Sistemas Empotrados Distribuidos

Automatizando el diseño: modelando un Sistema Empotrado Distribuido

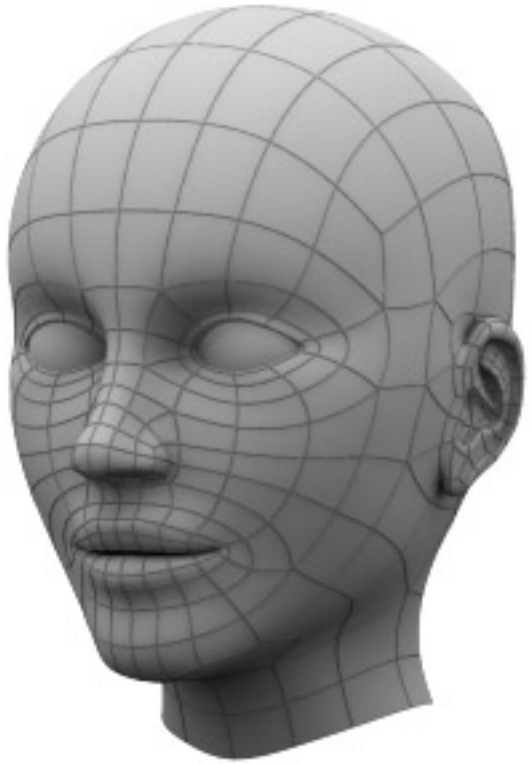
Prof. Guillermo Botella

Departamento de Arquitectura de Computadores y Automática

Curso 2022-2023

Contenidos

- Introducción: ¿por qué modelar?
- Requerimientos de un modelo
- Modelos de Computación
- Alternativas de modelado
  - CFSM
  - Petri Nets
- UML
  - Elementos
  - Relaciones
  - Diagramas
  - Automatizando el Diseño
- Bibliografía
  - P. Marwedel, "Embedded System Design" [Ch. 2]
  - G. Booch et al., "El Lenguaje Unificado de Modelado"



Introducción: ¿por qué modelar?

- ¿Por qué no escribir el código directamente?
  - Puede funcionar para 100 LC
  - La mayoría de los SEDs son bastante más grandes
  - Los SEDs tienen que ser verificados (antes del deadline)
  - Los problemas crecen exponencialmente con respecto a la complejidad del sistema
- No lo entiendo, ¿qué puede ir mal ?
  - En 1999 GM tuvo que retirar 3.5 millones de vehículos por un defecto en el software de control del ABS
    - Las distancias de frenado se aumentaron 15-20m
    - 2111 accidentes y 293 heridos
    - http://autopedia.com/html/Recall_GM072199.html
  - Desde entonces GM (y otros) han invertido en la calidad del SW

Introducción: ¿por qué modelar?

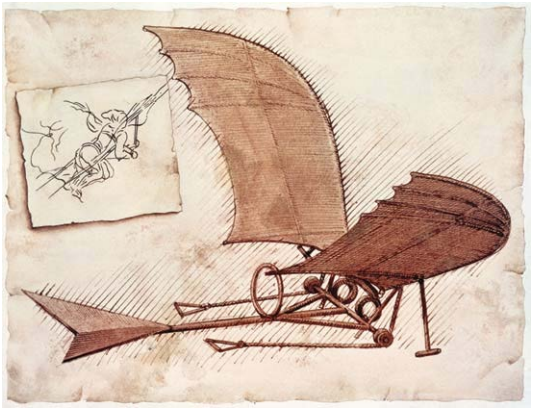
- ¿Qué es un modelo?
  - Una simplificación de la realidad
- Un modelo conlleva una pérdida de información respecto a la realidad … entonces ¿por qué modelar?
- Modelamos para comprender mejor el sistema que estamos desarrollando
- Objetivos que se consiguen al modelar:
  - Visualizar cómo queremos que sea el sistema
  - Especificar la estructura o el comportamiento de un sistema
  - Proporcionar plantillas para guiar la construcción del sistema
  - Documentar las decisiones de diseño

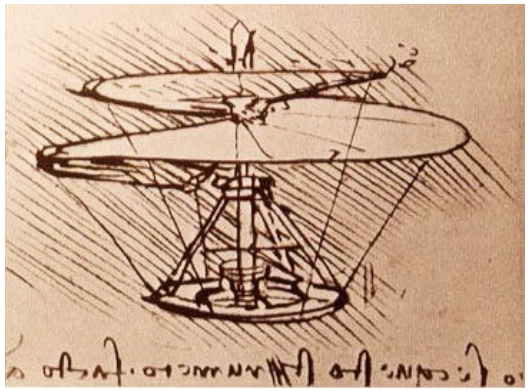
Introducción: ¿por qué modelar?

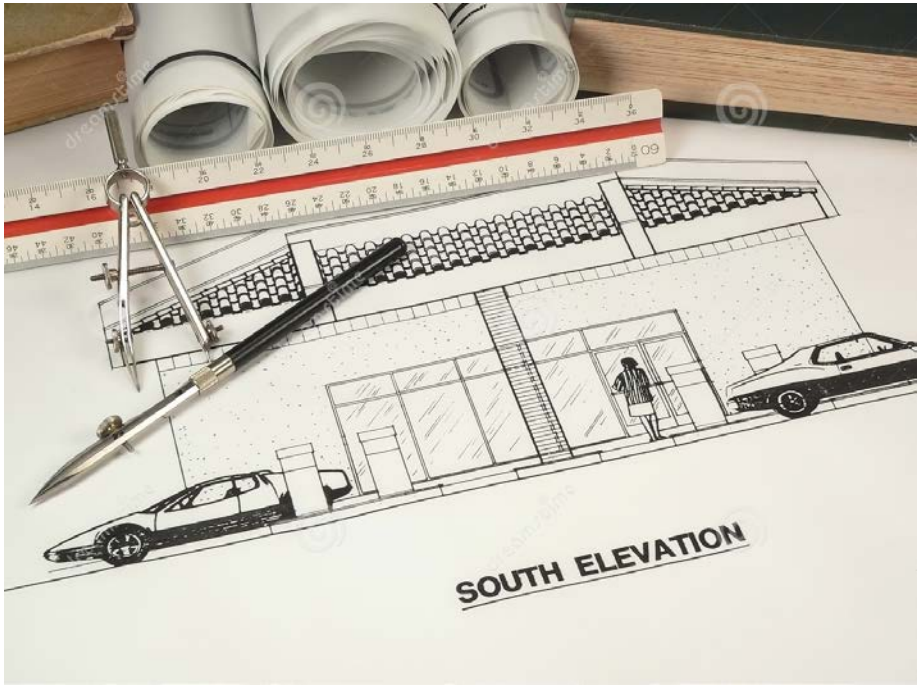
- Pero esto del modelado me suena a Ingeniería del Software, ¿para qué modelar un SED ?
  - Recordatorio: SED = HW + SW
- Una empresa con éxito es aquélla que produce de una manera consistente un producto de calidad que satisface las necesidades de los usuarios
  - Repetibilidad y reusabilidad ➔ Diseño Automático
- Modelar bien un sistema permite detectar fallos en una etapa temprana del **diseño**
  - Ahorra tiempo y dinero
  - Permite garantizar la correctitud (correctness) de un sistema con grano grueso ➔ Verificación
    - Ej. Evaluación de deadlocks y livelocks

Introducción: ¿por qué modelar?

- Modelo – Fase de diseño
  - El diseño no es implementación
  - Un buen diseño no contiene ninguna línea de código

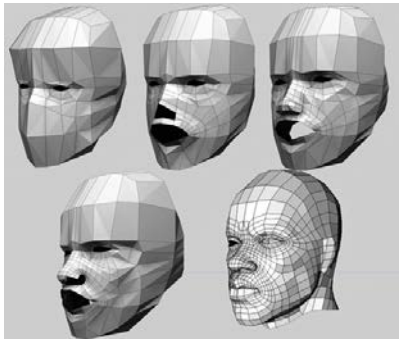






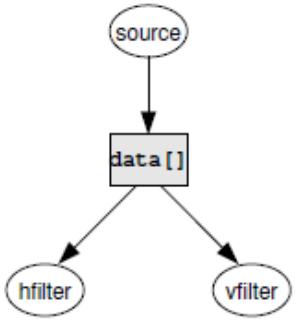
Requerimientos de un modelo

- Un buen modelo capta las propiedades fundamentales de un sistema y omite las menos relevantes
- Pero un único modelo puede no ser suficiente
  - Granularidad
- Propiedades deseables en un modelo
  - Jerarquía: estructural y conductual
  - Concurrencia. Un SED tiene múltiples elementos actuando en paralelo
  - Sincronización y comunicación. Ej. Exclusión mutua
  - Comportamiento en el tiempo. Muchos SED son sistemas RT
  - Comportamiento orientado al estado. Fácil de describir
  - Manejo de eventos
  - Flexibilidad y portabilidad



Modelos de Computación

- Un Modelo de Computación describe cómo se asume que se va a computar (obvio)
  - Determina cómo funcionará el sistema
- Elementos:
  - Componentes: procedimientos, funciones, procesos, FSMs, etc.
  - Protocolos de Comunicación. Mecanismo de interacción entre los componentes

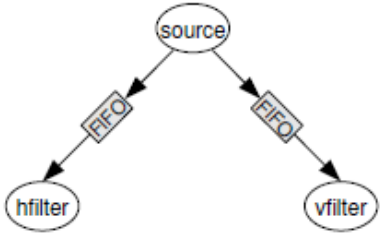


Shared memory model

```
int data[N];

for (i = 0; i < N; i++) {
  data[i] = source();
}

for (i = 0; i < N; i++) {
  hfilter(data[i]);
  vfilter(data[i]);
}
```

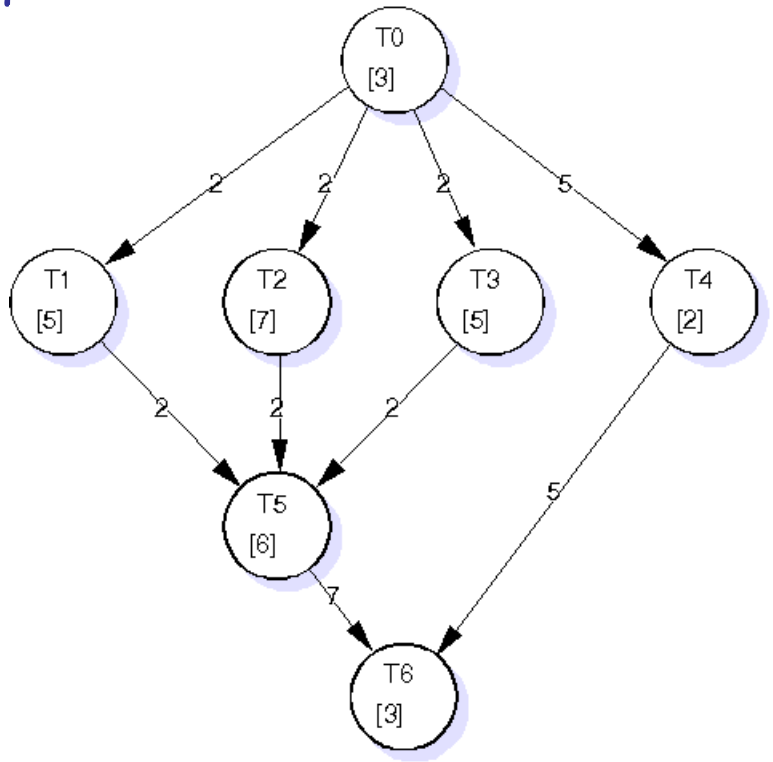


Distributed memory model

Van Haastregt et al., DATE '09

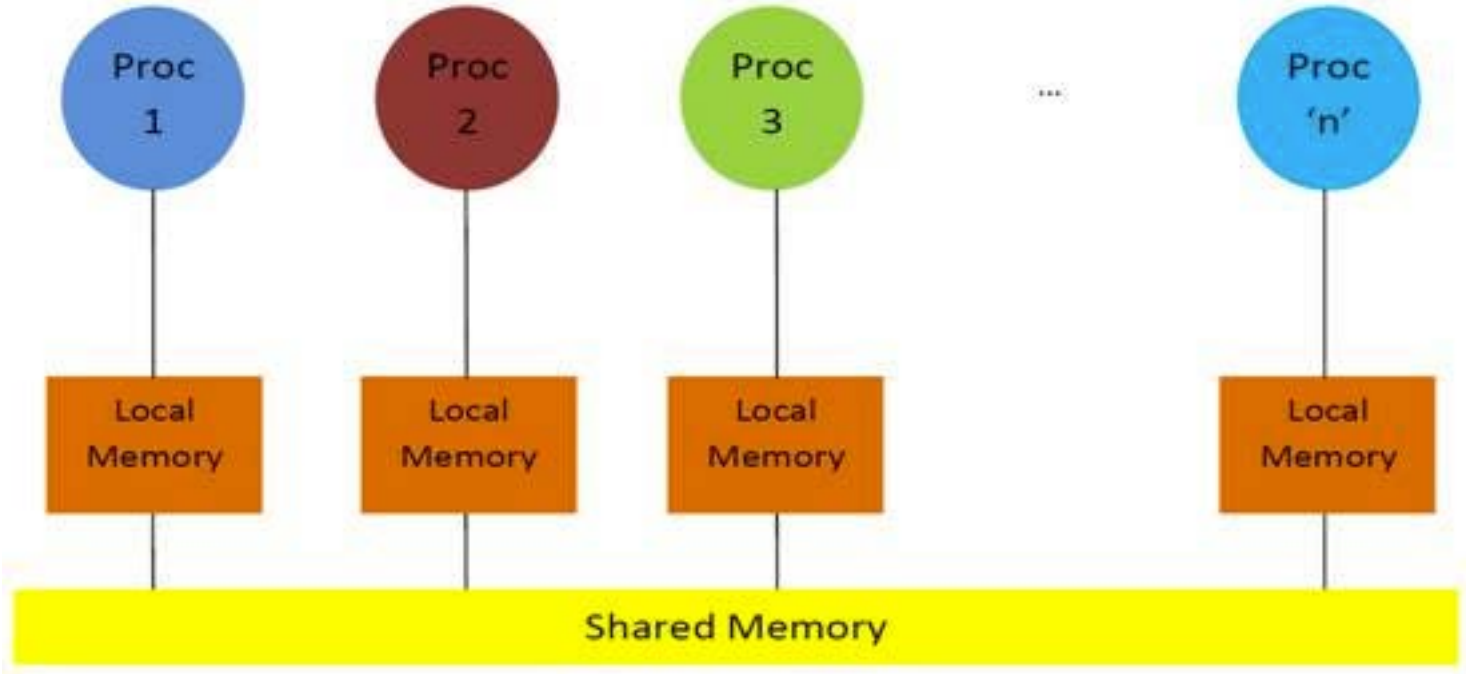
Modelos de Computación: Componentes

- La relación entre los componentes suele modelarse con **grafos de dependencias**
  - Los nodos o vértices son los componentes
  - Las aristas o ejes expresan la relación de precedencia entre dos nodos. A → B significa que A debe ejecutarse antes que B
- Extensiones
  - Etiquetas de tiempo
  - Relaciones de E/S
  - Grafos jerárquicos



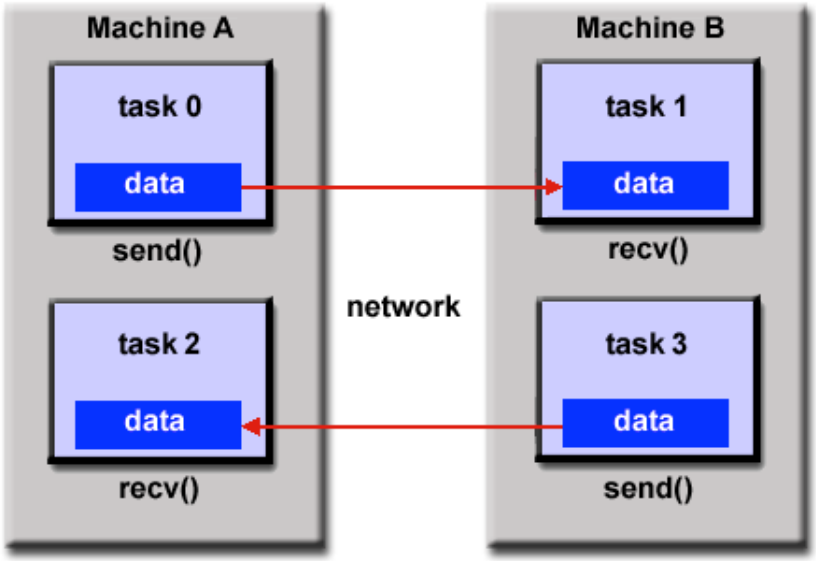
Modelos de Computación: Protocolos de Comunicación

- Memoria Compartida
  - Comunicación por medio de la memoria
    - Secciones críticas
  - Problemas de coherencia en sistemas multiprocesador si cada procesador tiene niveles privados de memoria



Modelos de Computación: Protocolos de Comunicación

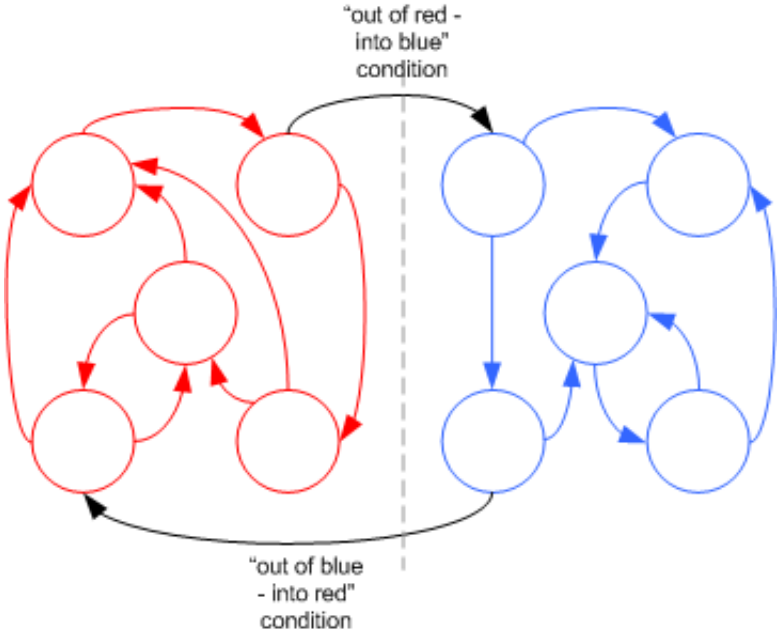
- Paso de Mensajes (Modelo Distribuido)
  - Asíncrono o no bloqueante
    - Los mensajes se mandan y se acumulan en un buffer
    - El emisor no espera a que el receptor esté listo
  - Síncrono o bloqueante (*rendez-vous*)
    - Emisor y receptor deben estar listos para la comunicación
  - Extended rendez-vous
    - El emisor puede continuar sus tareas si el receptor le envía un *ack*



Alternativas de modelado: CFSMs

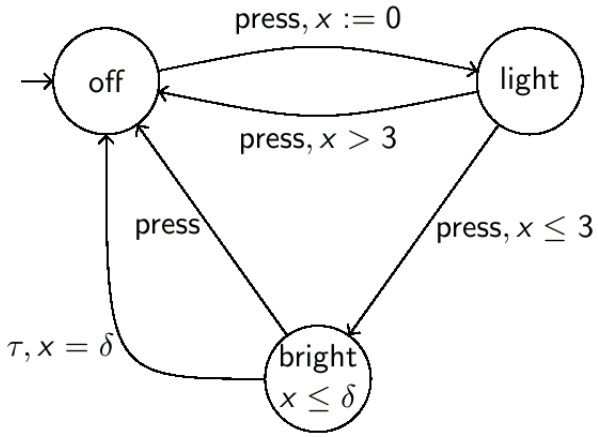
- Una máquina de estados (Finite State Machine) es un modelo matemático para circuitos secuenciales
  - La FSM está definida por los estados y por las transiciones
  - La FSM solo puede estar en un estado en cada instante
  - Cuando los eventos que disparan una transición **desde el estado actual** se cumplen, se transita al estado siguiente
  - Nos centraremos en las FSMs deterministas y síncronas
- Un sistema suele estar formado por más de una FSM
  - Necesidad de comunicación: Communicating FSMs (CFSMs)

Alternativas de modelado: CFSMs



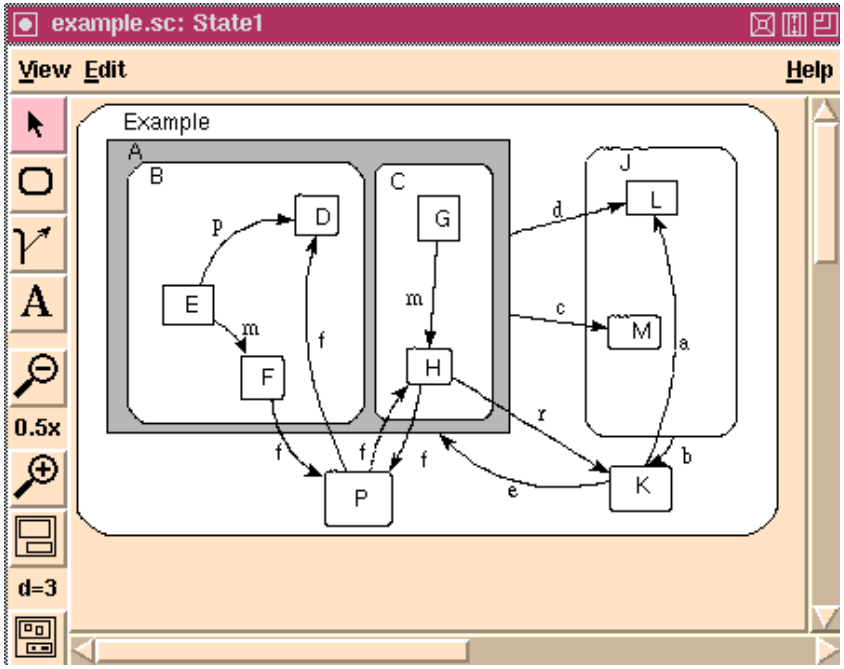
Capturan bien la jerarquía y el comportamiento orientado a estados

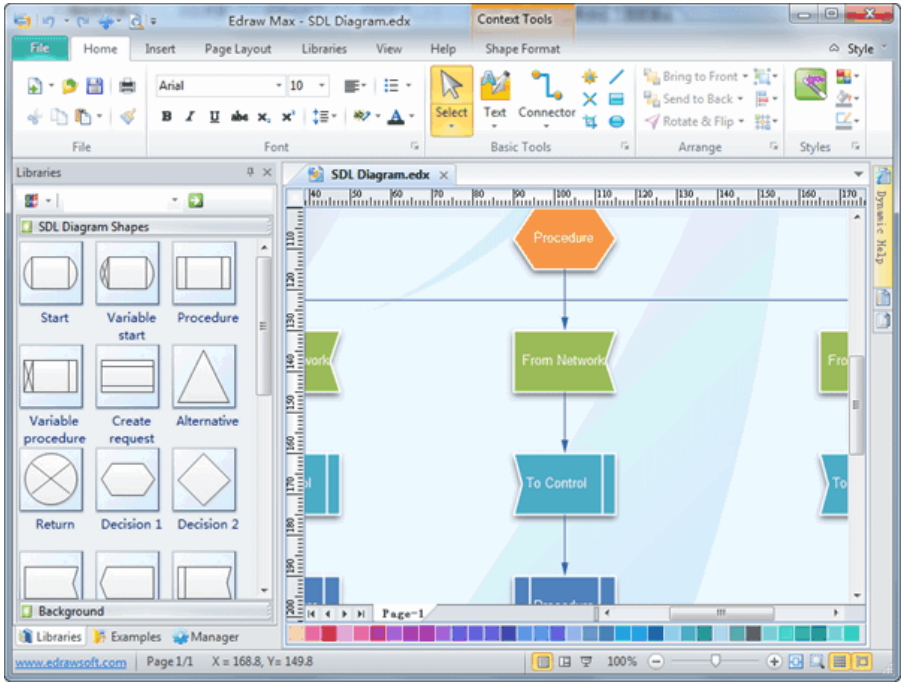
Problemas capturando el timing (sol. *Timed automata's*) y la concurrencia



Alternativas de modelado: CFSMs

- Lenguajes que lo implementan
  - StateCharts: basado en memoria compartida
  - SDL: basado en paso de mensajes, mejor para sistemas distribuidos

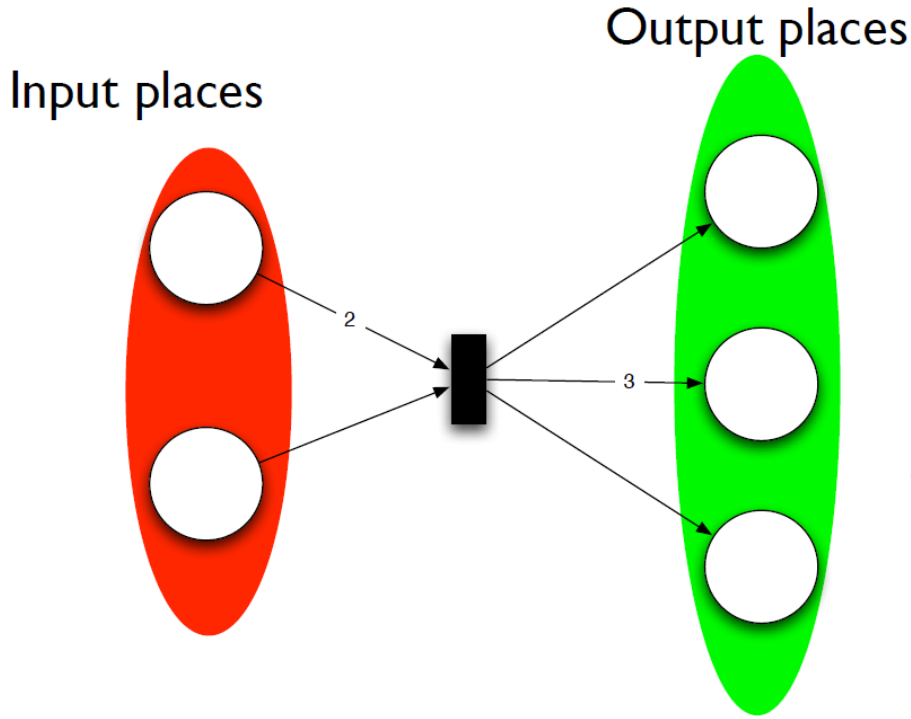


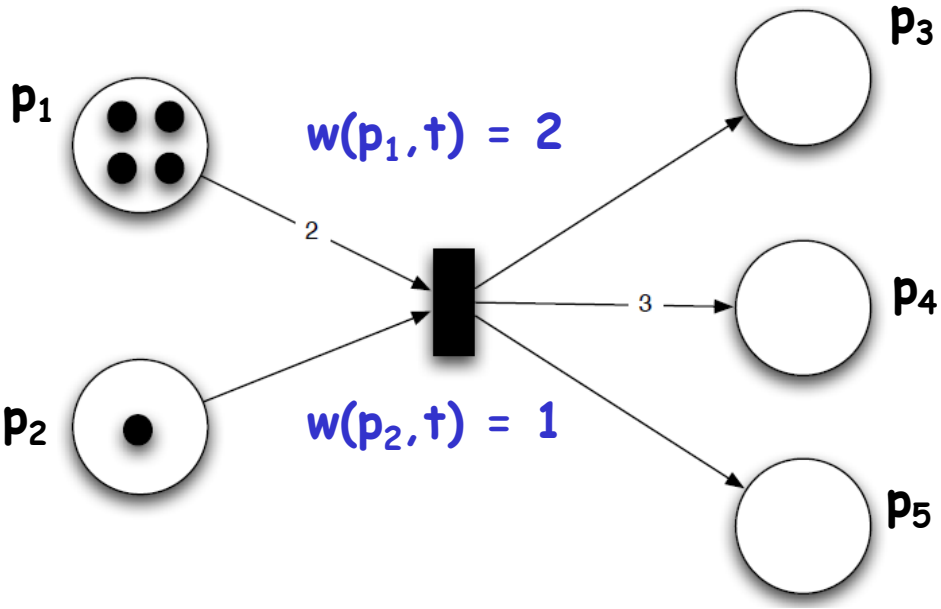


Alternativas de modelado: Petri Nets

- Las Petri Nets (PNs) modelan dependencias de control (dependencias causales, Adam Petri, 1962)
- Basadas en el modelo distribuido
- Componentes <P,T>
  - P: conjunto de nodos o *places* (p): modelan condiciones
  - T: conjunto de transiciones (t): modelan eventos
  - Tokens: fluyen por la red
  - Función de peso *w(p,t)* :determina las transiciones
- Cada transición es una tupla *t=<I,O>* donde
  - *I*: es una función tal que *t* consume *I(p)* tokens en cada place *p*
  - *O*: es una función tal que *t* produce *O(p)* tokens en cada place *p*
- Una transición se dispara en un instante de tiempo sii todos los nodos entrantes p_i contienen al menos $w(p_i,t)$ tokens

Alternativas de modelado: Petri Nets

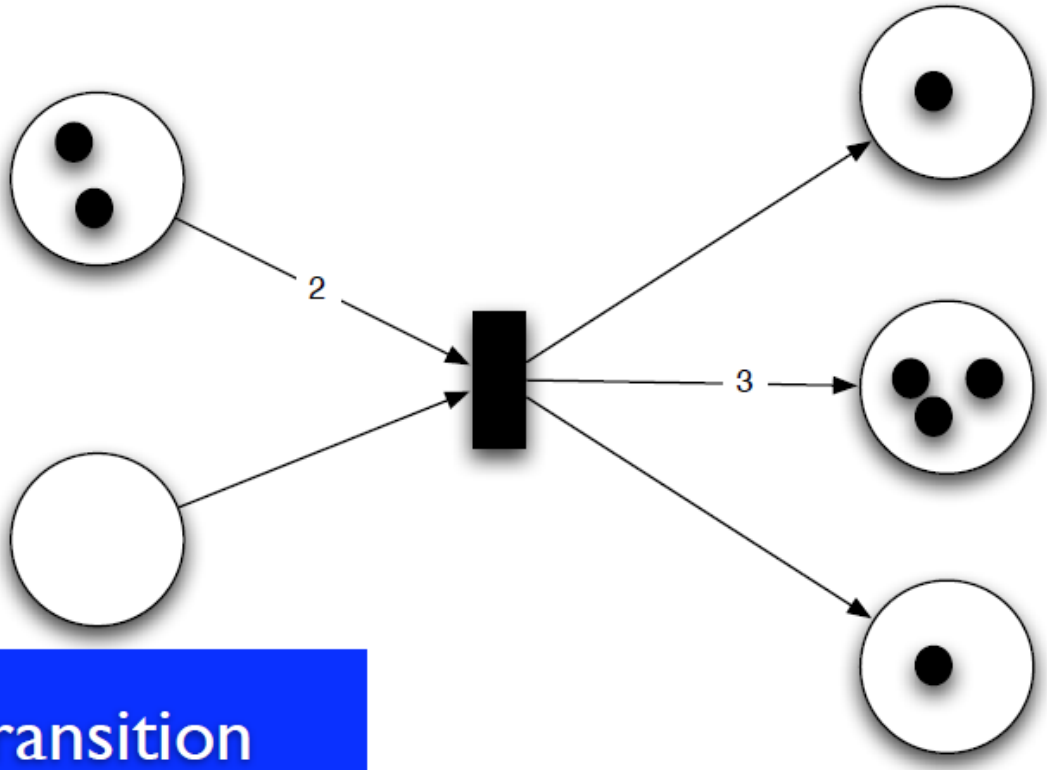




I(p1)=2 I(p2)=1 I(p3)=0 I(p4)=0 I(p5)=0

O(p1)=0 O(p2)=0 O(p3)=1 O(p4)=3 O(p5)=1

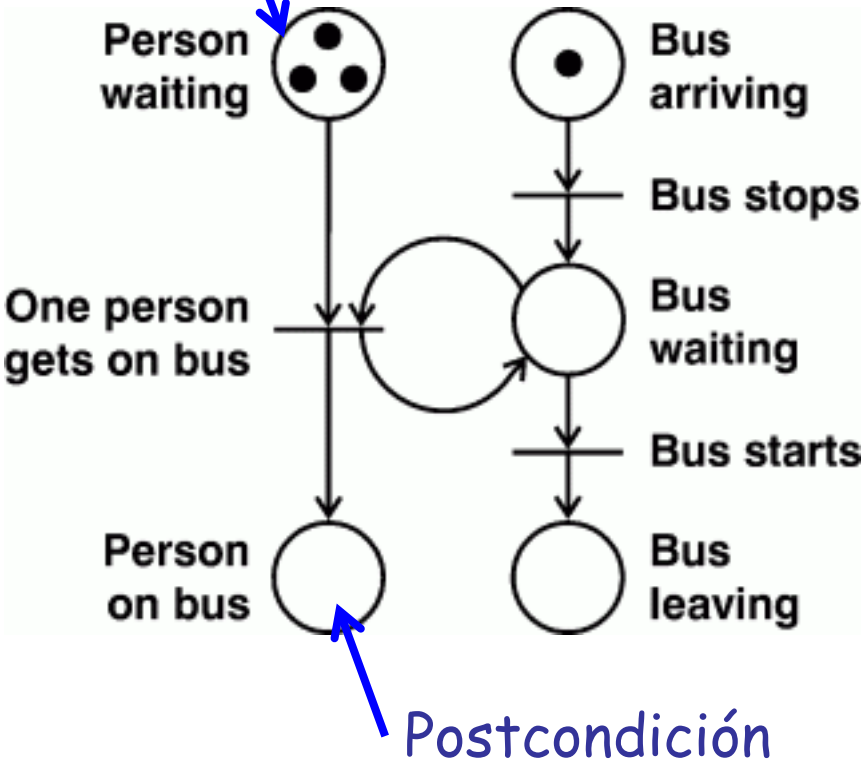
Transitions consume tokens from the input places and produce tokens in the output places

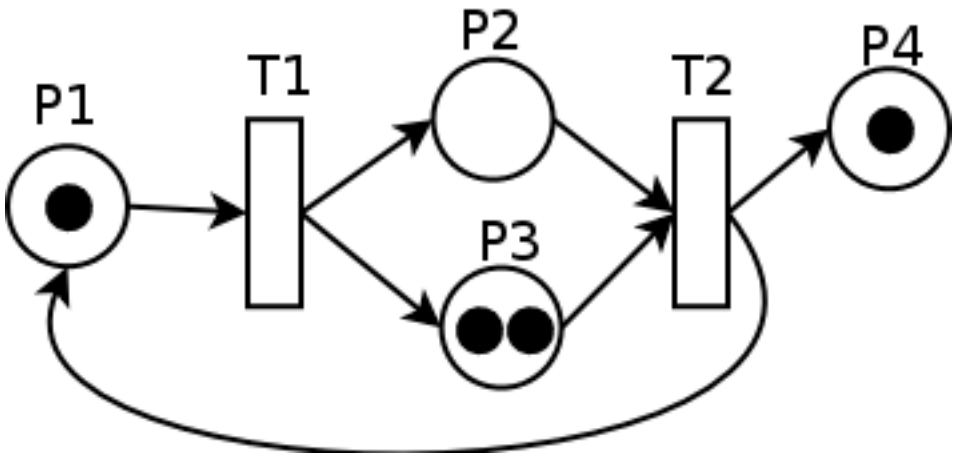


Now, the transition cannot be fired anymore

Alternativas de modelado: Petri Nets

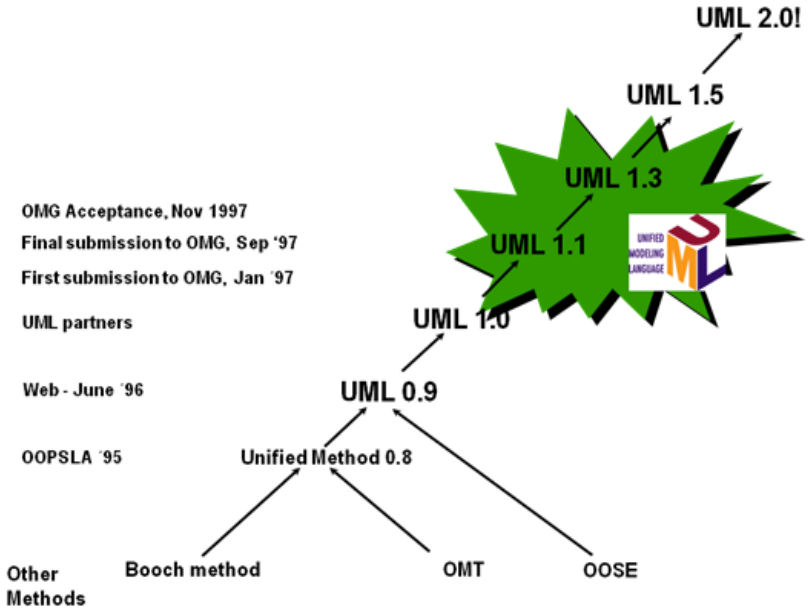
- Problemas
  - Modelado de datos
  - Modelado de tiempo
- Incluidas dentro de UML (diagramas de actividad)





UML

- Unified Modeling Language (UML)
- *Lingua Franca* en modelado
- Orientado a Objetos (OO)
- Estandarizado por el Object Management Group (OMG !!)
  - 1997: v1.0
  - 2005: v2.0
  - 2017: v2.5.1
- http://www.uml.org/



UML

- Varios diagramas describen distintas facetas de un mismo modelo
- Cada diagrama representa una *proyección* del modelo
- Los diagramas deben coexistir
  - Incoherencias entre diagramas ➔ Mal modelo

Structure diagrams

1. Class diagram
2. Composite structure diagram (*)
3. Component diagram
4. Deployment diagram
5. Object diagram
6. Package diagram
7. Profile diagram

Behavior diagrams

8. Use-case diagram
9. State machine diagram
10. Activity diagram

Interaction diagrams

11. Sequence diagram
12. Communication diagram
13. Interaction overview diagram (*)
14. Timing diagram (*)

(*) not existing in UML 1.x, added in UML 2.0

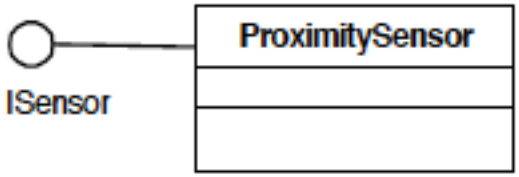
UML

- Elementos UML
  - Estructurales
  - De Comportamiento
  - De Agrupación
  - De Anotación
- Relaciones UML
  - Dependencias
  - Asociación
  - Generalización
  - Realización
- Reglas UML: sintácticas y semánticas (nombre, visibilidad, etc.)

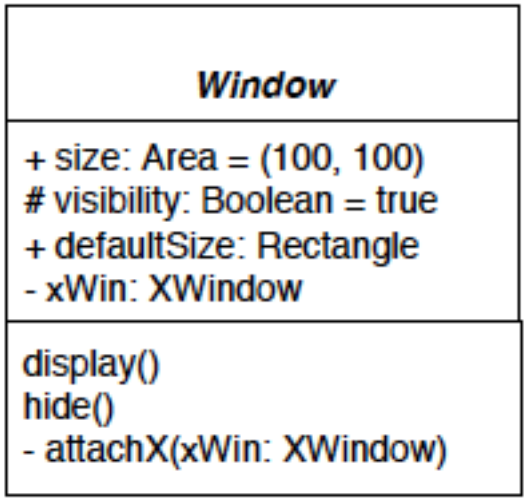
UML: Elementos Estructurales

- Partes estáticas de un modelo. Representan conceptos o cosas materiales
- También llamados clasificadores
- Tipos básicos
  - Interfaces. Describen el comportamiento externo de un componente del sistema
  - Clases. Colección de elementos que implementa una interfaz
  - Caso de uso. Descripción de un conjunto de secuencias de acciones que ejecuta un sistema
  - Componentes. Módulo de un sistema que oculta su implementación tras un conjunto de interfaces externas. Solo existen en tiempo de ejecución
  - Artefactos. Parte física de un sistema con información física (bits)
  - Nodo. Elemento físico que representa un recurso computacional

UML: Elementos Estructurales



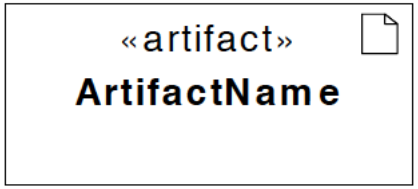
Interfaz ISensor



Clase Window con atributos

Caso de uso Withdraw

Componente. Ej. Formulario

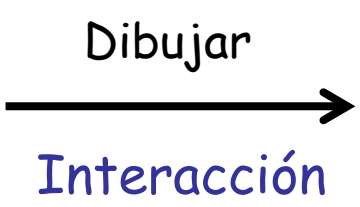


Artefacto. Ej. ventana.dll

Nodo. Ej. Servidor

UML: Elementos de Comportamiento

- Partes dinámicas de los modelos UML
- Representan verbos, comportamiento en el tiempo y el espacio
- Tipos básicos
  - Interacción. Intercambio de mensajes entre objetos
  - Máquina de estados. Secuencia de estados por los que pasa un objeto

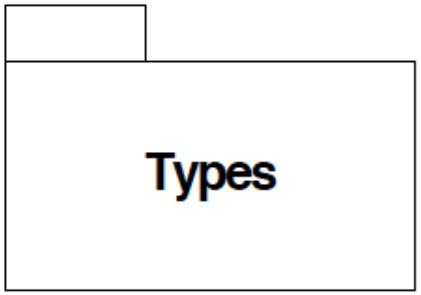


Interacción

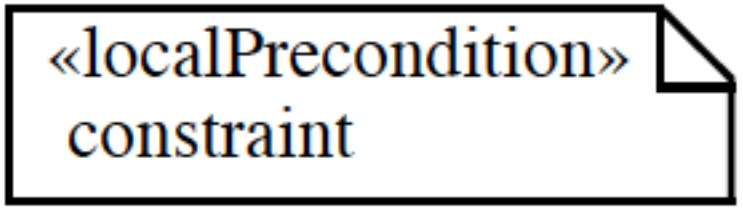
Estado/Acción

UML: Elementos de Agrupación y Anotación

- Elementos de Agrupación
  - Partes organizativas de un modelo
  - Tipo básico: el paquete. Agrupa varias clases
- Elementos de Anotación
  - Partes explicativas de un modelo
  - Tipo básico: la nota. Muestra restricciones y comentarios



Paquete Types

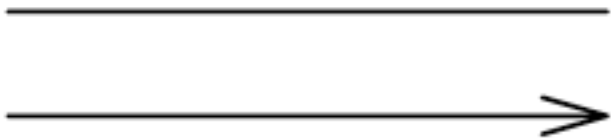


Restricción

UML: Relaciones

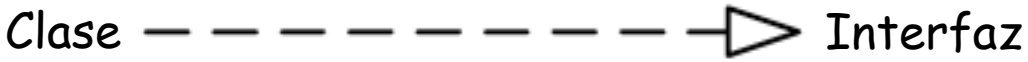
- Dependencias
  - Relación semántica entre dos elementos.
  - Un elemento influye en el otro.

- Asociación
  - Relación estructural entre clases



- Generalización/Especialización
  - Relación en la que el elemento hijo es una especificación del elemento generalizado (elemento padre)

- Realización
  - Un clasificador especifica un contrato que el elemento realizador cumplirá



UML: Diagramas

- Distintas facetas de un mismo modelo
- UML 2.4.1 incluye 14 tipos
- Estudiaremos solo los más importantes para la asignatura

Structure diagrams

1. Class diagram
2. Composite structure diagram (*)
3. Component diagram
4. Deployment diagram
5. Object diagram
6. Package diagram
7. Profile diagram

Behavior diagrams

8. Use-case diagram
9. State machine diagram
10. Activity diagram

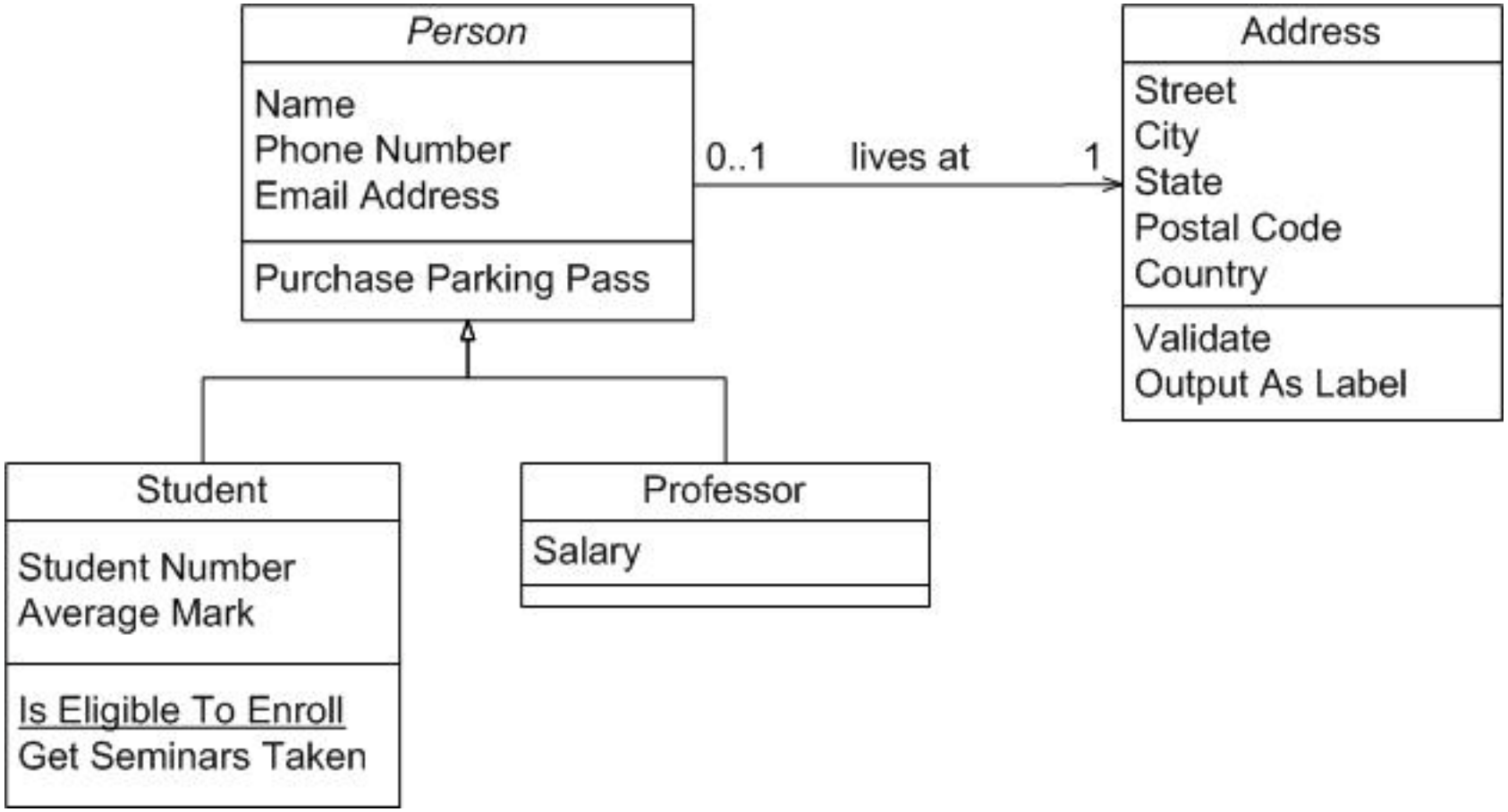
Interaction diagrams

11. Sequence diagram
12. Communication diagram
13. Interaction overview diagram (*)
14. Timing diagram (*)

(*) not existing in UML 1.x, added in UML 2.0

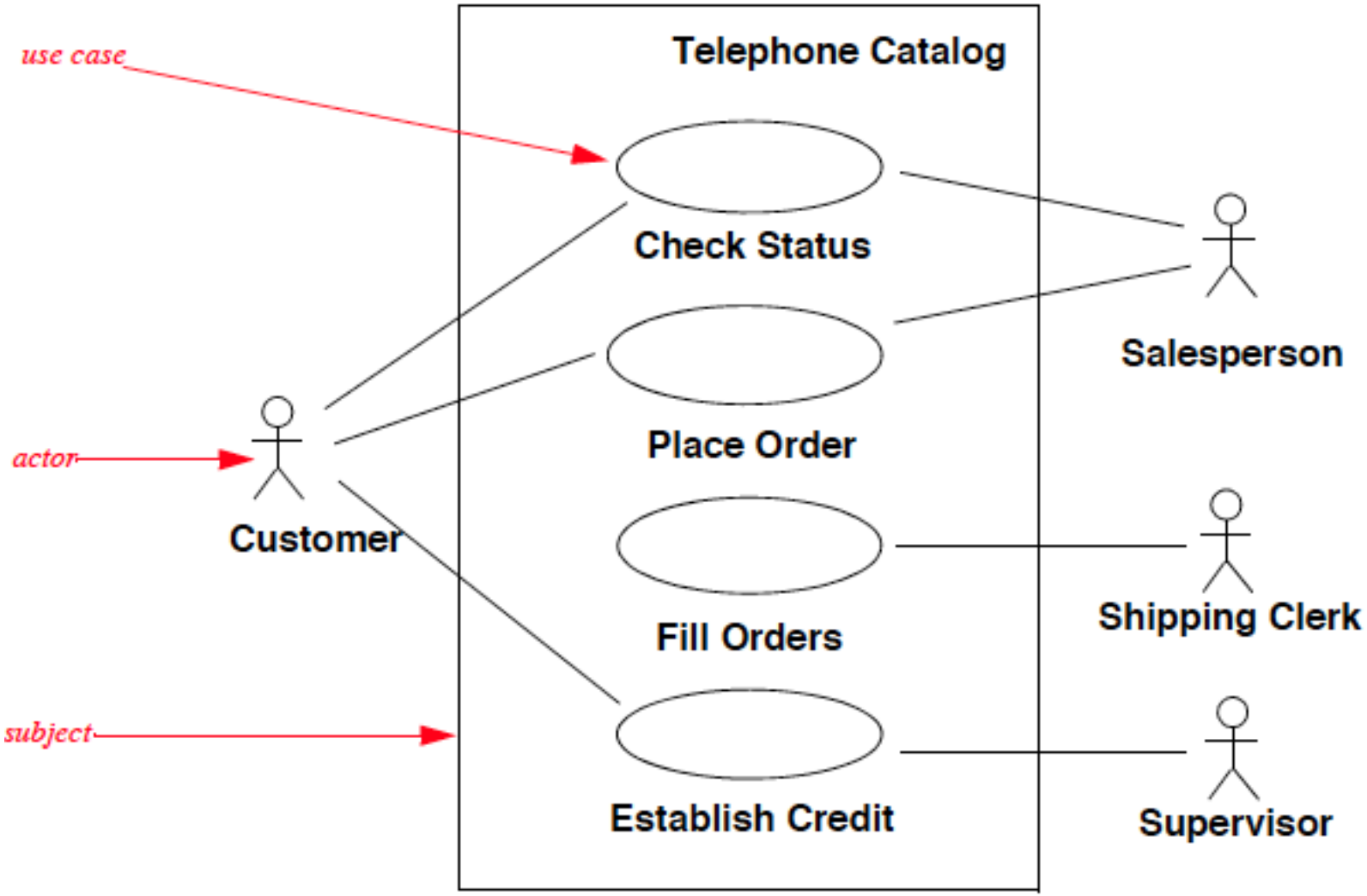
UML: Diagramas de Clases

- Muestra un conjunto de clases, interfaces y colaboraciones (cooperación entre varios elementos)
- También muestra las relaciones entre varios elementos



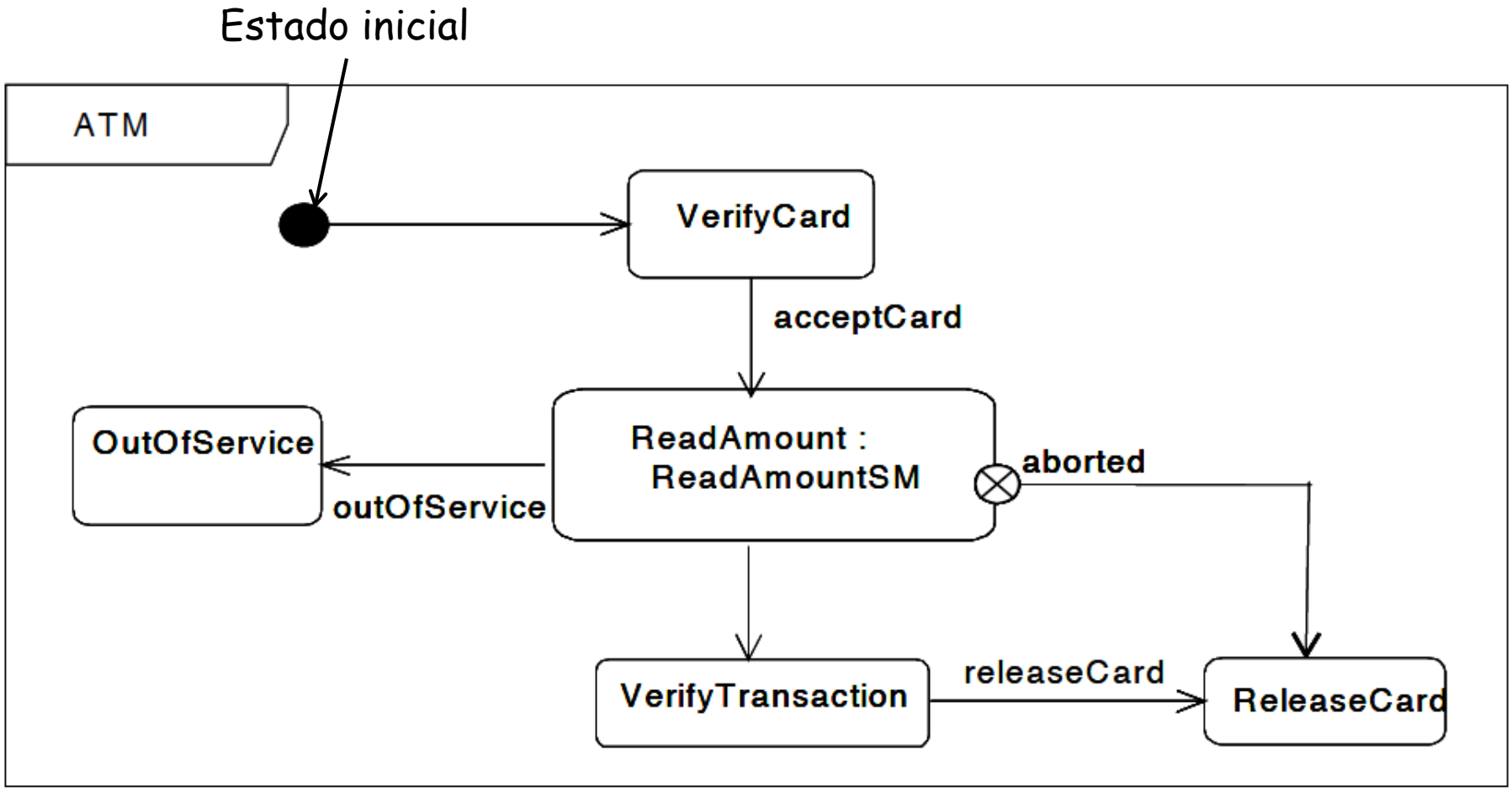
UML: Diagramas de Casos de Uso

- Muestra un conjunto de casos de uso, actores y sus relaciones



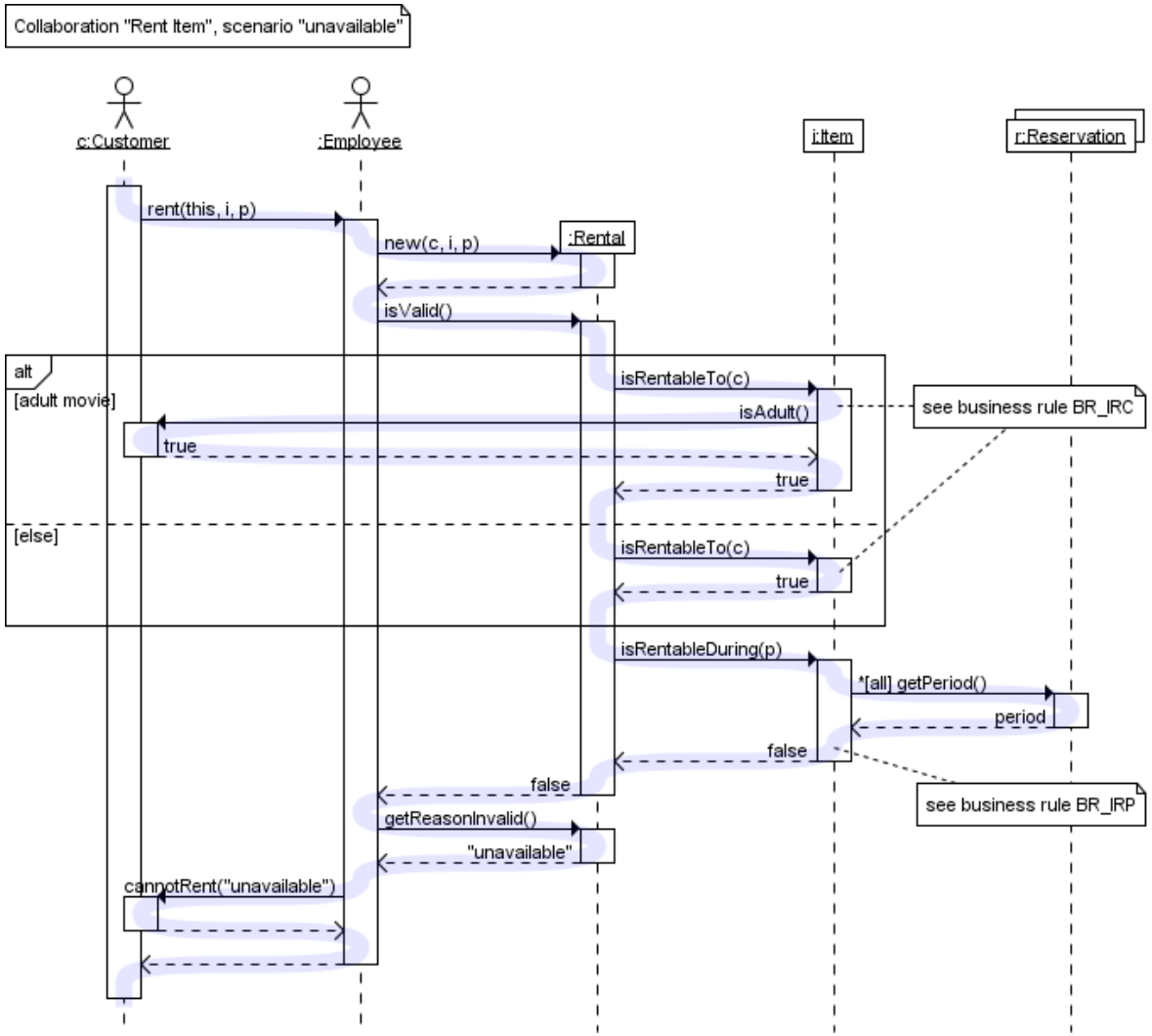
UML: Diagramas de estados

- Muestra una máquina de estados: estados, transiciones, eventos y actividades



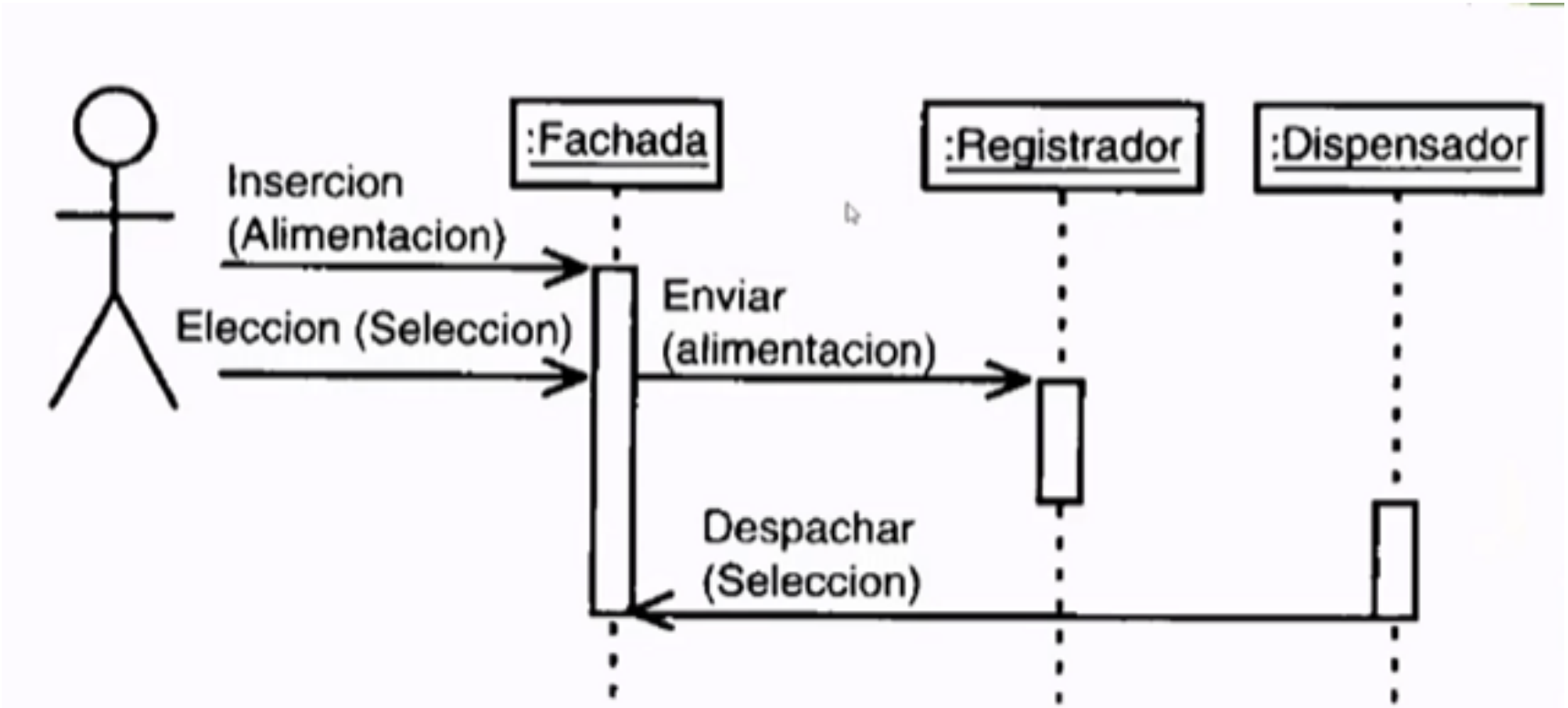
UML: Diagramas de secuencia

- Modela interacción entre objetos a lo largo del tiempo
- Uno por caso de uso (granularidad más fina)



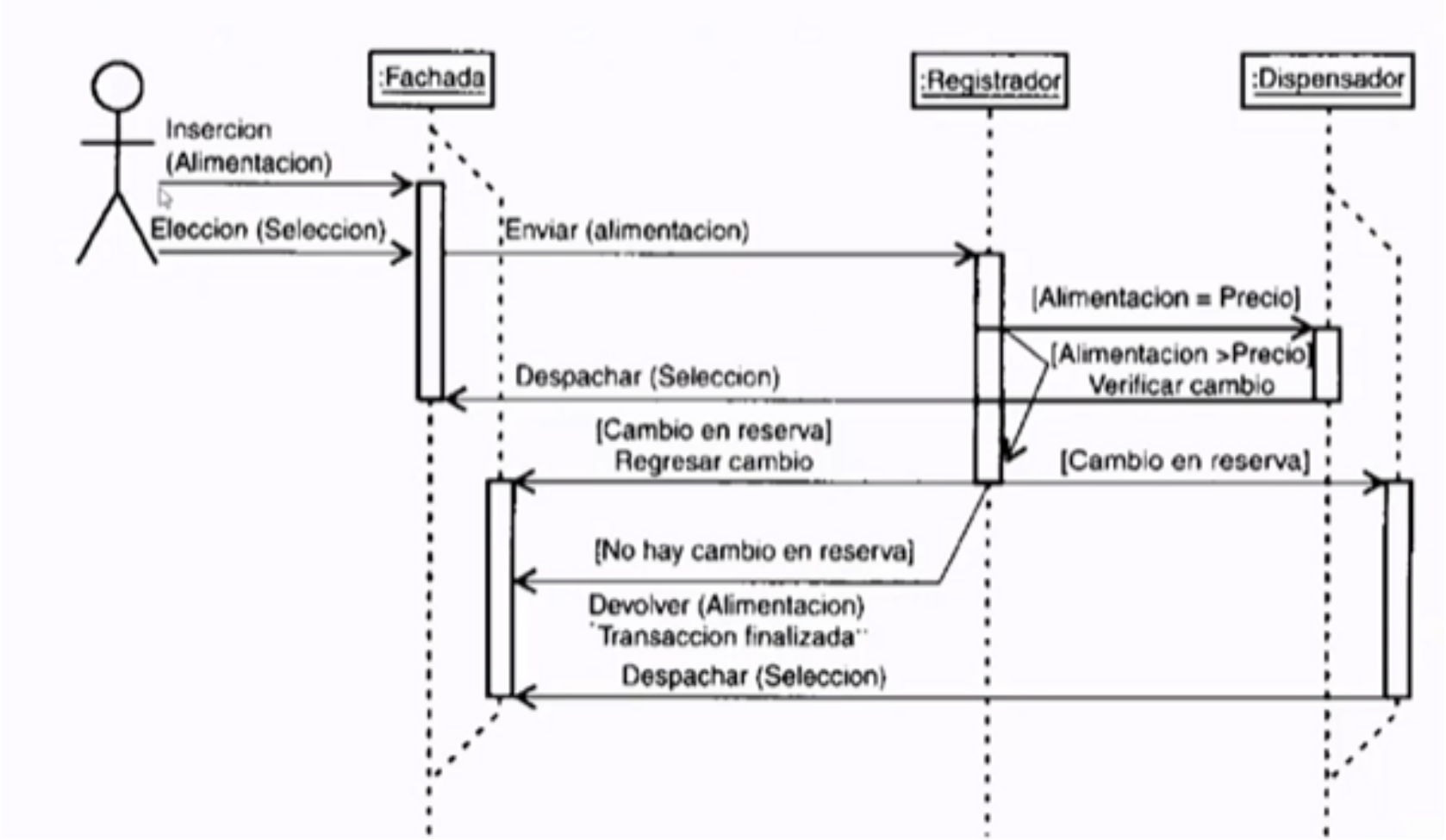
UML: Diagramas de secuencia

- Modela interacción entre objetos a lo largo del tiempo
- Uno por caso de uso (granularidad más fina)



UML: Diagramas de secuencia

- Modela interacción entre objetos a lo largo del tiempo
- Uno por caso de uso (granularidad más fina)



UML: Diagrama de actividad/flujo

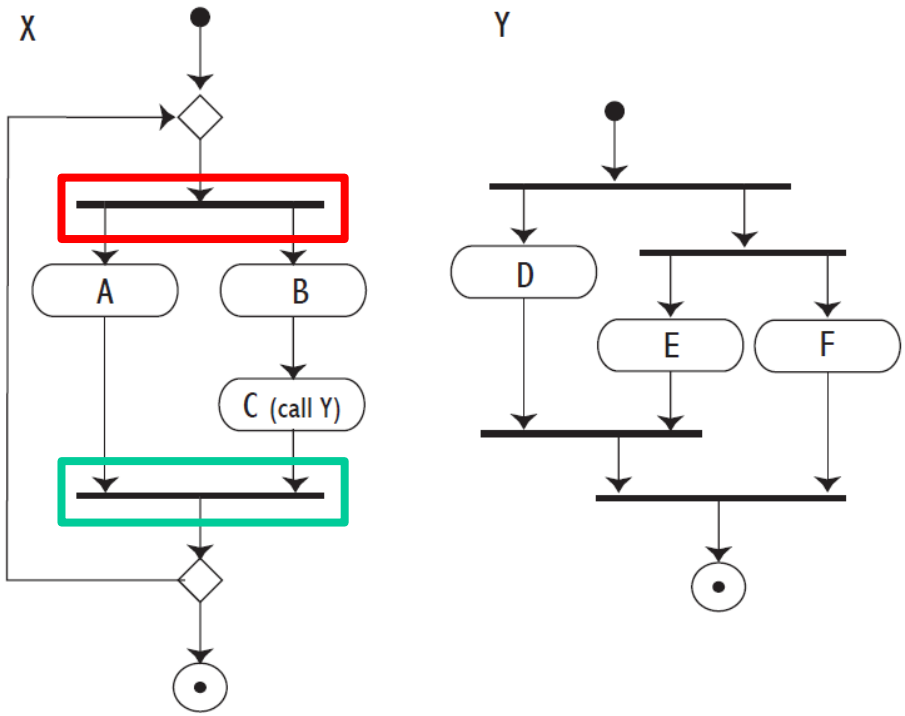
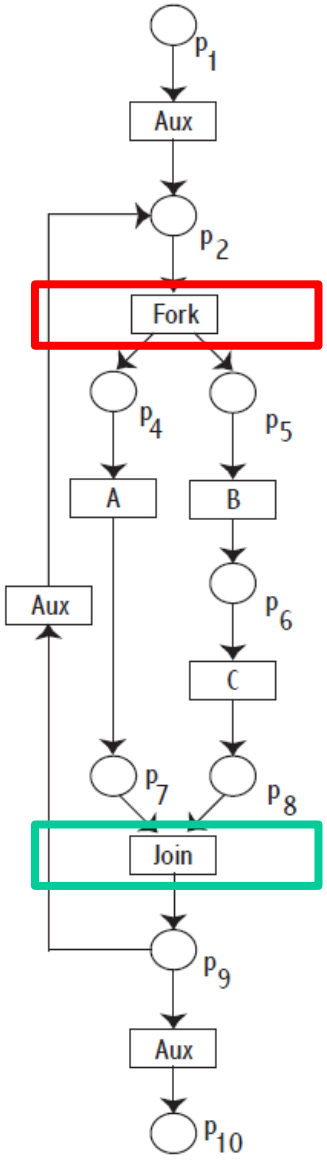
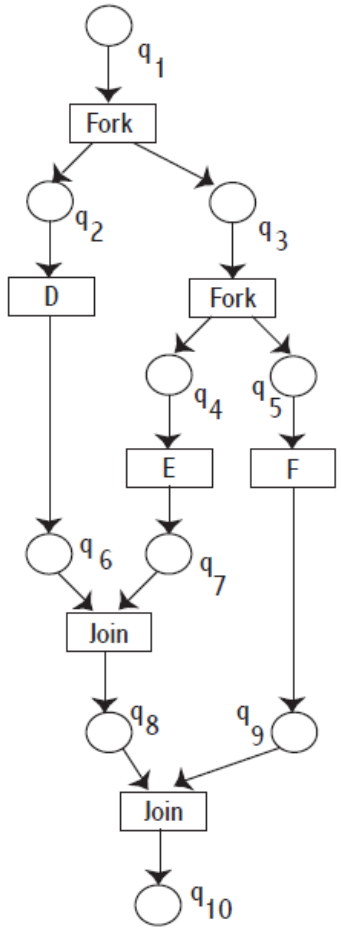


Diagrama de actividad





Red de Petri

H. Störrle, "**Semantics of UML 2.0 Activities**", 2004

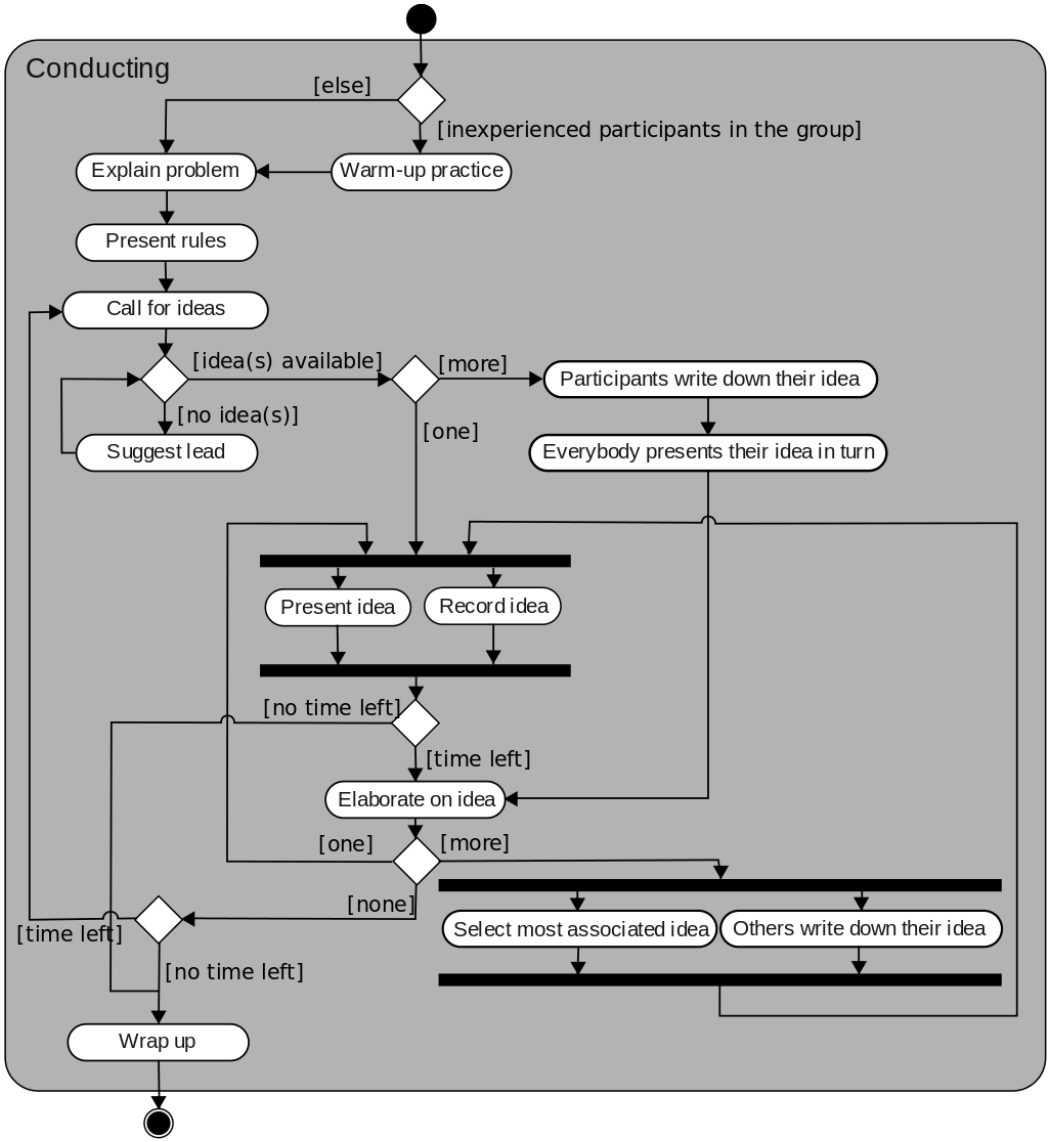
UML: Diagrama de actividad

- Parecido a una FSM que soporta concurrencia (redes de Petri)
- Componentes
  - Elipse: representa acciones
  - Rombo: representa decisiones
    - Condiciones if-then-else, while, repeat
  - Barras: representan un fork o un join
  - Círculo negro: estado inicial del diagrama de flujo
    - Si está rodeado de otra circunferencia, representa el estado final

UML: Diagrama de actividad

UML activity diagrams in version 2.x can be used in various domains, e.g. ***in design of embedded systems****. It is possible to verify such a specification using model checking technique.*

I. Grobelna, et al "Model Checking of UML Activity Diagrams in Logic Controllers Design", 2014

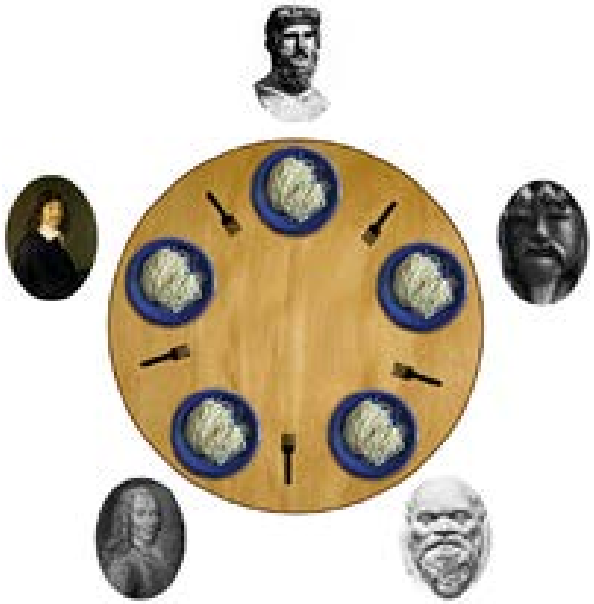


UML: Automatizando el Diseño

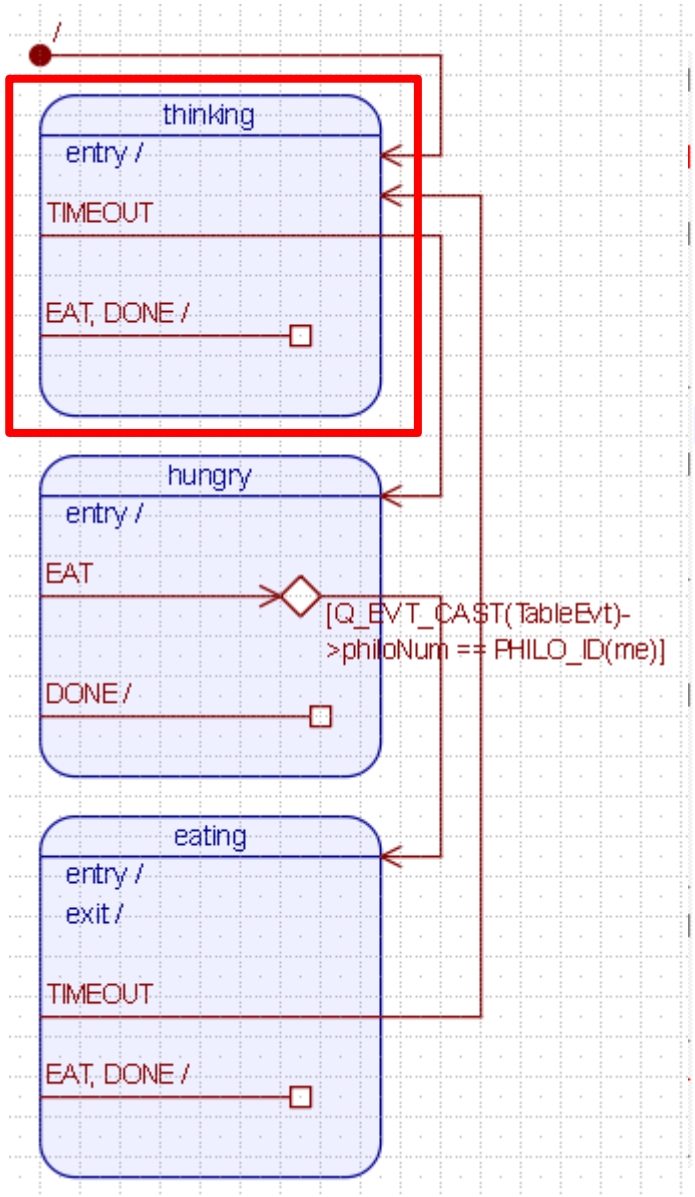
- UML se ha convertido en un estándar
- Múltiples herramientas disponibles que se encargan de generar el esqueleto de un proyecto (backbone) a partir del UML
  - Visual Paradigm (C, C++, Python)
  - QP + QM (Quantum Leaps)
    - QC, QCPP, QN
    - Múltiples soportes: ARM, Arduino, Raspberry
  - Embedded UML Studio (Keil)

UML: Automatizando el Diseño

- Un caso de estudio: Cena de Filósofos
  - 5 filósofos en una mesa, con un tenedor a cada lado
  - Cada filósofo puede pensar o comer
  - Solo pueden comer si tienen tenedor a ambos lados
  - Comida infinita
  - Objetivo: diseñar un sistema tal que los filósofos no se mueran de hambre

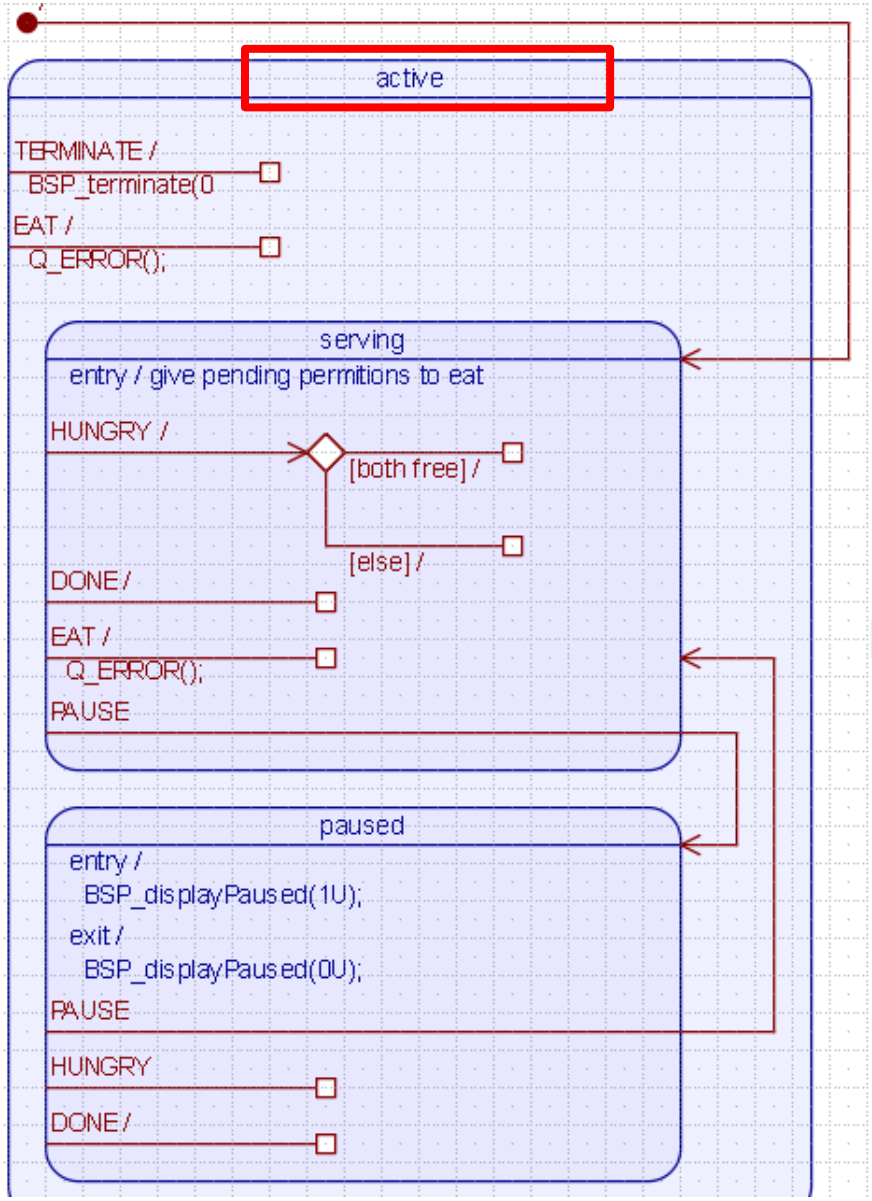


UML: Automatizando el Diseño




```
static QState Philo_thinking(Philo * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        /* @(/2/0/1/1) */
        case Q_ENTRY_SIG: {
            QTimeEvt_postIn(&me->timeEvt, &me->super, THINK_TIME);
            status = Q_HANDLED();
            break;
        }
        /* @(/2/0/1/1/0) */
        case TIMEOUT_SIG: {
            status = Q_TRAN(&Philo_hungry);
            break;
        }
        /* @(/2/0/1/1/1) */
        case EAT_SIG: /* intentionally fall through */
        case DONE_SIG: {
            /* EAT or DONE must be for other Philos than this one */
            Q_ASSERT(Q_EVT_CAST(TableEvt)->philoNum != PHILO_ID(me));
            status = Q_HANDLED();
            break;
        }
        default: {
            status = Q_SUPER(&QHsm_top);
            break;
        }
    }
    return status;
}
```

UML: Automatizando el Diseño




```
static QState Table_active(Table * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        /* @(/2/1/2/1/0) */
        case TERMINATE_SIG: {
            BSP_terminate(0);
            status = Q_HANDLED();
            break;
        }
        /* @(/2/1/2/1/1) */
        case EAT_SIG: {
            Q_ERROR();
            status = Q_HANDLED();
            break;
        }
        default: {
            status = Q_SUPER(&QHsm_top);
            break;
        }
    }
    return status;
}
```