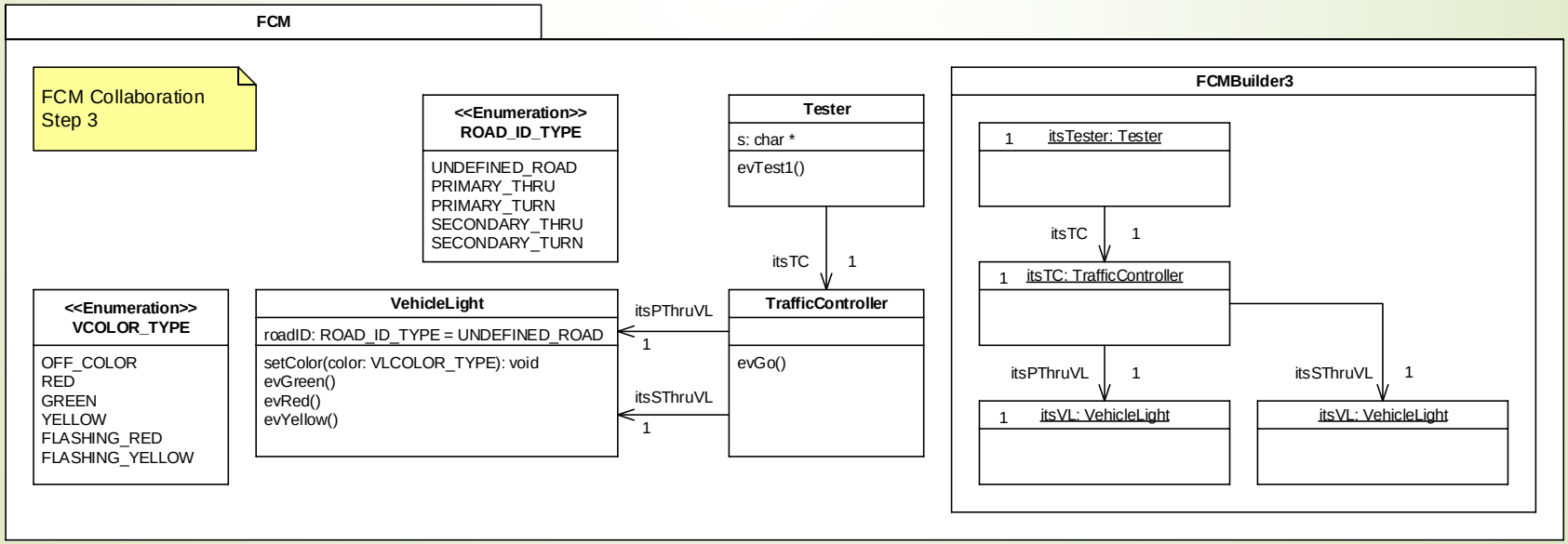
- Agregar una clase que permita controlar este modo
- Lograr que una única luz recorra sus ciclos, con la temporización correcta.

Ejemplo: FCM
Segundo paso - Comportamiento

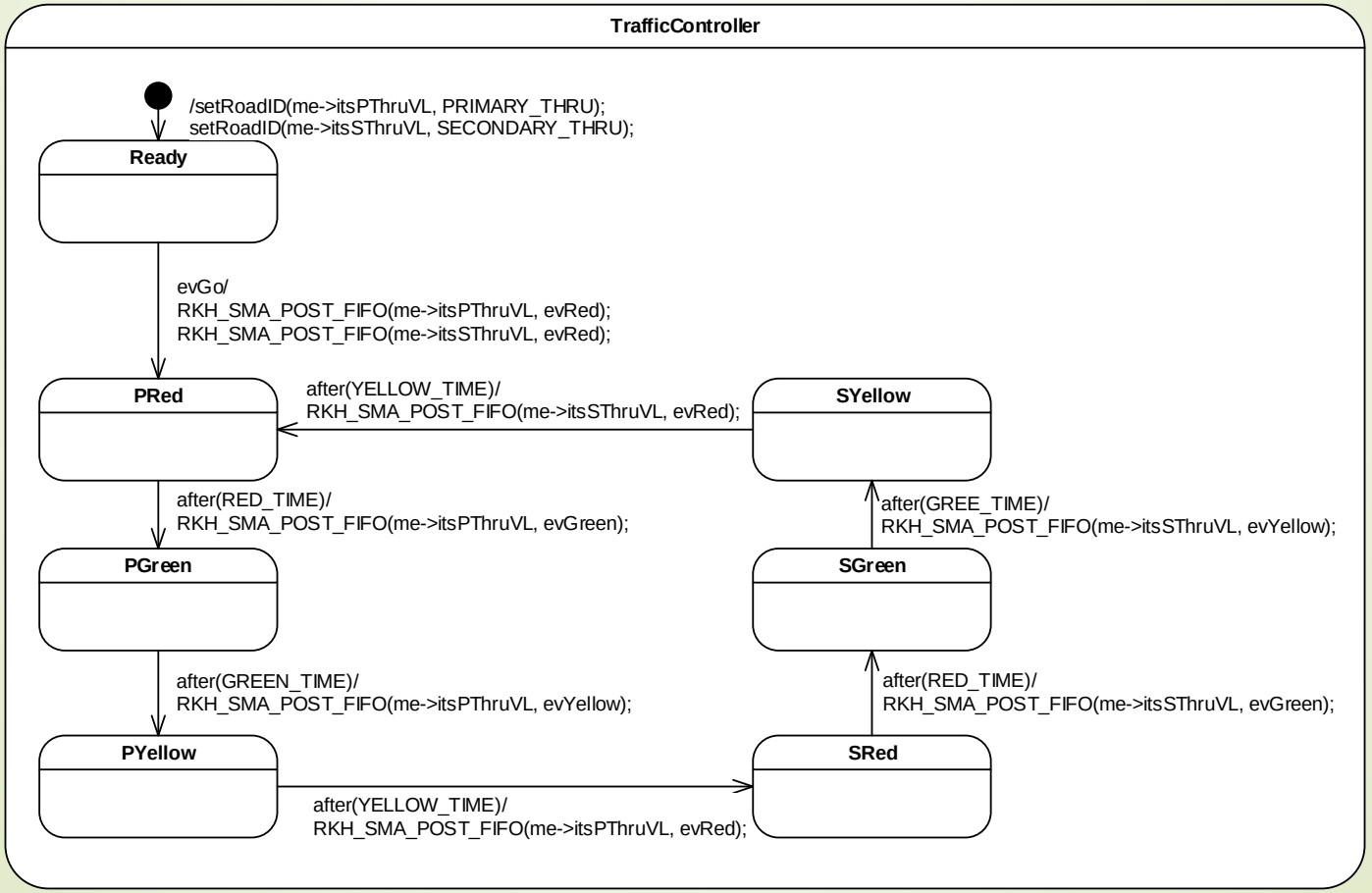


Ejemplo: FCM
Tercer paso - Colaboración

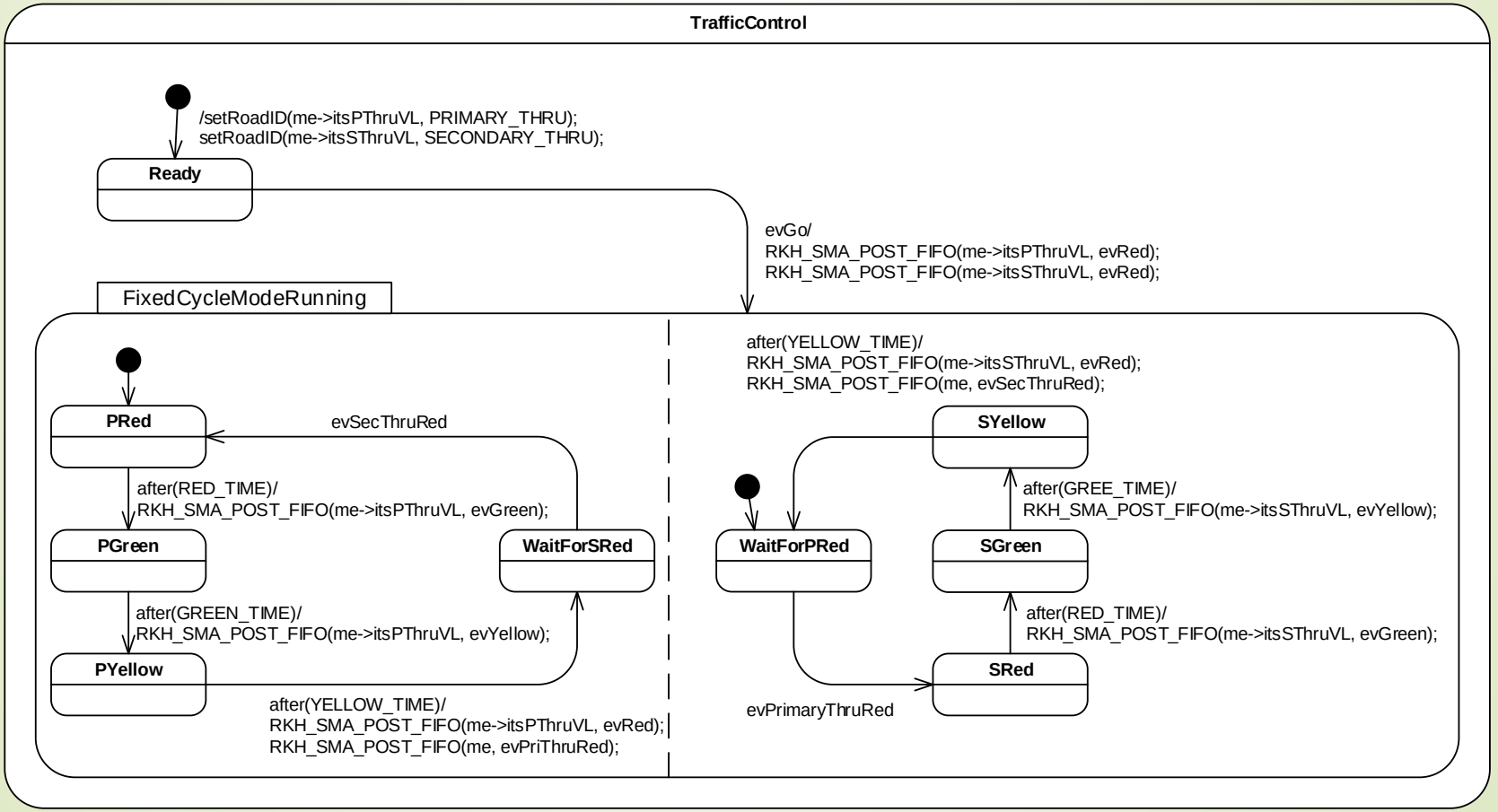
- Paso 3: agregar luces de paso (verde) para la calle principal y la secundaria. Lograr su control
- Agregar múltiples instancias de luces y la lógica de estados necesaria al controlador



Ejemplo: FCM
Tercer paso – Comportamiento I

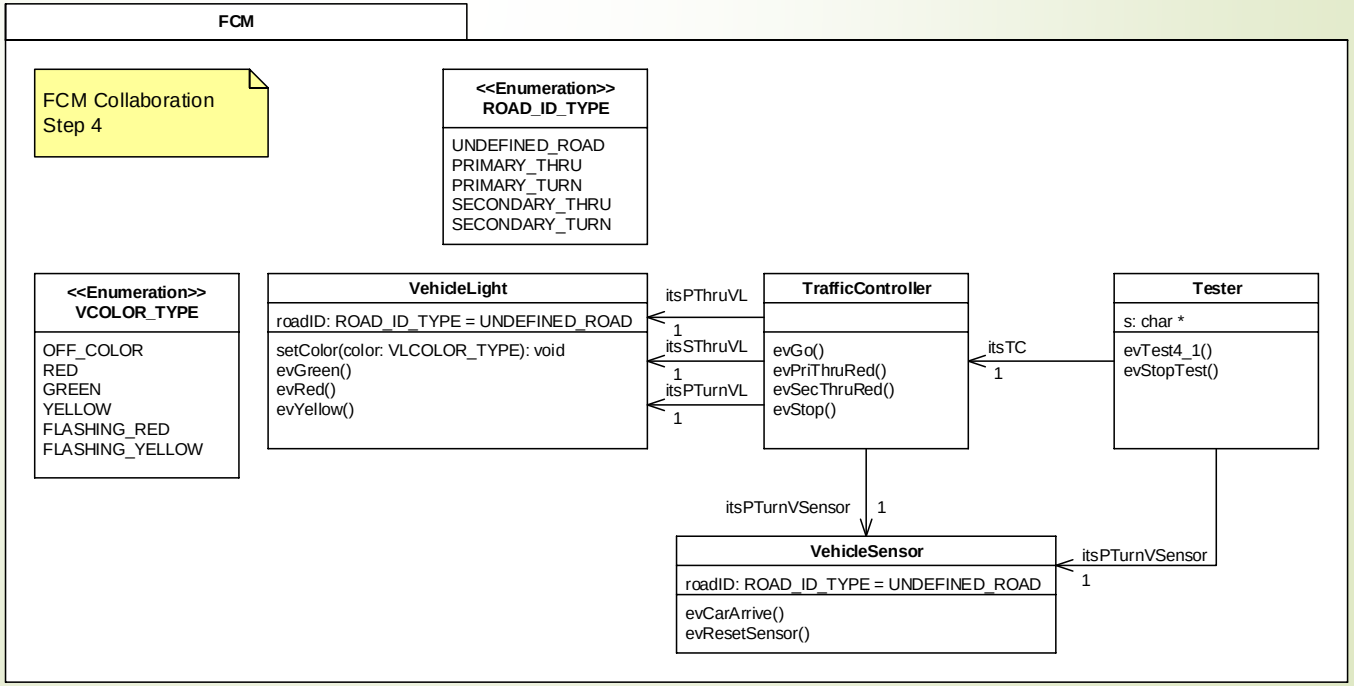


Ejemplo: FCM
Tercer paso – Comportamiento II

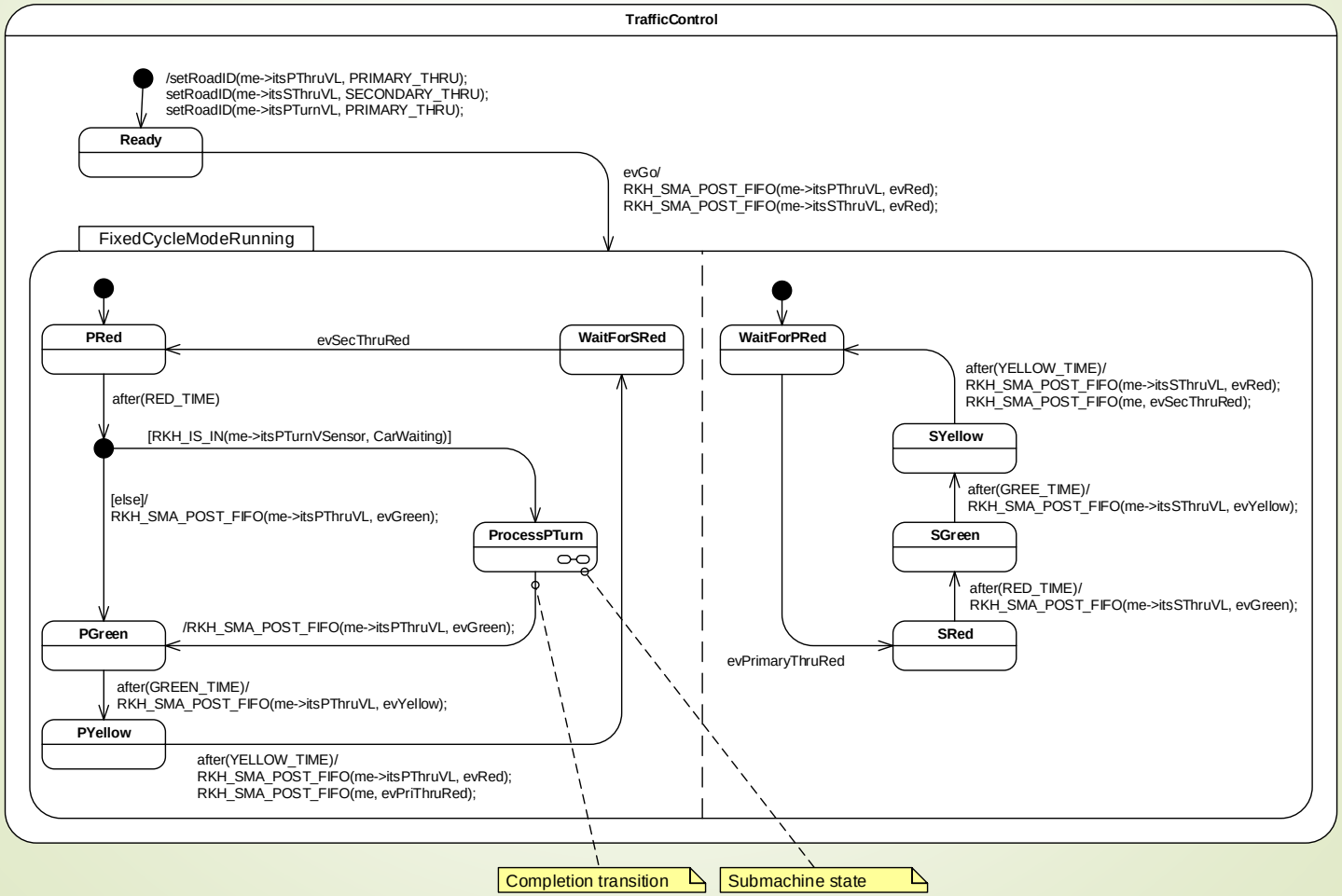


Ejemplo: FCM
Cuarto paso - Colaboración

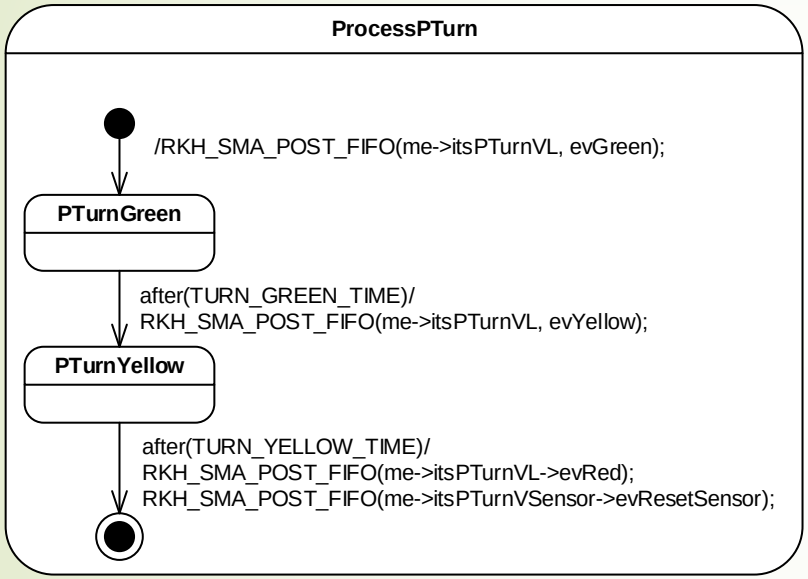
- Paso 4: agregar sensores de automóviles para los carriles de giro
- Agregar una clase para el sensor de automóviles que pueda enviar un evento cuando detecta un automóvil (en el carril de giro). Agregar la lógica en el controlador para recibir y mantener dicha información, para su posterior procesamiento.

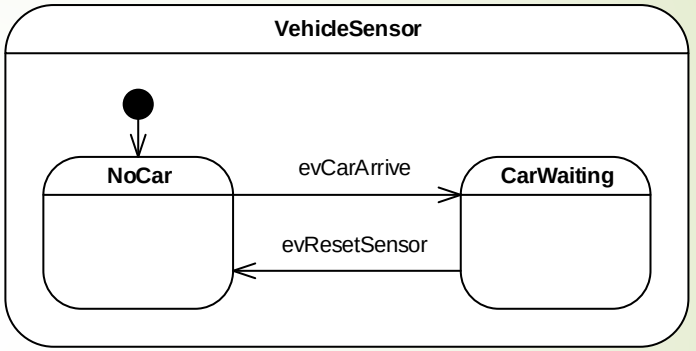


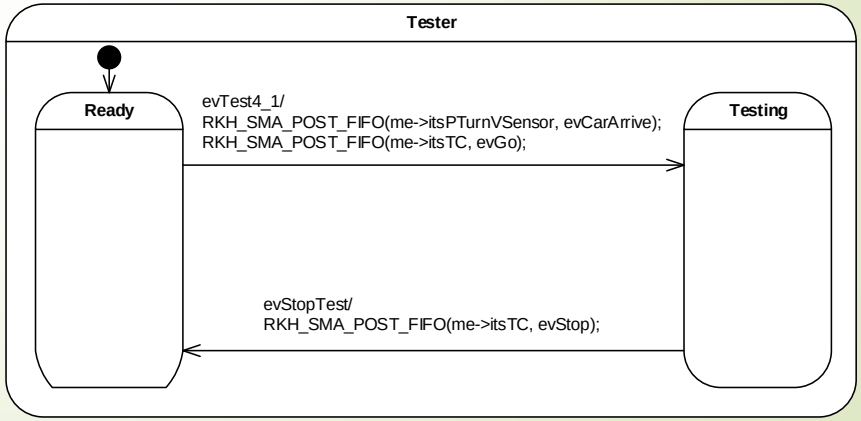
Ejemplo: FCM
Cuarto paso – Comportamiento I



Ejemplo: FCM
Cuarto paso – Comportamiento II

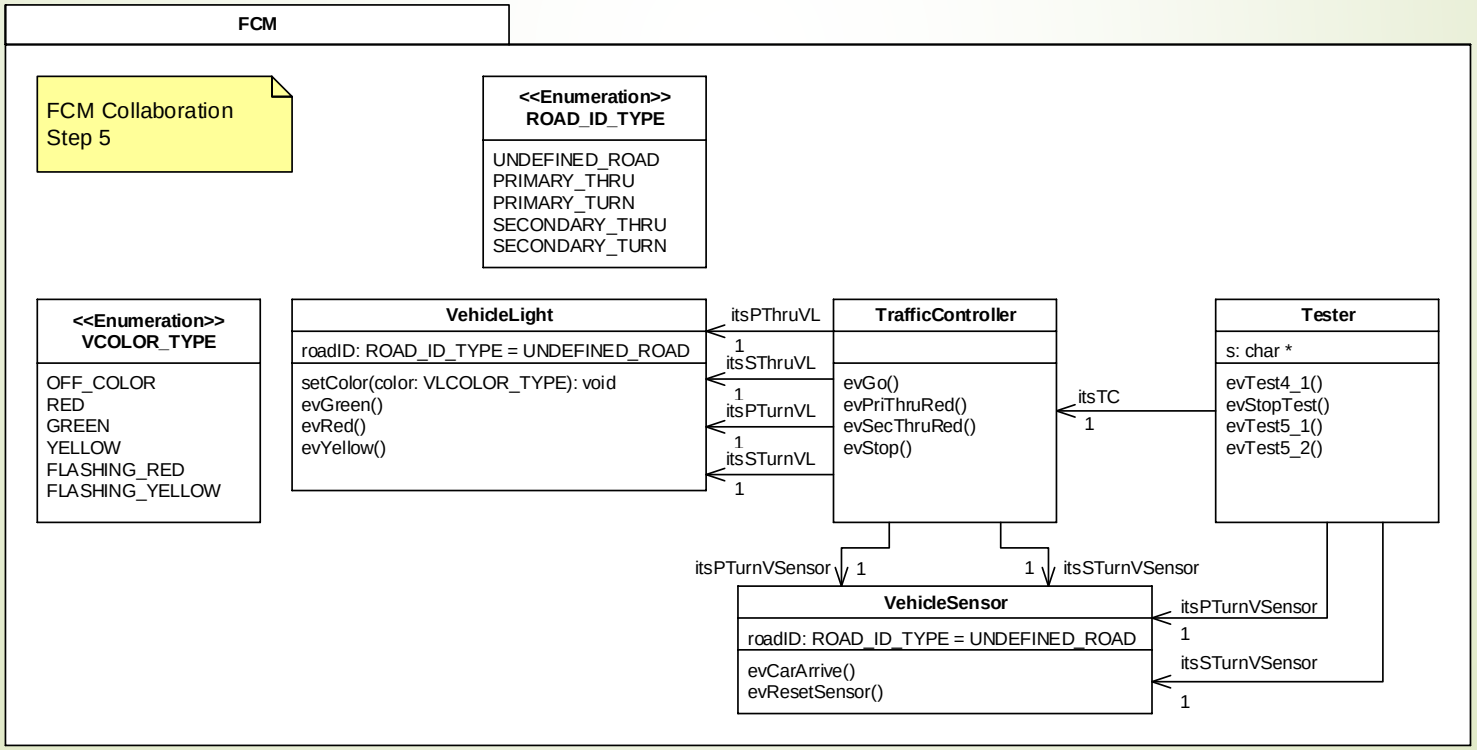




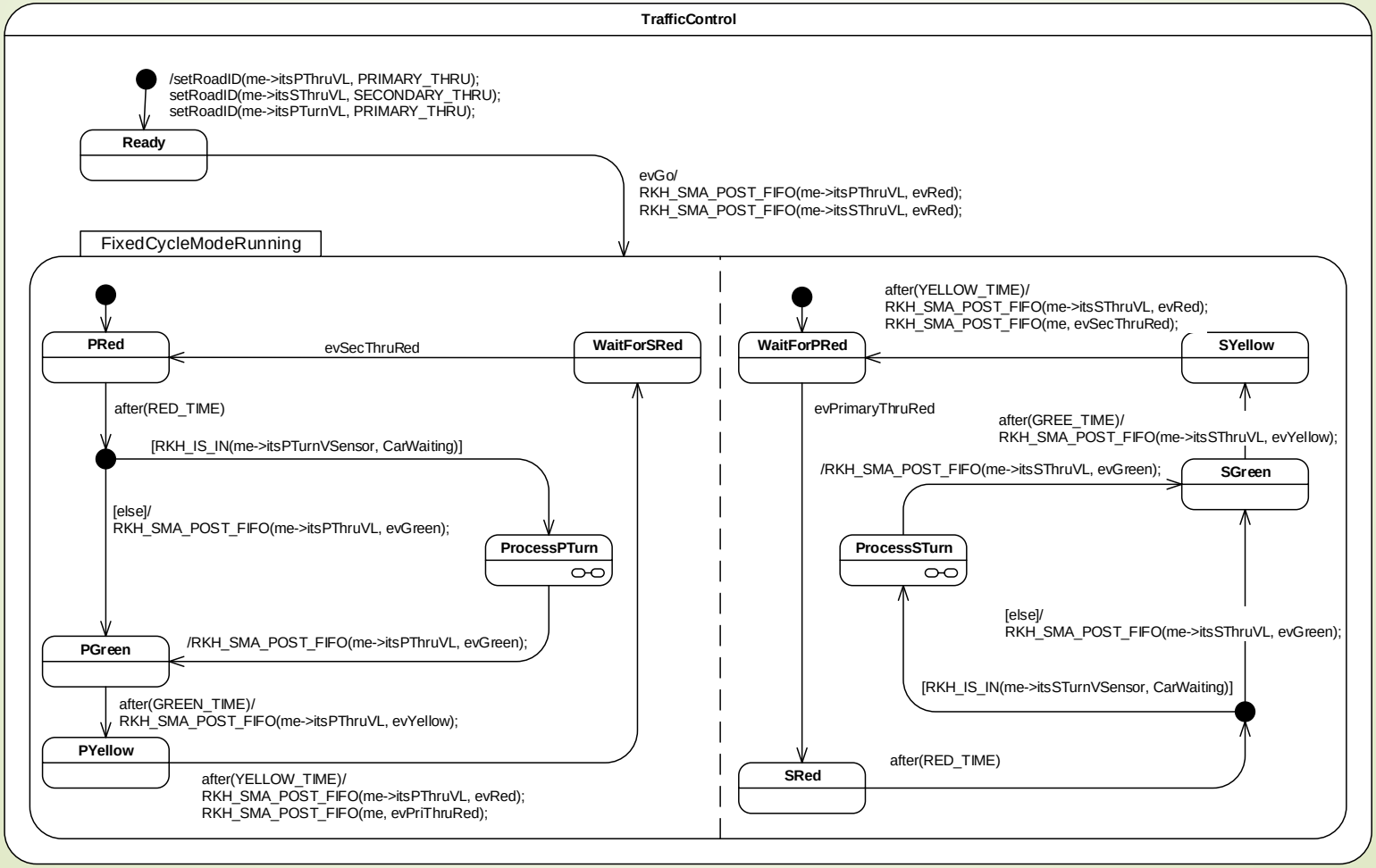


Ejemplo: FCM
Quinto paso - Colaboración

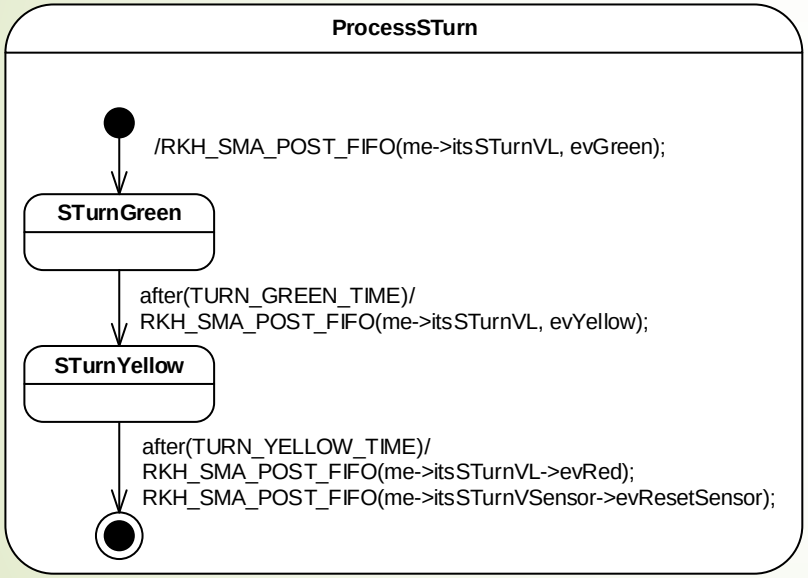
- Paso 5: agregar el segundo carril de giro
- Agregar instancias para los carriles de giro y procesar un caso con un automóvil detectado en el carril de giro

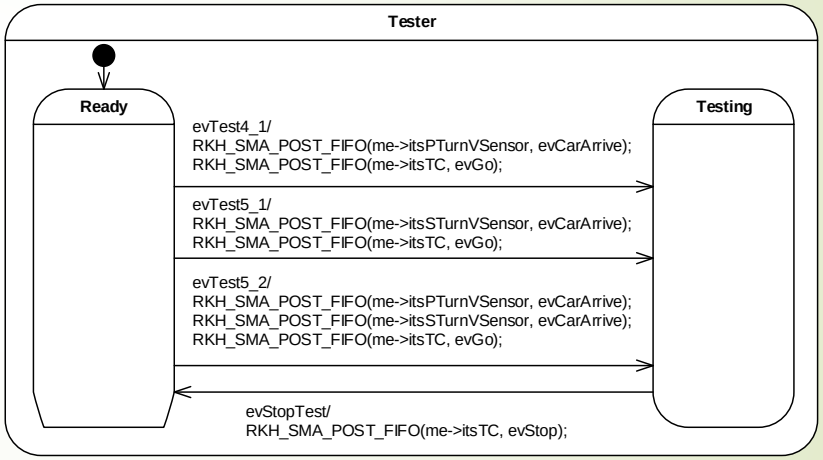


Ejemplo: FCM
Quinto paso – Comportamiento I



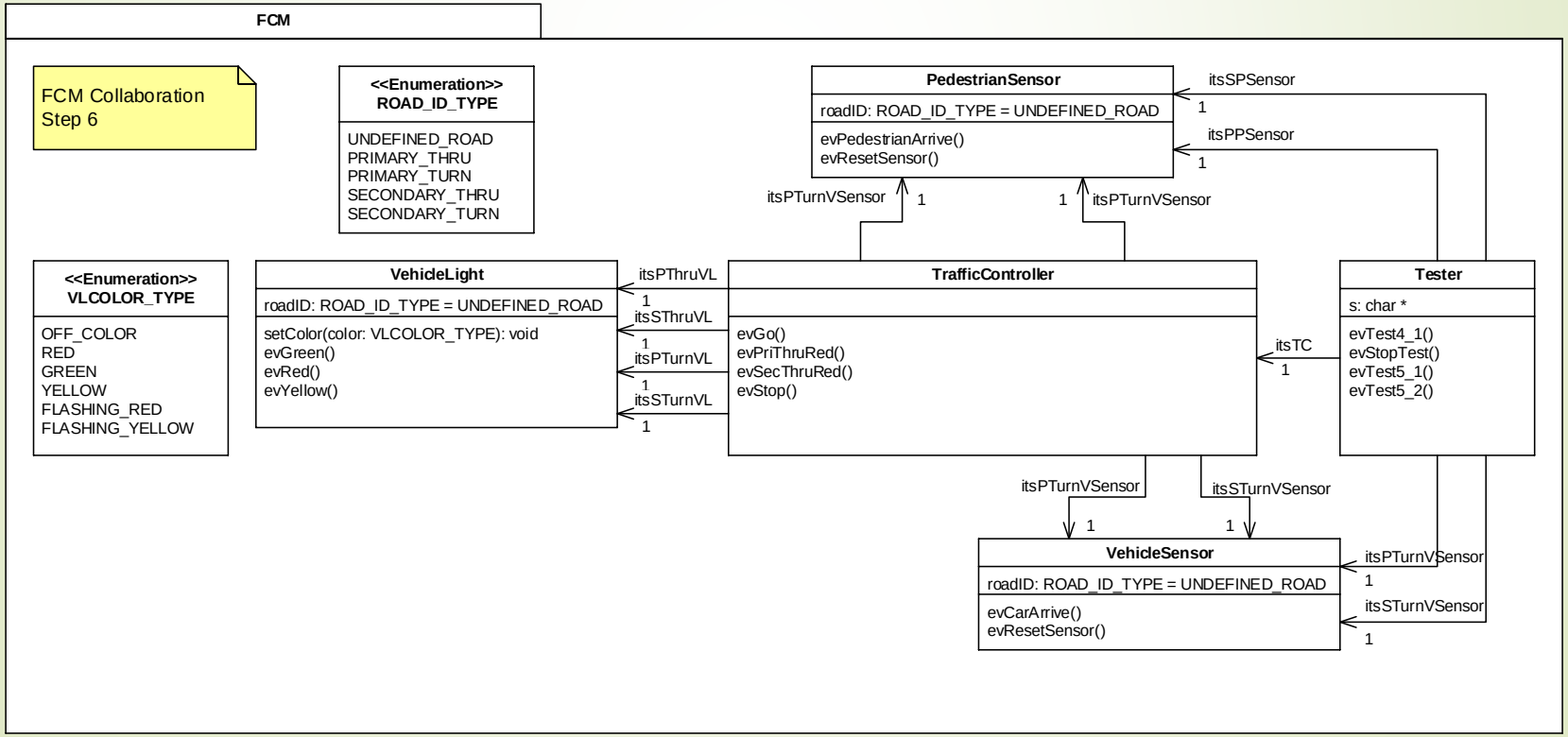
Ejemplo: FCM
Quinto paso – Comportamiento II





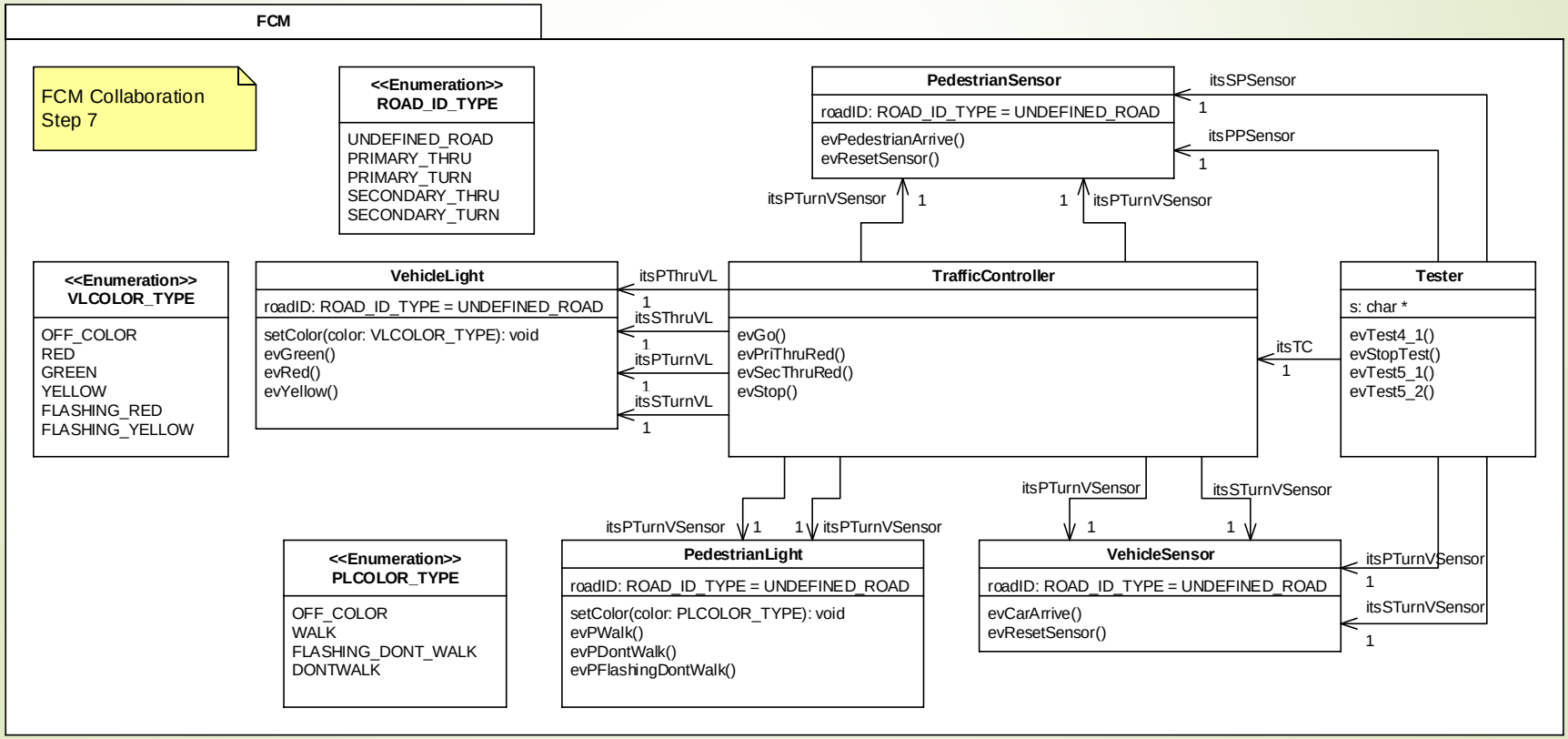
Ejemplo: FCM
Sexto paso - Colaboración

- Paso 6: agregar los botones de peatones e identificar dónde se procesarán sus solicitudes



Ejemplo: FCM
Séptimo paso - Colaboración

- Paso 7: manejar las solicitudes de los peatones



Ejemplo: FCM
Codificación

- Sólo un simple ejemplo de la posible codificación en lenguaje C de la operación `set_color()`
- Sus detalles permiten vislumbrar la manera en que pueden representarse los conceptos anteriores en lenguaje C.

```
void
PedestrianLight_setColor(PedestrianLight *const me, PLCOLOR_TYPE color)
{
   switch(me->roadID)
   {
      case PRIMARY_THRU:
         print("Primary Ped. Light is ");
         break;
      case SECONDARY_THRU:
         print("Secondary Ped. Light is ");
         break;
      default:
         print("Unknown road is ");
         break;
   }

   switch(color)
   {
      case WALK:
         print("Walk\n");
         break;
      case FLASHING_DONT_WALK:
         print("Flashing Don't Walk\n");
         break;
      case DONT_WALK:
         print("Don't Walk\n");
         break;
      default:
         print("Hug??\n");
         break;
   }
}
```

Preguntas

Referencias

[1] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Sci. Comput. Programming 8 (1987), pp. 231–274.

[2] RKH, "RKH Sourceforge download site,", http://sourceforge.net/projects/rkh-reactivesys/ , August 7, 2010.

[3] Object Management Group, "Unified Modeling Language: Superstructure version 2.1.1," formal/2007-02-05, Febrary 2007.

[4] B. P. Douglass, "Design Patterns for Embedded Systems in C,", Elseiver, October 7, 2010

[5] B. P. Douglass, "Real-Time UML: Advances in the UML for Real-Time Systems (3rd Edition),", Elseiver, October 4, 2006

[6] B. P. Douglass, "Real-Time Agility: The Harmony/ESW Method for Real-Time and Embedded Systems Development,", Elseiver, June 19, 2009

[7] M. Samek, "Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems," Elsevier, October 1, 2008.

[8] Kernighan & Ritchie, "C Programming Language (2nd Edition)", April 1, 1988.