LENGUAJES DE DOMINIO ESPECÍFICO

Desarrollo Automatizado de Software

4º Ingeniería Informática

Universidad Autónoma de Madrid

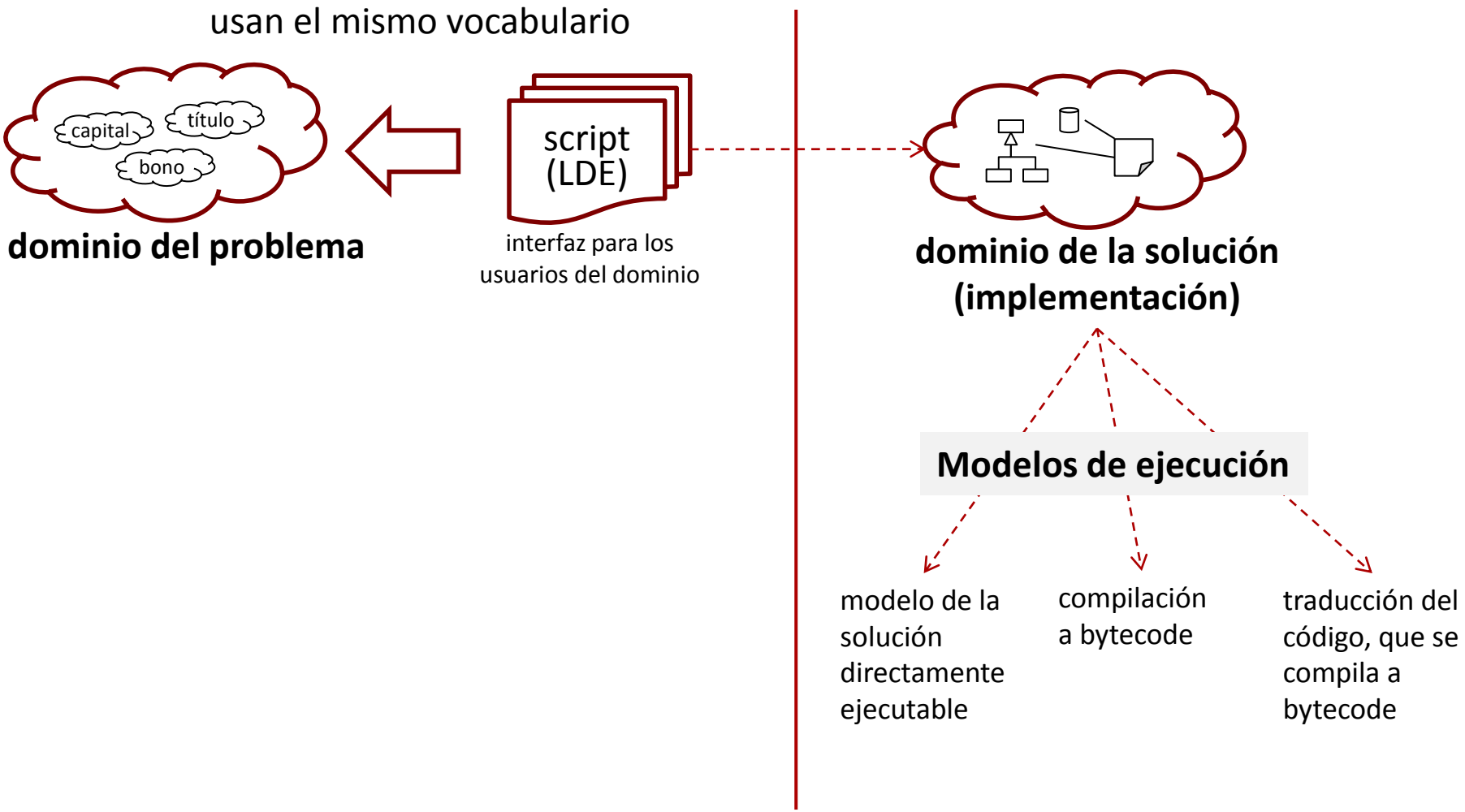
Lenguajes de dominio específico

- Lenguajes de programación orientados a un problema específico (vs lenguajes de propósito general).
- Proporcionan abstracciones del dominio del problema.
- Centrados en el dominio del problema, no en el dominio de la solución/implementación.
- Ejemplos: VHDL para describir circuitos, ANT para construir sistemas software, SQL para bases de datos relacionales…

ejemplo de script ANT:

```
<target name="jar" depends="compile">
  <mkdir dir="${build.dist}">
  <jar jarfile="${build.dist}/${name}-${version}.jar">
    <fileset dir="${build.classes}" includes="*" />
    <fileset dir="${src.dir}" includes="*" />
  </jar>
</target>
```

Lenguajes de dominio específico



Clasificación

- Lenguajes internos: usan la infraestructura de un lenguaje existente (ej. Ruby, Scala, Groovy) para construir el LDE encima.
  - Menor tiempo de desarrollo
  - Atado a la sintaxis del lenguaje anfitrión

- Lenguajes externos: se desarrollan desde cero. Similar a implementar un nuevo lenguaje de programación (implica desarrollar parser, analizador sintáctico, intérprete o compilador, editor, etc.).
  - Mayor complejidad
  - Posibilidad de usar frameworks de desarrollo de lenguajes (ej. Xtext)
  - Única opción si el lenguaje no es textual

¿Cuándo usarlos?

- Ventajas
  - Son expresivos y concisos
  - Trabajan a un mayor nivel de abstracción
  - Reutilizan conocimiento del dominio
  - Mayor beneficio a largo plazo
  - Desarrollo basado en LDEs es escalable

- Desventajas
  - El diseño de lenguajes es difícil, requiere programadores expertos
  - Coste de desarrollo
  - En ocasiones, puede incurrir en problemas de eficiencia
  - Necesitan soporte por herramientas (integración en IDEs, pruebas…)
  - Otro lenguaje más que aprender
  - Cacofonía de lenguajes (composición de distintos LDEs)

Ejemplo

- Compañía que construye sistemas de seguridad para el hogar.
- El sistema permite abrir compartimentos ocultos cuando se activan en secuencia una serie de sensores ocultos en la casa.
  - Por ejemplo: abrir un cajón, encender la lámpara y coger un libro de una estantería. Puede haber otras secuencias, a convenir con el usuario.
  - Sensores que resetean la secuencia (por ejemplo, girar la llave de la puerta).
- Cuando se activan, los sensores mandan señales con códigos (por ejemplo, D2OP “Drawer Open”) a un controlador.
- El controlador va embebido en un dispositivo, que también se instala en la casa. Los dispositivos se programan en Java.
- Cada instalación (cada casa) tiene una configuración distinta de sensores y secuencias de activación, que hay que programar en cada controlador.

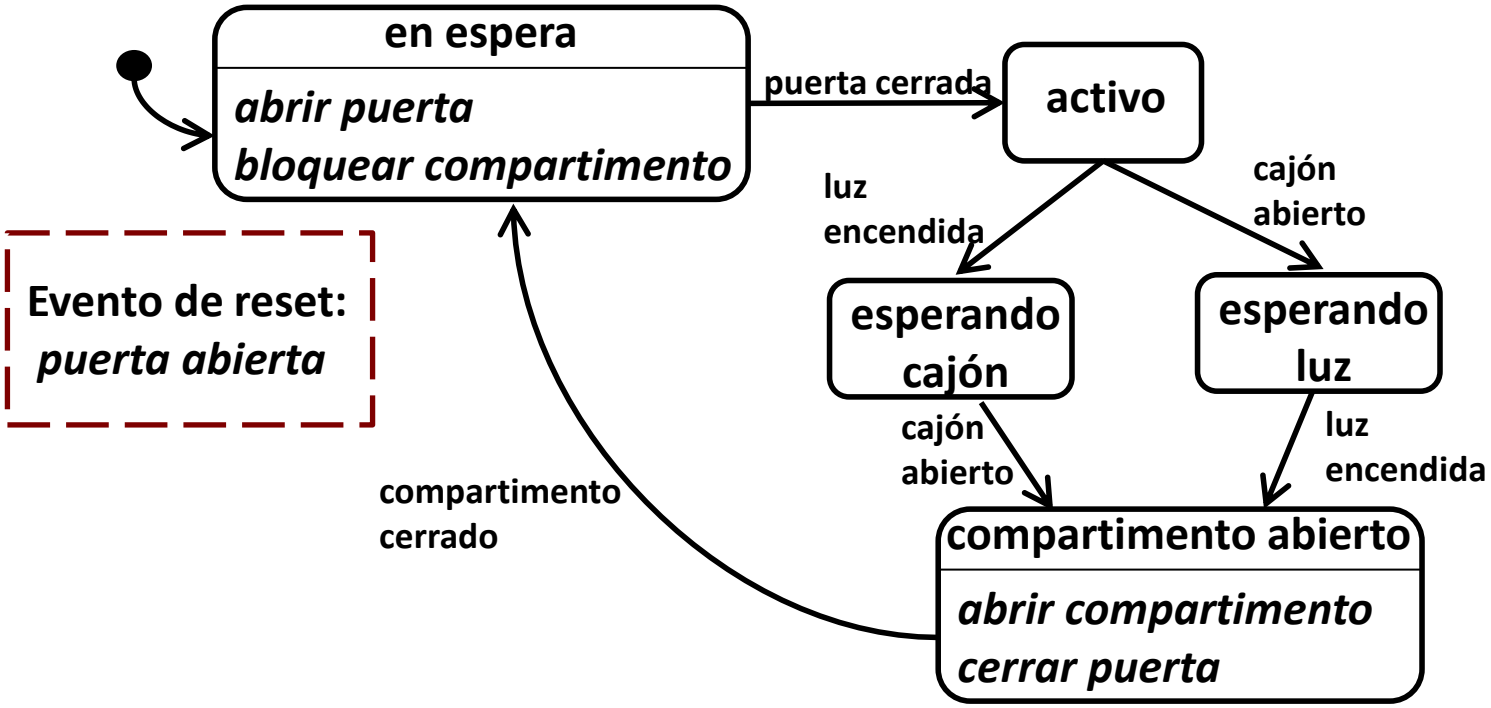
Un ejemplo de controlador

- Abrir el compartimento secreto cuando:
  - Se cierra la puerta, se abre el segundo cajón de la cómoda, y se enciende la luz de la mesilla (estos dos últimos eventos en cualquier orden).
  - Cuando se abre el compartimento secreto, además se cierra la puerta.
  - Cuando se cierra el compartimento, queremos volver a abrir la puerta.
  - En cualquier momento, abrir la puerta resetea la secuencia.

- ¿Cómo expresar de manera precisa estos requisitos?
  - Con una máquina de estados!

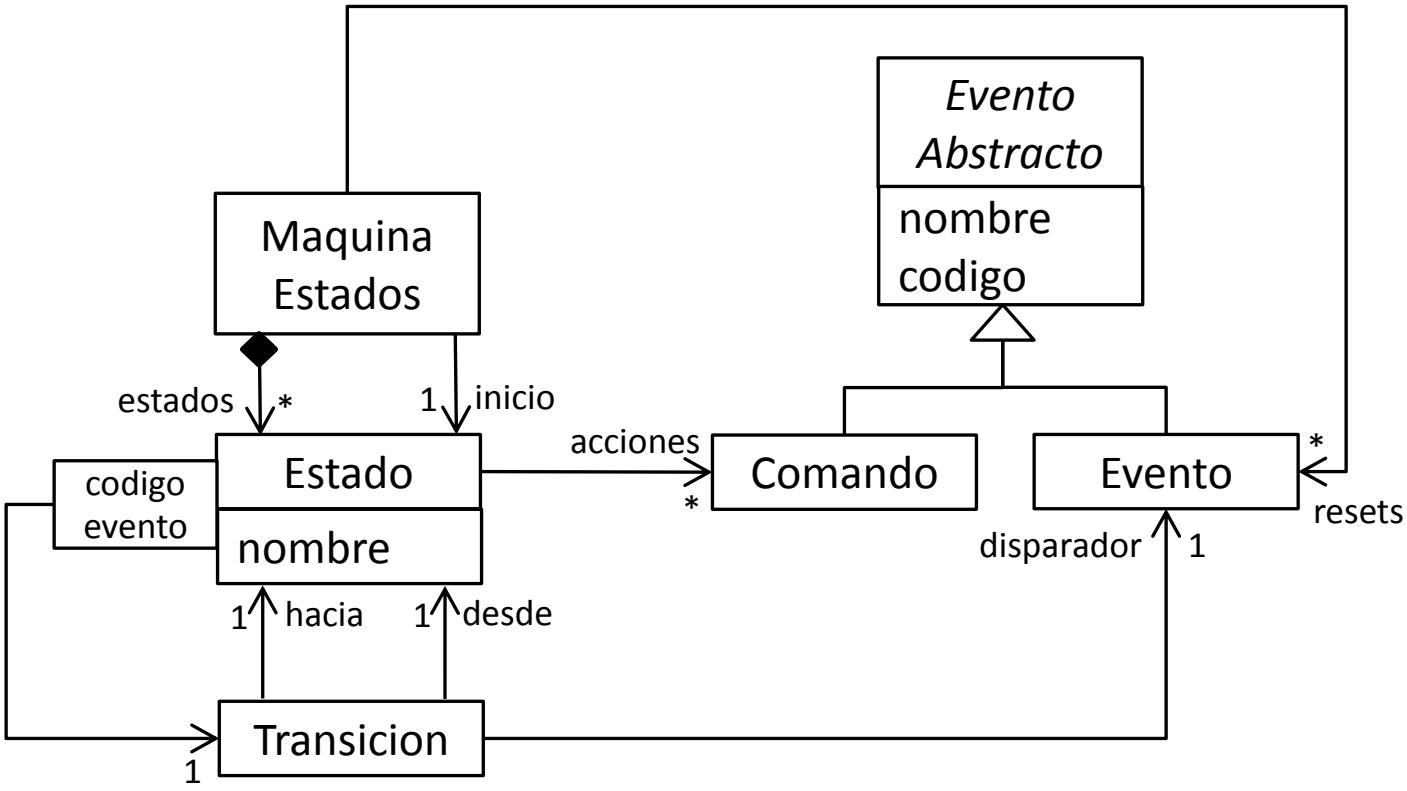
Un ejemplo de controlador

- Máquina de estados para el controlador:



- El concepto “máquina de estados” es una buena abstracción de cómo debe funcionar el controlador.

Diseño del Modelo del Controlador



Código Java del Modelo del Controlador

```
public class EventoAbstracto {
  private String nombre, codigo;

  public EventoAbstracto (String n, String c) {
    this.nombre = n;
    this.codigo = c;
  }

  public String getNombre() { return nombre; }
  public String getCodigo() { return codigo; }

  //...
}

public class Comando extends EventoAbstracto { ... }

public class Evento extends EventoAbstracto { ... }
```

Código Java del Modelo del Controlador

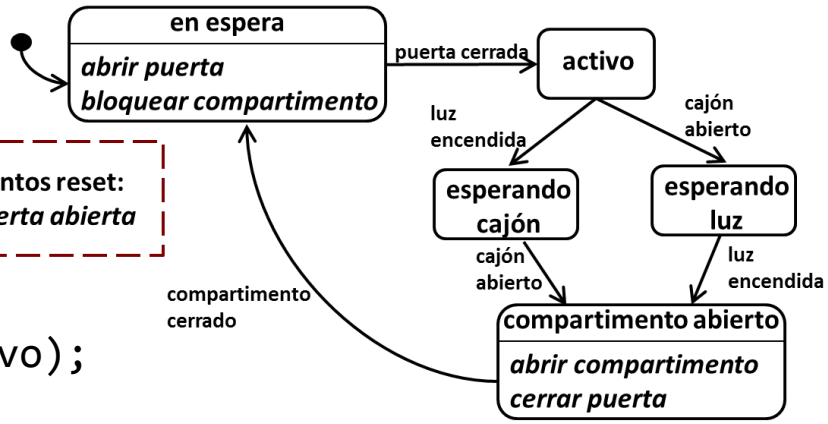
```
public class MaquinaEstados {

  private List<Estado> estados = new ArrayList<Estado>();
  private Estado inicial;
  private List<Evento> resets = new ArrayList<Evento>();

  public void addResets (Evento... resets) {
    for (Evento e: resets) this.resets.add(e);
  }

  //...
}
```

Código Java del Modelo del Controlador

```
public class Controlador {
  private Estado actual;
  private MaquinaEstados maquina;

  public void actuar (String codigoEvento) {
    if (actual.tieneTransicion(codigoEvento))
      moverA(actual.estadoDestino(codigoEvento));
    else if (maquina.esReset(codigoEvento))
      moverA(maquina.getInicial());
    // ignorar eventos desconocidos
   }

  private void moverA(Estado estadoDestino) {
    this.actual = estadoDestino;
    this.actual.ejecutarAcciones();
  }

  //...
}
```

Código Java del controlador ejemplo

```
Evento puertaCerrada = new Evento("puerta cerrada", "PTCL");
Evento luzEncendida  = new Evento("luz encendida", "LUON");
// ...
Comando bloquearCompartimento = new Comando ("bloquear...", "CMLK");
Comando abrirPuerta   = new Comando ("abrir puerta", "PTUL");
// ...
Estado inicial = new Estado("en espera");
Estado activo  = new Estado("activo");
// ...
MaquinaEstados maquinaEjemplo =
new MaquinaEstados(inicial);

inicial.añadeTransicion(puertaCerrada, activo);
inicial.añadeAccion(abrirPuerta);
inicial.añadeAccion(bloquearCompartimento);

activo.añadeTransicion(luzEncendida, esperandoCajon);
activo.añadeTransicion(cajonAbierto, esperandoLuz);
// ...
maquinaEjemplo.addResets(puertaAbierta);
```

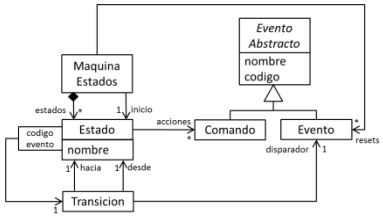


Estrategia de diseño

- Hemos creado un ***framework*** (o ***librería***) con el código común:
  - Clases Estado, Transicion, MaquinaEstados, Controlador.
- Añadimos código de ***configuración*** para describir cada controlador concreto.
- Separar código común del variable.

- El código de configuración crea un "*modelo*" interno (una máquina de estados).
- En este caso, la clase **`Controlador`** interpreta dicho modelo.

Estrategia de diseño

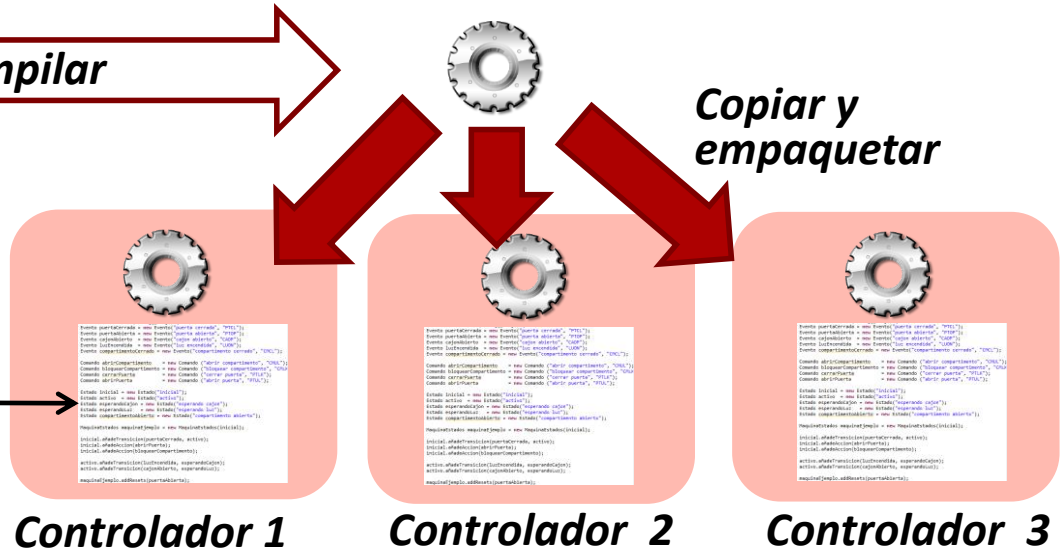


Framework máquina estados

compilar

Copiar y empaquetar

código de configuración



Controlador 1 ***Controlador 2*** ***Controlador 3***

- ¿Es realmente ***Java*** la mejor alternativa para describir el código de configuración?

Alternativas para la configuración: XML

- XML
- Haría falta un ***parser*** de este XML, integrado en el framework.

```
<maquinaEstados inicio="en espera">
  <evento nombre="puerta cerrada" codigo="PTCL"/>
  <evento nombre="luz encendida" codigo="LUON"/>
  …
  <comando nombre="abrir compartimento", "CMUL"/>
  <comando nombre="abrir puerta", "PTUL"/>
  …
  <estado nombre="en espera">
    <transicion evento="puerta cerrada" destino="activo"/>
    <accion comando="luz encendida"/>
    <accion comando="bloquear compartimento"/>
  </estado>
  …
  <resets name = "puerta abierta"/>
</maquinaEstados>
```

Alternativas para la configuración: LDE

- Lenguaje de dominio específico ***externo***.
- Sintaxis más clara y concisa.

```
eventos:
  puertaCerrada PTCL
  puertaAbierta PTOP

comandos:
  abrirCompartimento CMUL
  abrirPuerta PTUL
  bloquearCompartimento CMLK

estado inicial:
  acciones {abrirPuerta, bloquearCompartimento}
  puertaCerrada => activo

estado activo:
  luzEncendida => esperandoCajon
  cajonAbierto => esperandoLuz
```

Alternativas para la configuración: LDE

- Lenguaje de dominio específico ***interno***.
- En un lenguaje dinámico como **Ruby.**

```
evento :puertaCerrada, “PTCL”
evento :puertaAbierta, “PTOP”

comando :abrirCompartimento, “CMUL”
comando :abrirPuerta, “PTUL”
comando :bloquearCompartimento, “CMLK”

estado :inicial do
  acciones :abrirPuerta, :bloquearCompartimento
  transiciones :puertaCerrada => :activo
end

estado :activo do
  transiciones :luzEncendida => :esperandoCajon,
               :cajonAbierto => :esperandoLuz
end
```

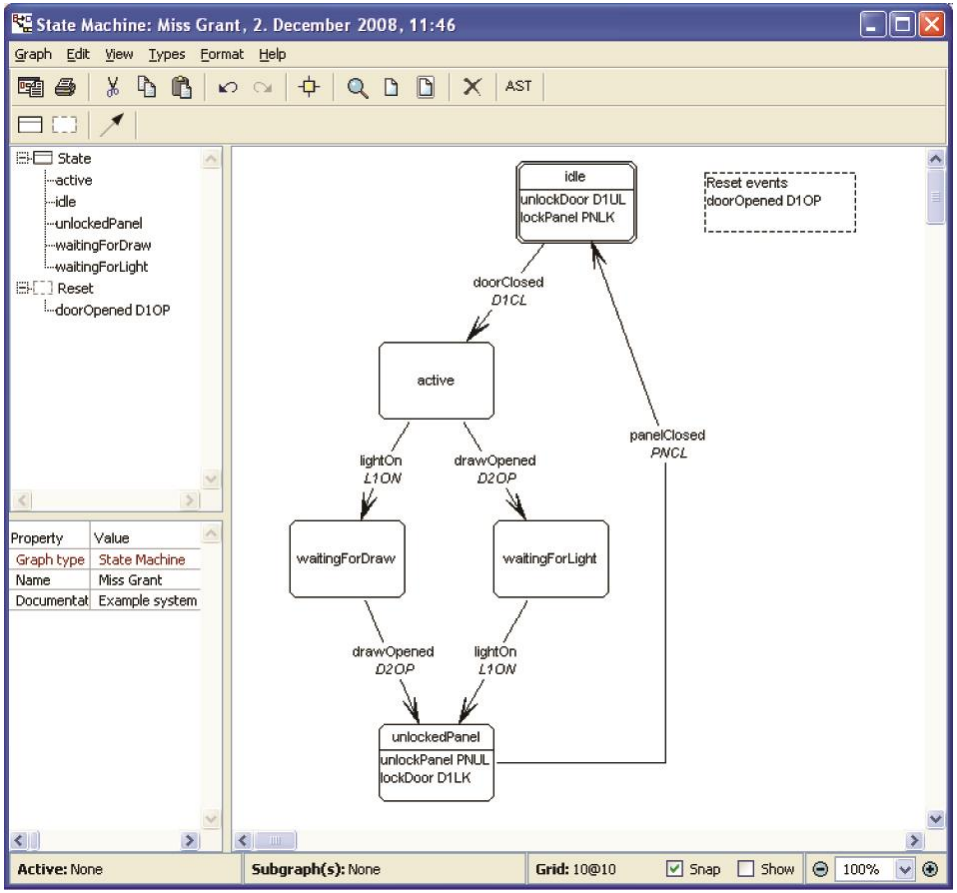
Alternativas para la configuración. API.

- "*Method Chaining*", "*Fluent API*".
- Similar a LDE interno, pero en un lenguaje estático como Java

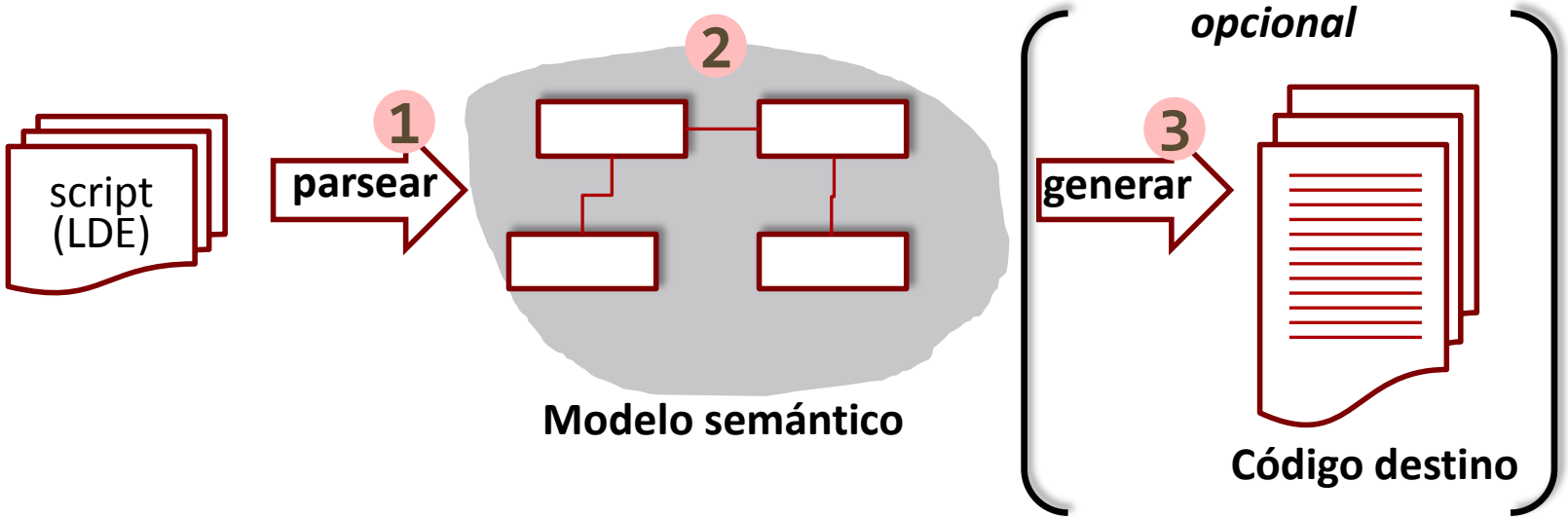
```
public class MaquinaEstadosBasica extends BuilderMaquinaEstados {
  Eventos  puertaCerrada, luzEncendida, ...;
  Comandos abrirCompartimento, bloquearCompartimento, ...;
  Estados  inicial, activo, ...;
  Resets   puertaAbierta;
  protected void definirMaquinaEstados() {
    puertaCerrada.codigo("PTCL");
    cajonAbierto.codigo("CAOP");

    inicial
       .acciones(abrirPuerta, bloquearCompartimento)
       .transicion(puertaCerrada).hacia(activo)
    ;

    activo
      .transicion(luzEncendida).hacia(esperandoCajon)
      .transicion(cajonAbierto).hacia(esperandoLuz)
    ; // ...
  }
}
```

Alternativas para la configuración. Lenguaje Visual.

- Lenguaje Visual.
- El editor del lenguaje generaría un fichero de configuración, con alguna de las opciones anteriores.



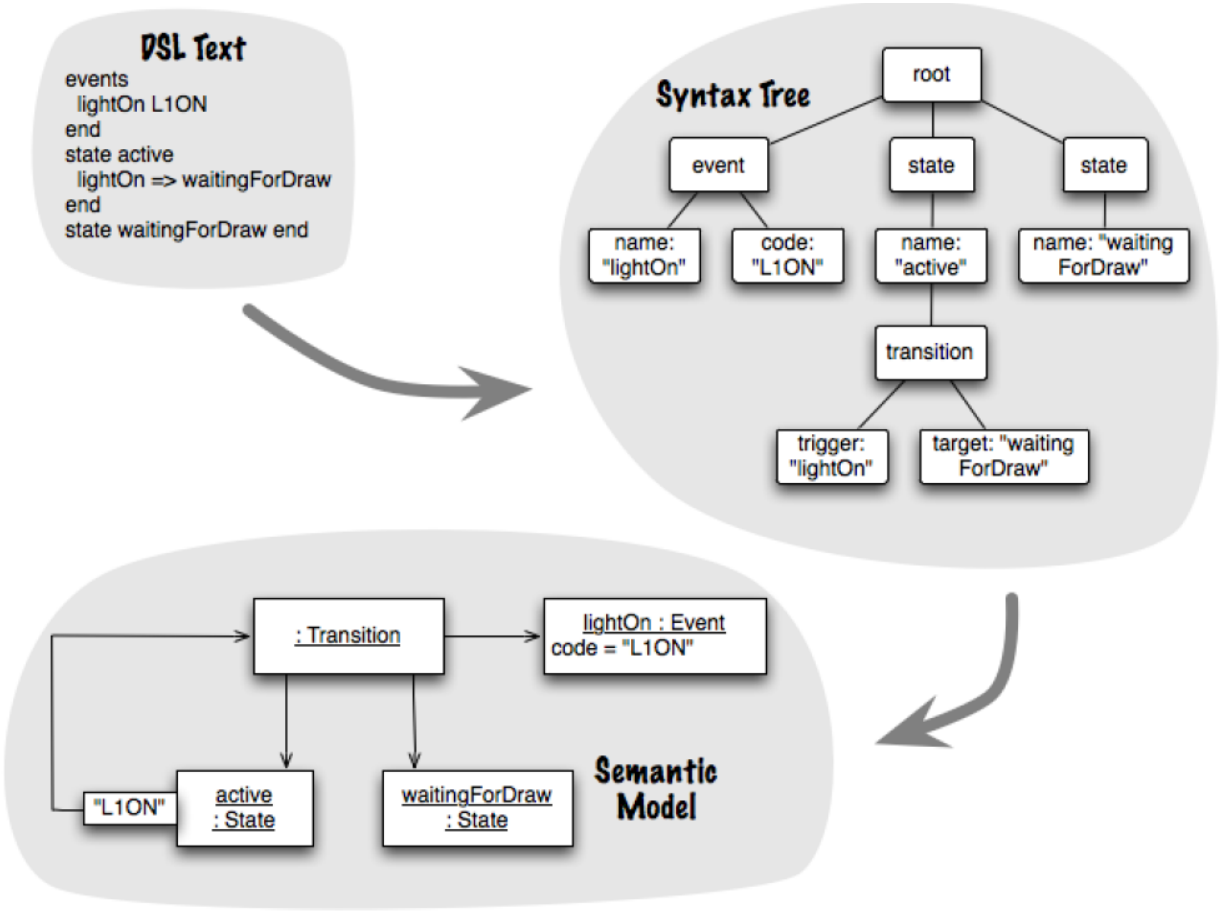
Implementación



1

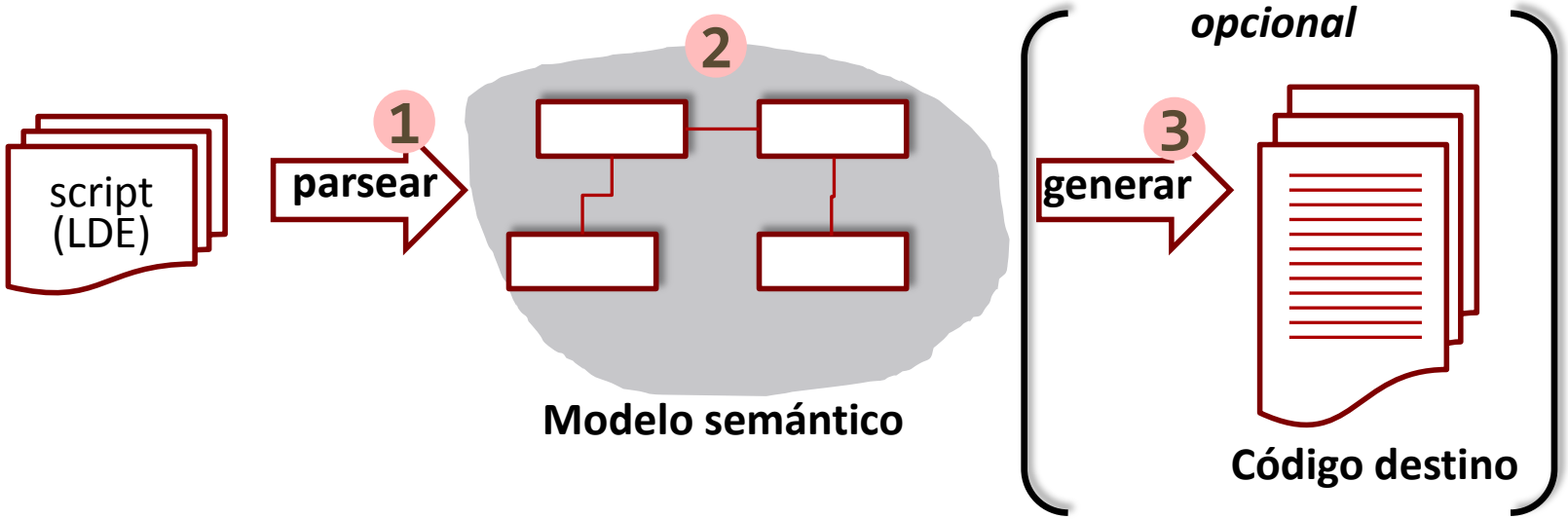
- En el caso de un **LDE interno**, hay dos fases:
  - Parsing del lenguaje anfitrión (p.ej., Ruby)
  - Cuando el programa se ejecuta, se va creando el modelo semántico.
- La gramática del LDE está implícita.

Parsing de un LDE externo



- En el proceso de parsing, es común crear una tabla de símbolos, para resolver referencias.

Implementación

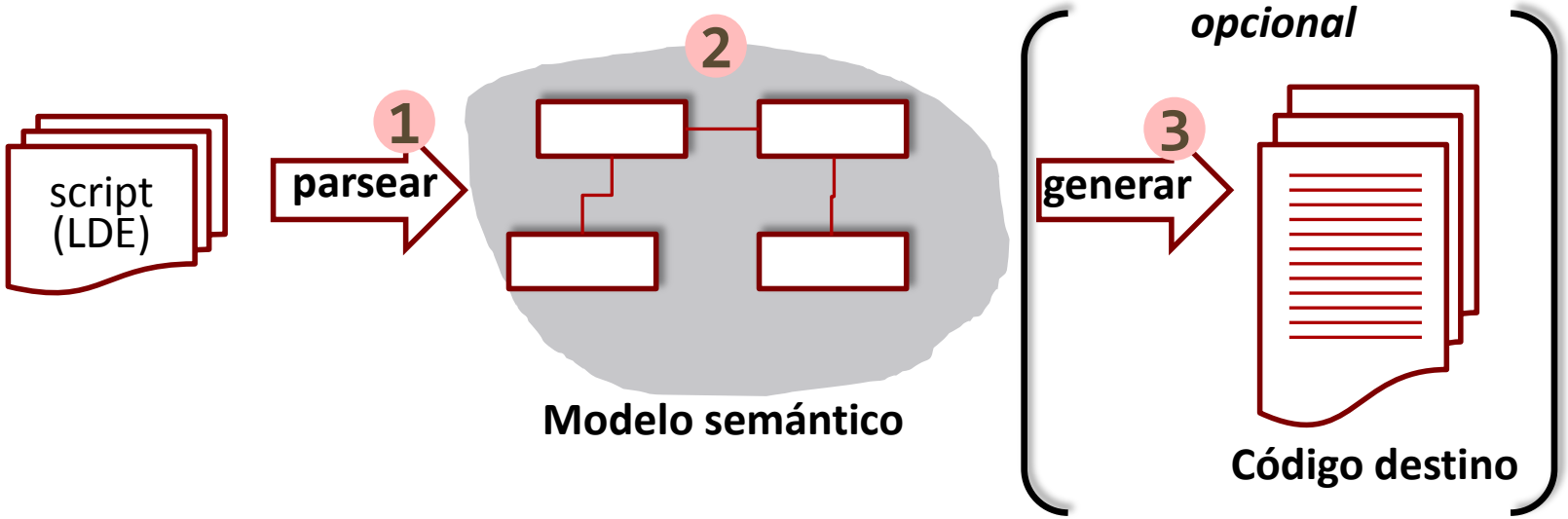


1

- En el caso de un **LDE externo**, suele haber dos fases:
  - Parsing de la sintaxis del LDE a un árbol sintáctico.
  - Conversión del árbol sintáctico al modelo semántico.

- La gramática del LDE es explícita.

Implementación



2

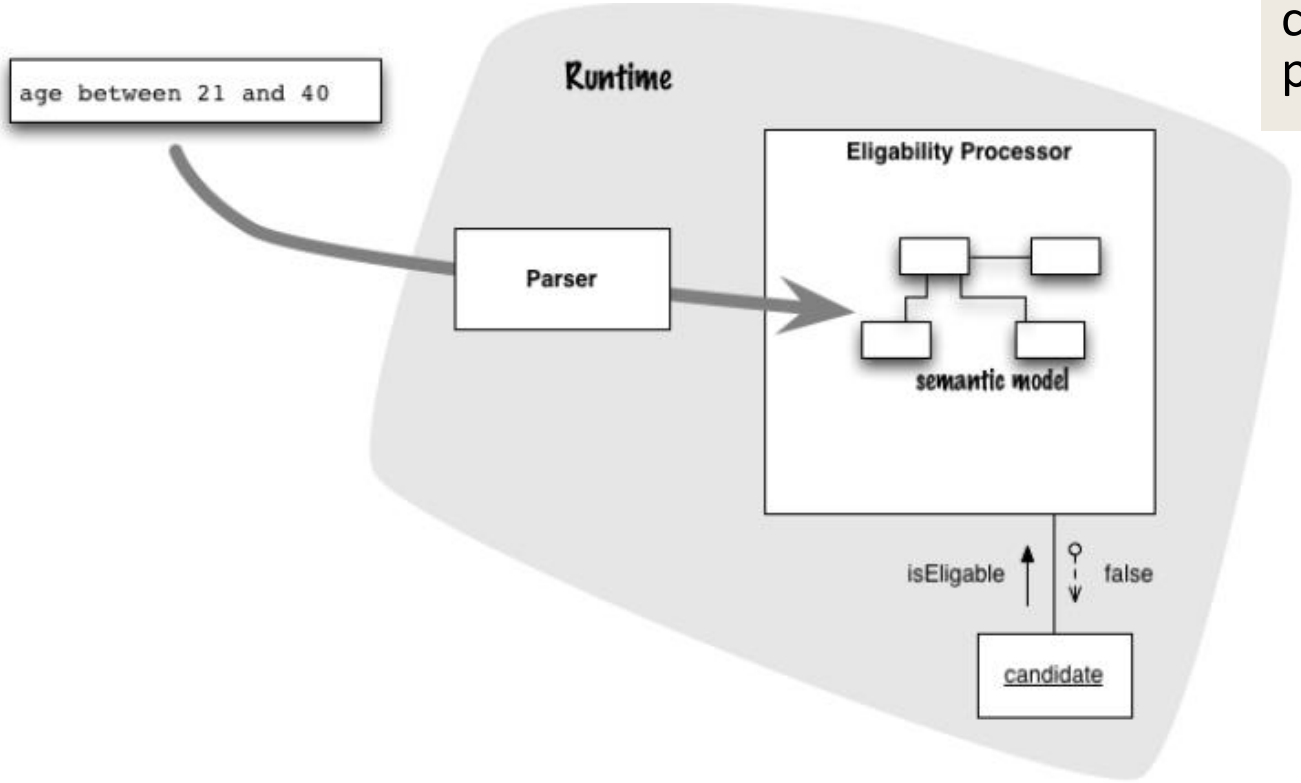
- Una vez construido, el modelo semántico puede ejecutarse.
- Sería un enfoque basado en la creación de un ***intérprete***.

3

- Opcionalmente, podemos generar código
- Sería un enfoque basado en la creación de un ***compilador***.

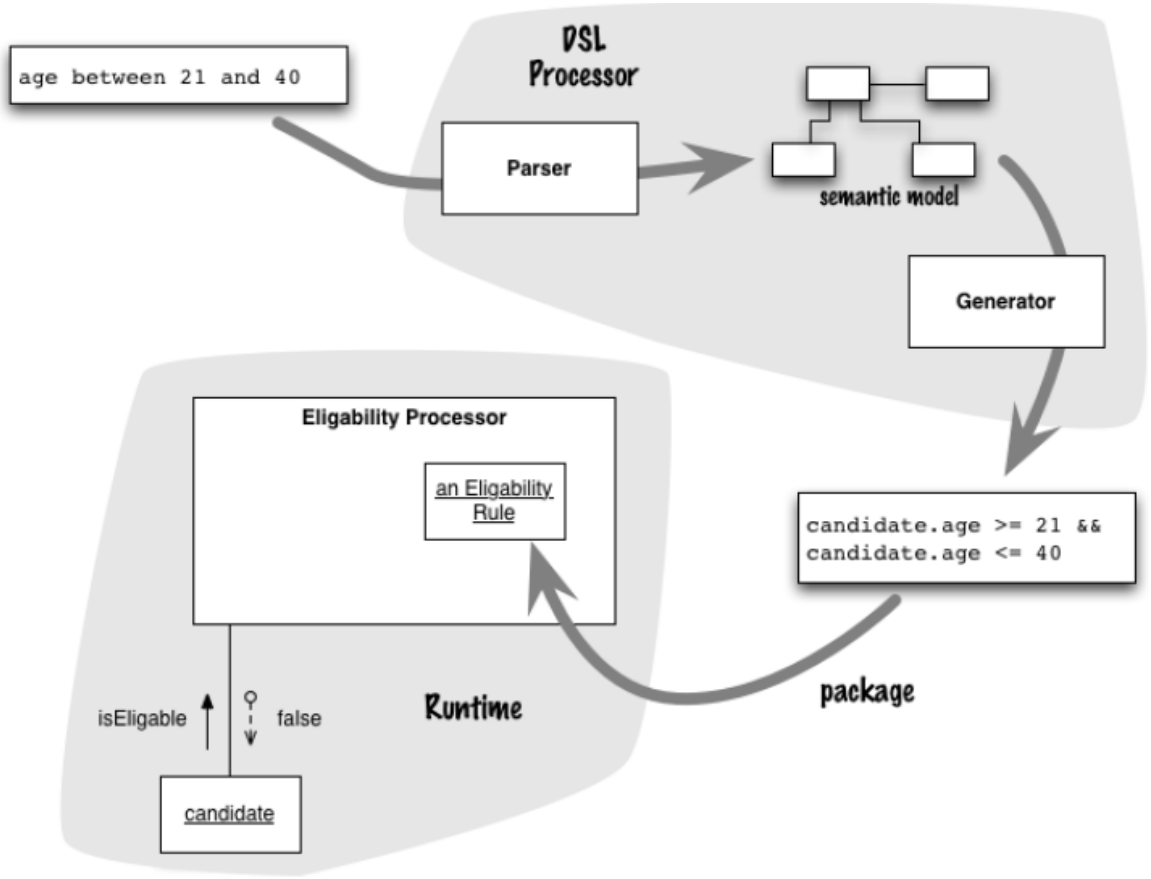
Intérprete vs. Compilador

Ejemplo: LDE para testear si un candidato es elegible para un seguro.



- El intérprete parsea el script LDE y crea el modelo semántico.
- Cuando testea un candidato, ejecuta el modelo semántico contra el candidato para obtener un resultado.

Intérprete vs. **Compilador**



Ejemplo: LDE para testear si un candidato es elegible para un seguro.

- El compilador parsea el script LDE, crea el modelo semántico y genera código en un lenguaje destino, como por ejemplo Java.
- Cuando se testea un candidato, se ejecuta el código generado contra el candidato para obtener un resultado.

Bibliografía

- DSLs in Action. Debasish Ghosh. Manning Publications (2010)

- Domain-Specific Languages. Martin Fowler. Addison-Wesley (2011)