Linguagens de Domínio Específico

Fabio Mascarenhas – 2016.1

http://www.dcc.ufrj.br/~fabiom/dsl

Por que DSLs?

- Melhorar a produtividade dos programadores

```
input =~ /\d{3}-\d{3}-\d{4}/
```

- Facilitar a escrita e manutenção, diminuir a chance de erros
- Código legível para experts no domínio – melhorar a comunicação entre programadores e clientes
- Não necessariamente para experts do domínio *escreverem* código na DSL

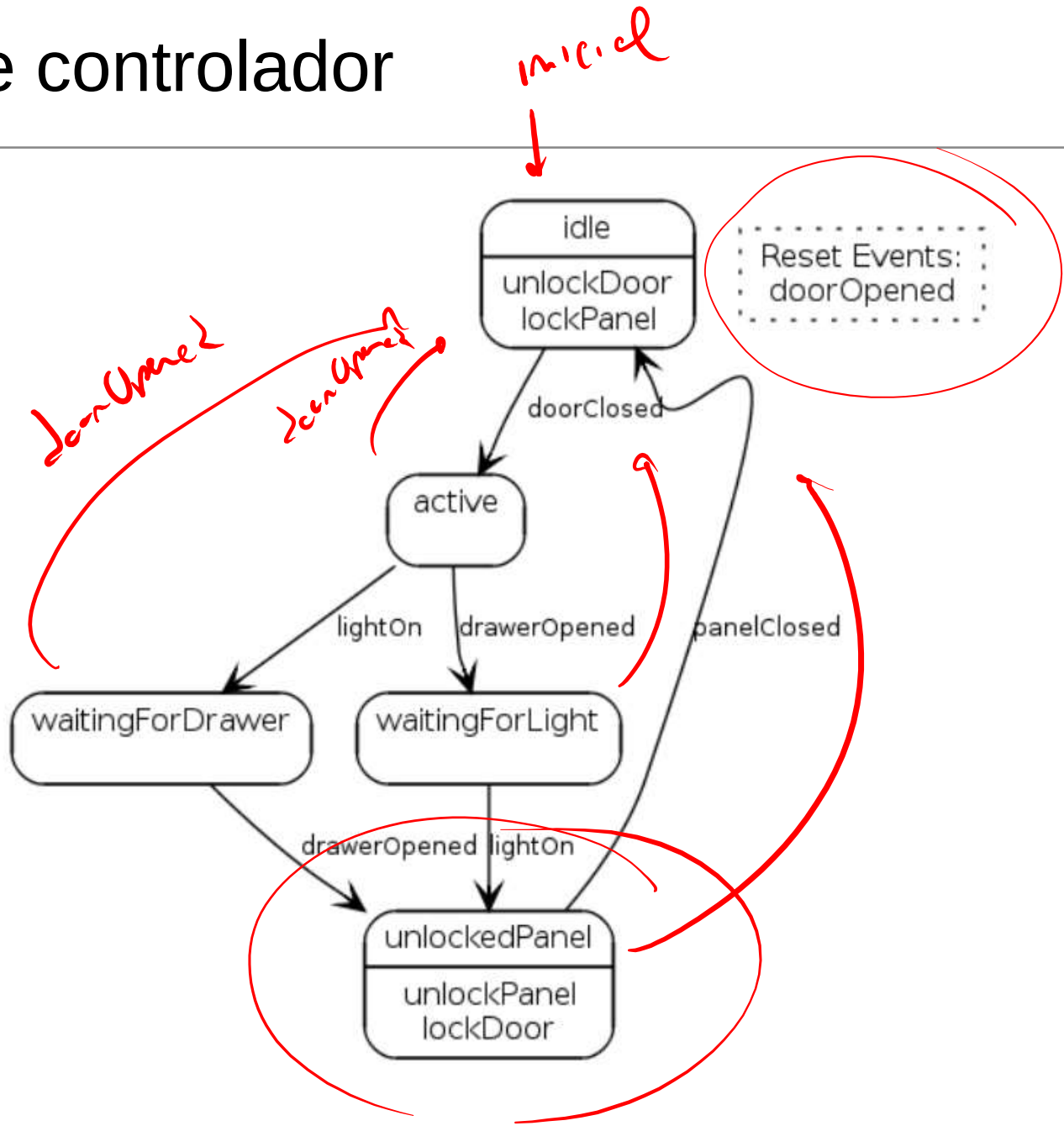
Exemplo – “Segurança Gótica”

- Uma empresa de segurança instala sistemas compostos de diversos sensores e atuadores sem fio
- Os sensores enviam mensagens quando algo acontece, e os atuadores fazem algo quando recebem uma mensagem
- Por exemplo, um sensor pode mandar uma mensagem “D2OP” quando um gaveta é aberta, e um atuador pode destrancar uma porta quando recebe a mensagem “D1UL”
- Um software controlador faz a intermediação, recebendo mensagens de eventos dos sensores e enviando mensagens de comando aos atuadores

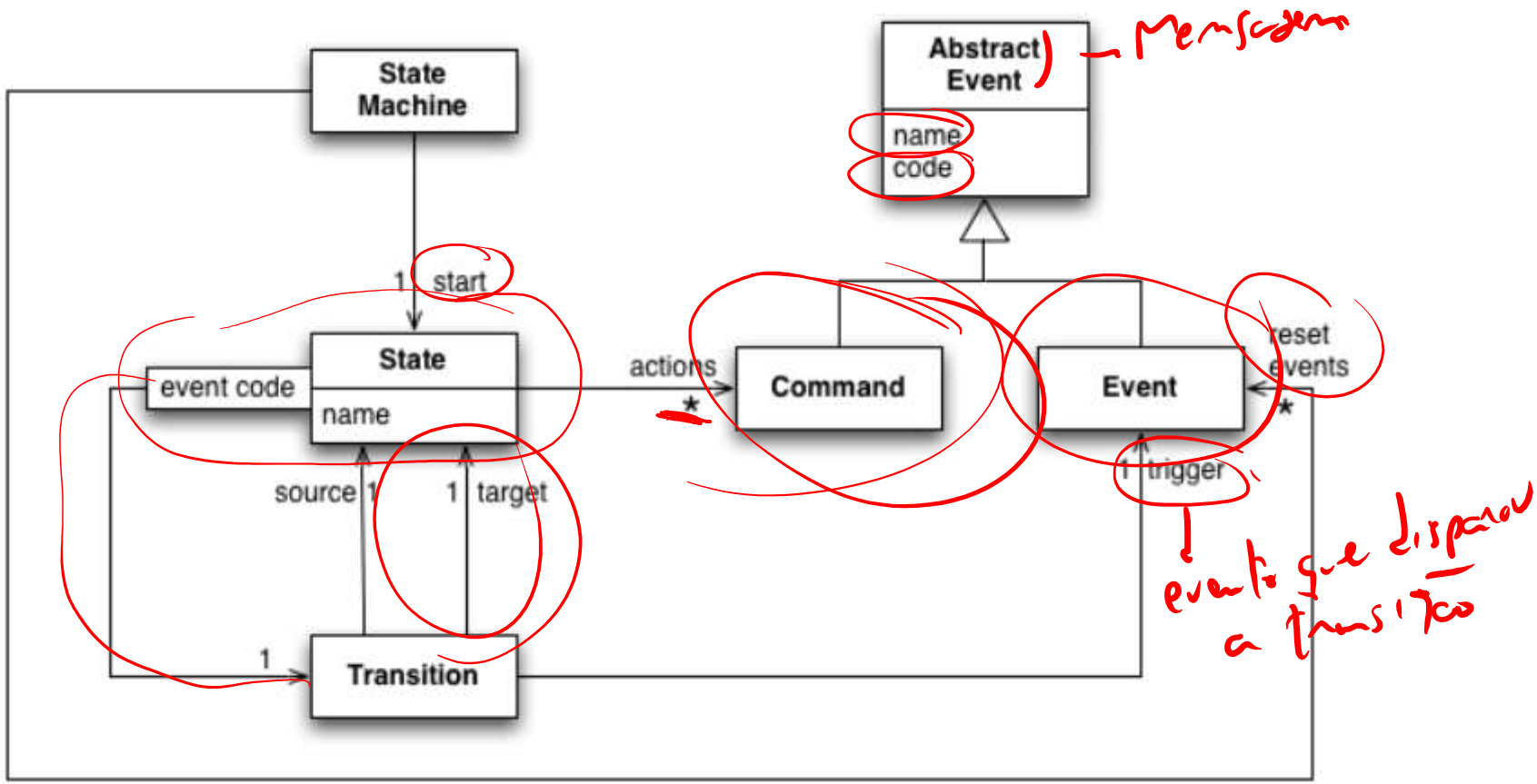
Software controlador

- Cada cliente do sistema de segurança precisa de um software controlador que responde a eventos diferentes e manda comandos diferentes
- Mas a maneira de receber e enviar mensagens é a mesma
- Podemos separar o *mecanismo* do controlador da *política* que ele implementa
- Uma maneira é *modelar* o controlador como uma *máquina de estados*, onde as transições indicam eventos e estados podem ter comandos associados

Exemplo de controlador



Modelo da máquina de estados



Explicando o modelo

- O controlador recebe mensagens de eventos e envia mensagens de comando
- Tanto eventos como comandos são especializações de *mensagens* (Abstract Event, no modelo)
- Cada estado possui um ou mais códigos de evento associados, cada um disparando uma *transição* para outro estado
- Cada estado também opcionalmente possui um conjunto de ações que são executadas (enviadas) na entrada naquele estado
- A máquina de estados contém o estado inicial, e opcionalmente eventos que resetam a máquina para o estado inicial

Mensagens, eventos e comandos

```
class AbstractEvent...
  private String name, code;

  public AbstractEvent(String name, String code) {
    this.name = name;
    this.code = code;
  }
  public String getCode() { return code;}
  public String getName() { return name;}


public class Command extends AbstractEvent


public class Event extends AbstractEvent
```

Estados e transições

```
class State...
  private String name;
  private List<Command> actions = new ArrayList<Command>();
  private Map<String, Transition> transitions = new HashMap<String, Transition>();

class State...
  public void addTransition(Event event, State targetState) {
    assert null != targetState;
    transitions.put(event.getCode(), new Transition(this, event, targetState));
  }

class Transition...
  private final State source, target;
  private final Event trigger;

  public Transition(State source, Event trigger, State target) {
    this.source = source;
    this.target = target;
    this.trigger = trigger;
  }
  public State getSource() {return source;}
  public State getTarget() {return target;}
  public Event getTrigger() {return trigger;}
  public String getEventCode() {return trigger.getCode();}
```

Máquina de estados – fecho

```
class StateMachine...
  private State start;

  public StateMachine(State start) {
    this.start = start;
  }

  public Collection<State> getStates() {
    List<State> result = new ArrayList<State>();
    collectStates(result, start);
    return result;
  }

  private void collectStates(Collection<State> result, State s) {
    if (result.contains(s)) return;
    result.add(s);
    for (State next : s.getAllTargets())
      collectStates(result, next);
  }
```

Máquina de estados - reiniciliazação

```
class State...
  Collection<State> getAllTargets() {
    List<State> result = new ArrayList<State>();
    for (Transition t : transitions.values()) result.add(t.getTarget());
    return result;
  }

class StateMachine...
  private List<Event> resetEvents = new ArrayList<Event>();

  public void addResetEvents(Event... events) {
    for (Event e : events) resetEvents.add(e);
  }

  private void addResetEvent_byAddingTransitions(Event e) {
    for (State s : getStates())
      if (!s.hasTransition(e.getCode())) s.addTransition(e, start);
  }
```

Amarrando as pontas - controlador

```
class Controller...
  private State currentState;
  private StateMachine machine;

  public CommandChannel getCommandChannel() {
    return commandsChannel;
  }

  private CommandChannel commandsChannel;

  public void handle(String eventCode) {
    if (currentState.hasTransition(eventCode))
      transitionTo(currentState.targetState(eventCode));
    else if (machine.isResetEvent(eventCode))
      transitionTo(machine.getStart());
      // ignore unknown events
  }

  private void transitionTo(State target) {
    currentState = target;
    currentState.executeActions(commandsChannel);
  }
```

Fazendo transições

```
class State...
  public boolean hasTransition(String eventCode) {
    return transitions.containsKey(eventCode);
  }
  public State targetState(String eventCode) {
    return transitions.get(eventCode).getTarget();
  }
  public void executeActions(CommandChannel commandsChannel) {
    for (Command c : actions) commandsChannel.send(c.getCode());
  }
```

```
class StateMachine...
  public boolean isResetEvent(String eventCode) {
    return resetEventCodes().contains(eventCode);
  }

  private List<String> resetEventCodes() {
    List<String> result = new ArrayList<String>();
    for (Event e : resetEvents) result.add(e.getCode());
    return result;
  }
```

A máquina de estados do slide 5

```
Event doorClosed = new Event("doorClosed", "D1CL");
Event drawerOpened = new Event("drawerOpened", "D2OP");
Event lightOn = new Event("lightOn", "L1ON");
Event doorOpened = new Event("doorOpened", "D1OP");
Event panelClosed = new Event("panelClosed", "PNCL");

Command unlockPanelCmd = new Command("unlockPanel", "PNUL");
Command lockPanelCmd = new Command("lockPanel", "PNLK");
Command lockDoorCmd = new Command("lockDoor", "D1LK");
Command unlockDoorCmd = new Command("unlockDoor", "D1UL");

State idle = new State("idle");
State activeState = new State("active");
State waitingForLightState = new State("waitingForLight");
State waitingForDrawerState = new State("waitingForDrawer");
State unlockedPanelState = new State("unlockedPanel");

StateMachine machine = new StateMachine(idle);

idle.addTransition(doorClosed, activeState);
idle.addAction(unlockDoorCmd);
idle.addAction(lockPanelCmd);

activeState.addTransition(drawerOpened, waitingForLightState);
activeState.addTransition(lightOn, waitingForDrawerState);

waitingForLightState.addTransition(lightOn, unlockedPanelState);

waitingForDrawerState.addTransition(drawerOpened, unlockedPanelState);

unlockedPanelState.addAction(unlockPanelCmd);
unlockedPanelState.addAction(lockDoorCmd);
unlockedPanelState.addTransition(panelClosed, idle);

machine.addResetEvents(doorOpened);
```

Código de configuração

- O código do slide anterior é um código de *configuração*, ele configura o modelo dado nos slides anteriores para um controlador específico
- Essa separação entre mecanismo e política é comum em diversas bibliotecas e frameworks
- Em frameworks Java é comum que o código de configuração use *XML* ao invés de código Java diretamente

Configuração em XML

```
<stateMachine start = "idle">
  <event name="doorClosed" code="D1CL"/>
  <event name="drawerOpened" code="D2OP"/>
  <event name="lightOn" code="L1ON"/>
  <event name="doorOpened" code="D1OP"/>
  <event name="panelClosed" code="PNCL"/>

  <command name="unlockPanel" code="PNUL"/>
  <command name="lockPanel" code="PNLK"/>
  <command name="lockDoor" code="D1LK"/>
  <command name="unlockDoor" code="D1UL"/>

  <state name="idle">
    <transition event="doorClosed" target="active"/>
    <action command="unlockDoor"/>
    <action command="lockPanel"/>
  </state>

  <state name="active">
    <transition event="drawerOpened" target="waitingForLight"/>
    <transition event="lightOn" target="waitingForDrawer"/>
  </state>

  <state name="waitingForLight">
    <transition event="lightOn" target="unlockedPanel"/>
  </state>

  <state name="waitingForDrawer">
    <transition event="drawerOpened" target="unlockedPanel"/>
  </state>

  <state name="unlockedPanel">
    <action command="unlockPanel"/>
    <action command="lockDoor"/>
    <transition event="panelClosed" target="idle"/>
  </state>

  <resetEvent name = "doorOpened"/>
</stateMachine>
```

Vantagens e desvantagens

- Representar a configuração em XML ao invés de código Java tem a vantagem de não precisar compilar código Java para cada controlador

- A desvantagem é que erros na construção do arquivo de configuração só são executados quando ele é lido e transformado em uma configuração

- Outra vantagem é não precisar da duplicação nos nomes dos estados, onde o nome aparece nos construtores e nos nomes da variáveis que fazem as amarrações

- Outra desvantagem é que o XML precisa ser *interpretado*, mesmo que usemos uma ferramenta pronta para ler o arquivo de configuração XML

Configuração como DSL (externa)

```
events
  doorClosed   D1CL
  drawerOpened  D2OP
  lightOn      L1ON
  doorOpened   D1OP
  panelClosed PNCL
end

resetEvents
  doorOpened
end

commands
  unlockPanel PNUL
  lockPanel   PNLK
  lockDoor    D1LK
  unlockDoor  D1UL
end

state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end

state active
  drawerOpened => waitingForLight
  lightOn     => waitingForDrawer
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDrawer
  drawerOpened => unlockedPanel
end

state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end
```

Vantagens e desvantagens

- O arquivo de configuração do slide anterior também é código, mas em uma linguagem especializada pro modelo que estamos usando
- Em relação a XML, ela tem a vantagem de ser mais fácil de ler e escrever
- Mas tem a desvantagem de precisar de código especializado para leitura, e não apenas a sua interpretação (embora possamos fazer os dois com o mesmo código)
- A DSL do exemplo anterior é bem simples: em particular, ela não tem nenhuma construção que reconhecemos de linguagens de propósito geral

Controlador em Ruby

```
event :doorClosed, "D1CL"
event :drawerOpened,  "D2OP"
event :lightOn, "L1ON"
event :doorOpened,  "D1OP"
event :panelClosed, "PNCL"

command  :unlockPanel, "PNUL"
command  :lockPanel,   "PNLK"
command  :lockDoor,    "D1LK"
command  :unlockDoor,  "D1UL"

resetEvents :doorOpened

state :idle do
  actions :unlockDoor, :lockPanel
  transitions :doorClosed => :active
end

state :active do
  transitions :drawerOpened => :waitingForLight,
              :lightOn => :waitingForDrawer
end

state :waitingForLight do
  transitions :lightOn => :unlockedPanel
end

state :waitingForDrawer do
  transitions :drawerOpened => :unlockedPanel
end

state :unlockedPanel do
  actions :unlockPanel, :lockDoor
  transitions :panelClosed => :idle
end
```

DSLs internas

- O controlador do slide anterior se parece com o do slide 18, mas também é código executável em uma linguagem de propósito geral (no caso, Ruby)
- Existem linguagens de propósito geral com sintaxe e semântica flexível o suficiente para *embutir* DSLs simples na própria linguagem
- Dizemos que essas DSLs são *internas*, *embarcadas* ou *embutidas*
- Vamos usar *internas* para não confundir com o uso de linguagens de propósito geral embutidas em aplicações como *linguagem de script*

Controlador Java, take 2

```
public class BasicStateMachine extends StateMachineBuilder {

  Events doorClosed, drawerOpened, lightOn, panelClosed;
  Commands unlockPanel, lockPanel, lockDoor, unlockDoor;
  States idle, active, waitingForLight, waitingForDrawer, unlockedPanel;
  ResetEvents doorOpened;

  protected void defineStateMachine() {
    doorClosed. code("D1CL");
    drawerOpened. code("D2OP");
    lightOn.    code("L1ON");
    panelClosed.code("PNCL");

    doorOpened. code("D1OP");

    unlockPanel.code("PNUL");
    lockPanel.  code("PNLK");
    lockDoor.   code("D1LK");
    unlockDoor. code("D1UL");

    idle
      .actions(unlockDoor, lockPanel)
      .transition(doorClosed).to(active)
      ;

    active
      .transition(drawerOpened).to(waitingForLight)
      .transition(lightOn).   to(waitingForDrawer)
      ;

    waitingForLight
      .transition(lightOn).to(unlockedPanel)
      ;

    waitingForDrawer
      .transition(drawerOpened).to(unlockedPanel)
      ;

    unlockedPanel
      .actions(unlockPanel, lockDoor)
      .transition(panelClosed).to(idle)
      ;
  }
}
```

DSLs internas em Java

- Como vemos no slide anterior, mesmo em Java podemos criar uma DSL interna, embora ela não fique tão elegante

- No contexto de Java e linguagens similares, uma DSL interna no estilo do slide 22 também é chamada de *interface fluente* ou *API fluente*

- Para diferenciar, vamos chamar uma API no estilo do slide 14 de *API comando-consulta*

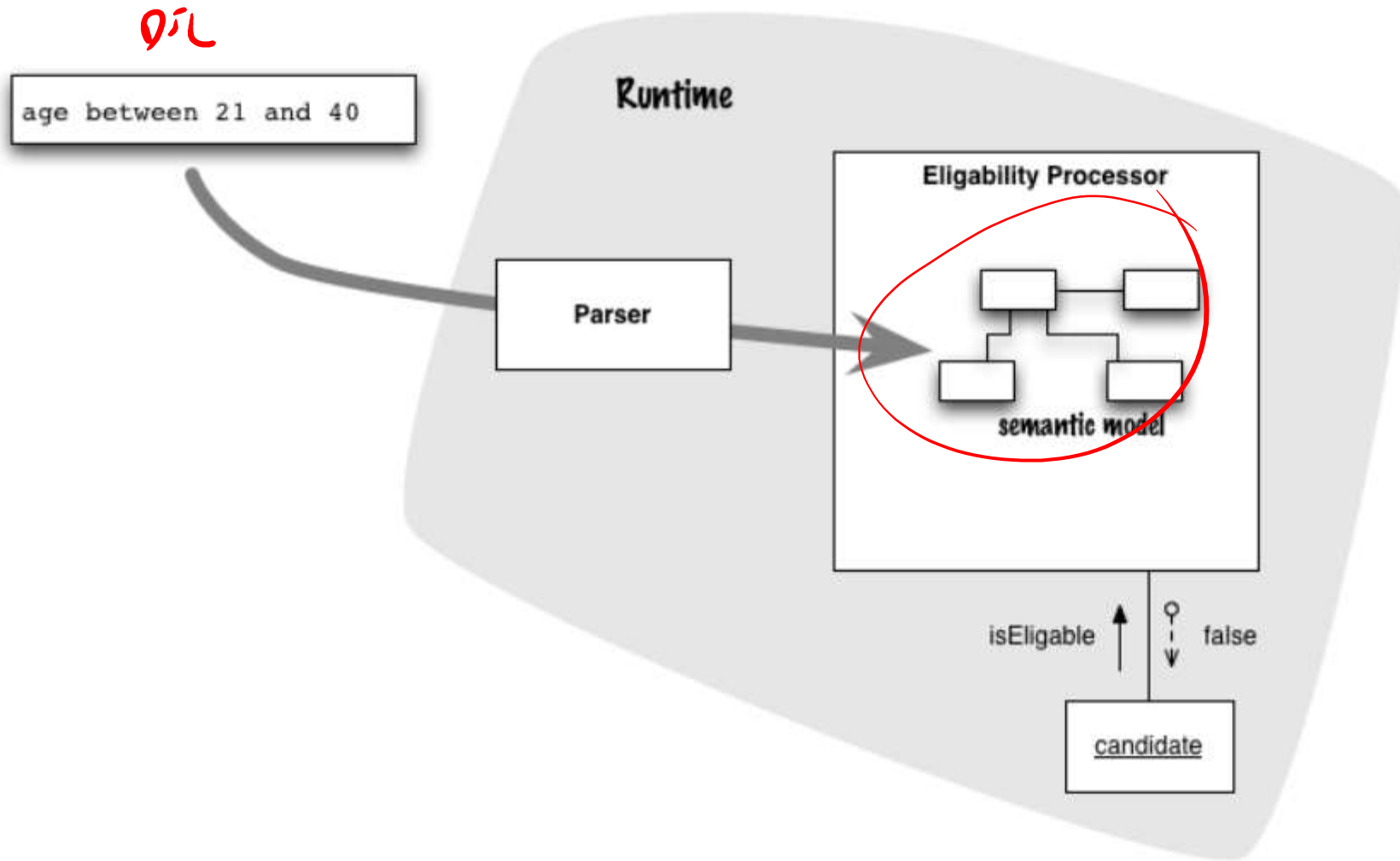
Modelo semântico

- O modelo subjacente que uma DSL instancia é o seu *modelo semântico*
- No caso das DSLs de máquina de estados, é o modelo do slide 6 e 8-13
- Em uma linguagem de propósito geral, o modelo semântico normalmente é uma *sintaxe abstrata* ou outra *representação intermediária*, como uma *máquina virtual*
- DSLs costumam ter modelos semânticos alternativos, como máquinas de estados, sistemas de regras de produção, redes de dependências...

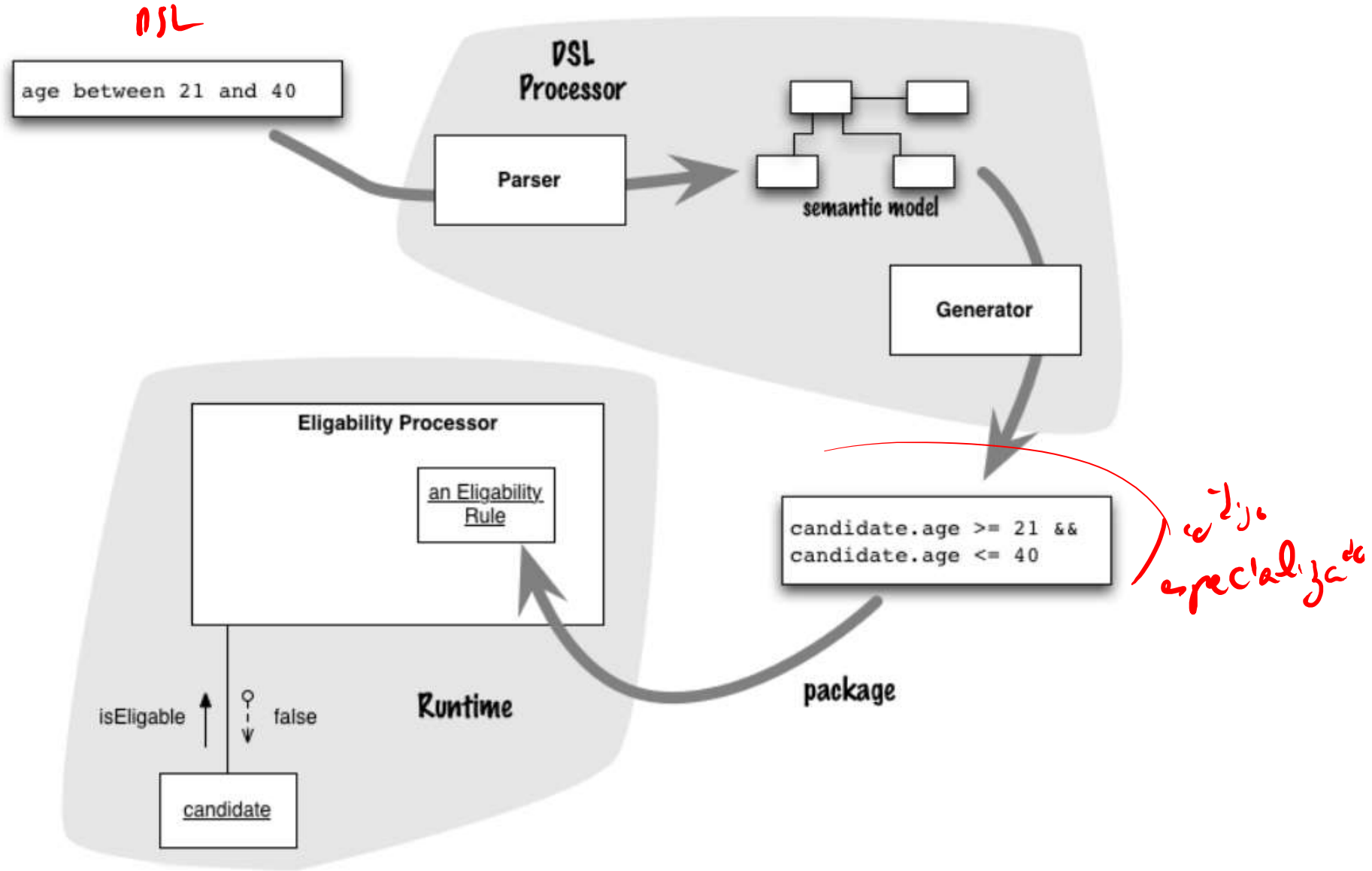
Interpretação vs geração de código

- Nosso exemplo usa as DSLs para preencher o modelo semântico, e depois executa o modelo
- No mundo de linguagens de programação, nosso exemplo usa um *intrepretador*
- Uma alternativa à interpretação é a *compilação*; no contexto de DSLs vamos usar o termo *geração de código*
- Nessa abordagem o modelo semântico é usado para produzir código especializado para uma instância específica do modelo

Interpretação



Geração de código



Geração de código

- Uma vantagem da geração de código é poder usar uma linguagem de implementação diferente da linguagem de execução de uma DSL
- O gerador de código pode gerar código em qualquer linguagem
- Outra vantagem, mas que não é tão relevante no contexto de DSLs, é que geração de código pode ser mais eficiente que interpretação
- A principal desvantagem é a complexidade; um interpretador quase sempre é mais simples que um gerador de código, especialmente quando se leva tratamento de erros em consideração