Engenharia de Software

Implementação

Luís Morgado

Instituto Superior de Engenharia de Lisboa
Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

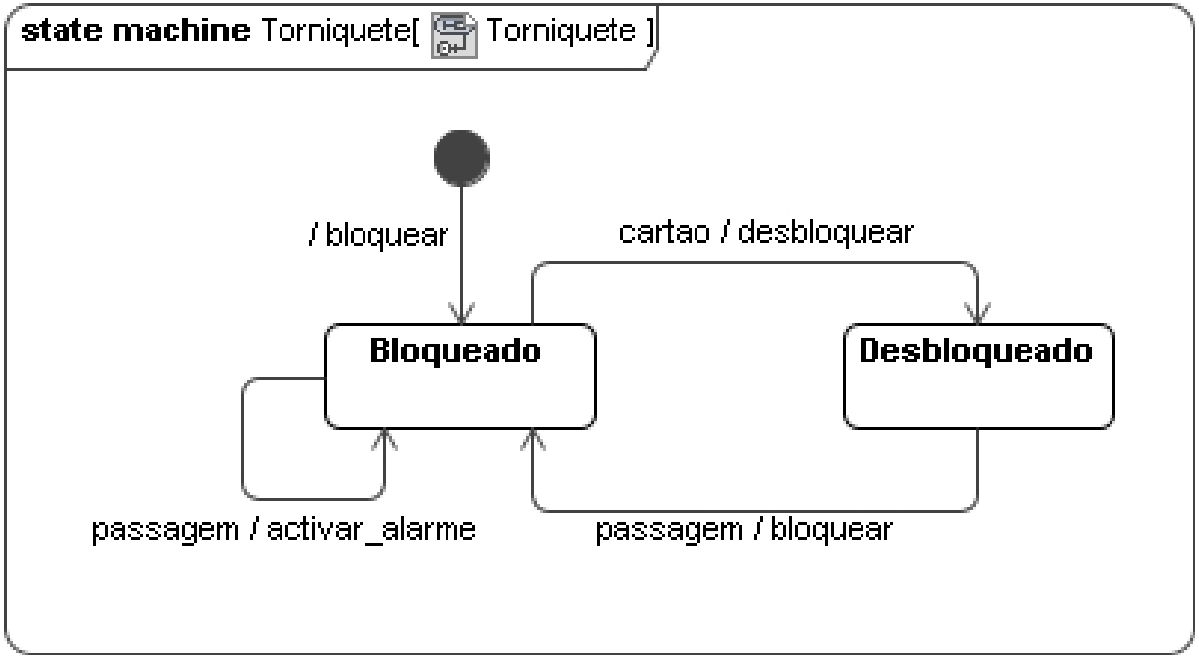
Modelo de Dinâmica

- **Diferentes formas de implementação**
  - **Procedimental**
  - **Padrão de arquitectura “*State*”**
  - **Máquinas de estados finitos**
    - Tabela de transição de estado
    - Tabela de activação de estado
  - **Máquinas de estados finitos hierárquicas**

Implementação de Dinâmica

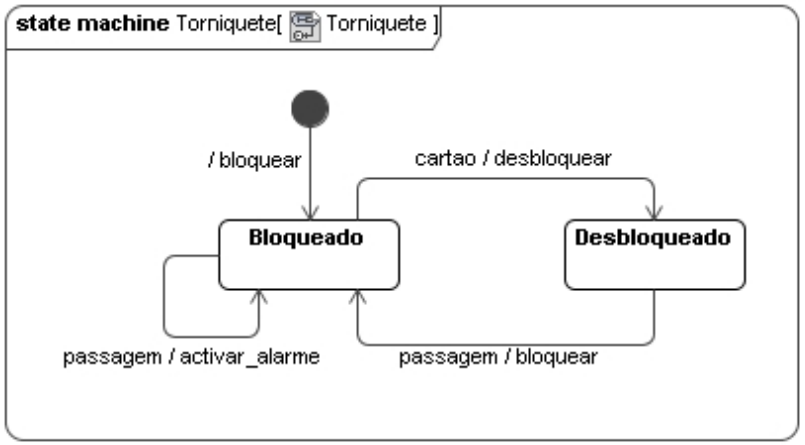
Diagramas de Transição de Estado

Exemplo: Torniquete



Torniquete: Implementação

Implementação
Procedimental da Dinâmica




```
public enum EstadoTorn
{
    BLOQUEADO,
    DESBLOQUEADO
}
```

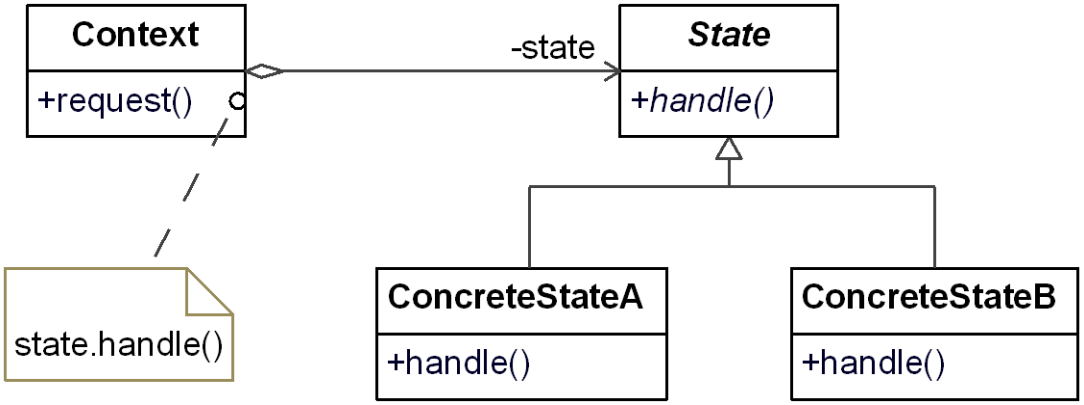
```
public void iniciar() {
    actuador.bloquear();
    setEstado(EstadoTorn.BLOQUEADO);
}
```

```
public void processar(EventoTorn evento) {
    switch(estado) {
    case BLOQUEADO:
        switch(evento) {
        case CARTAO:
            actuador.desbloquear();
            setEstado(EstadoTorn.DESBLOQUEADO);
            break;
        case PASSAGEM:
            actuador.activarAlarme();
            break;
        default:
            break;
        }
        break;

    case DESBLOQUEADO:
        switch(evento) {
        case PASSAGEM:
            actuador.bloquear();
            setEstado(EstadoTorn.BLOQUEADO);
            break;
        default:
            break;
        }
    }
}
```

Máquinas de Estados Finitos

- **Implementação Procedimental**
  - Viável para dinâmicas simples
    - Número reduzido de estados e de eventos
  - Inadequado para dinâmicas complexas
  - **Problemas**
    - Procedimentos longos
    - Baixa modularidade
    - Baixa coesão
    - Dificuldade de compreensão e comunicação
    - Dificuldade de evolução

Padrão de Arquitectura "*State*"



Context

- Defines the interface of interest to clients
- Maintains an instance of a *ConcreteState* subclass that defines the current state

State

- Defines an interface for encapsulating the behavior associated with a particular state of the *Context*

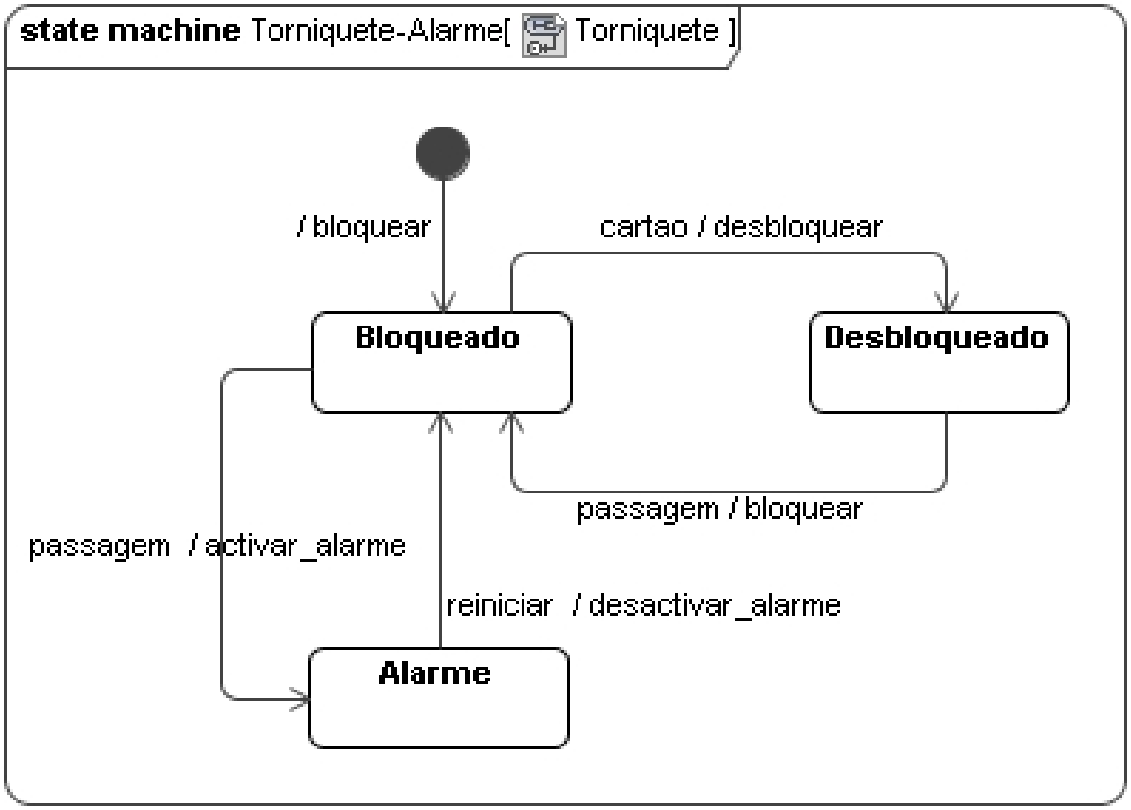
ConcreteState subclasses

- Each subclass implements a behavior associated with a state oft he *Context*

[*Design Patterns: Elements of Reusable Object-Oriented Software*. Gamma *et al*., 1994]

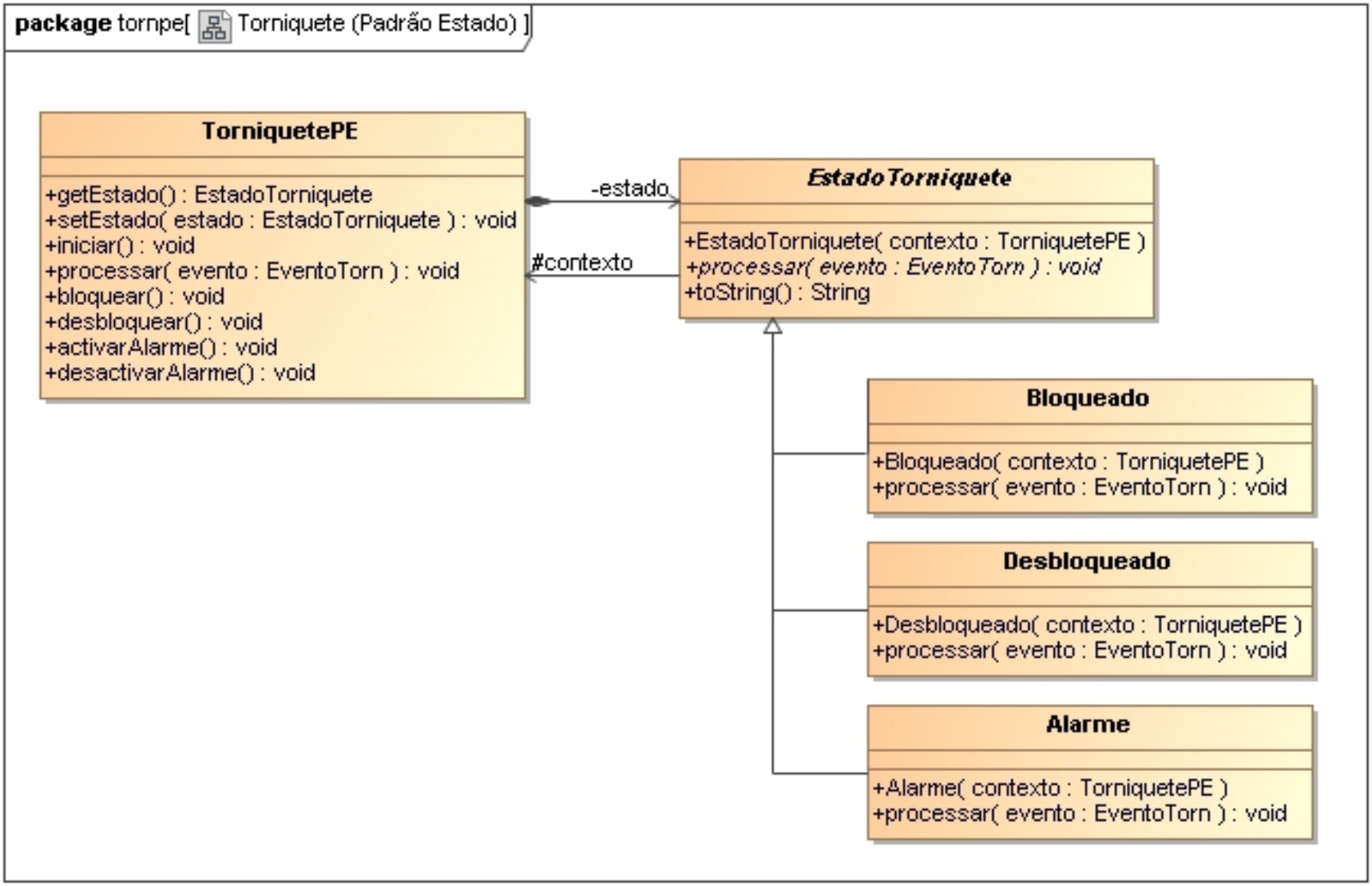
Modelo de Dinâmica

Exemplo: Torniquete



Torniquete: Arquitectura

Torniquete com padrão *Estado*



Torniquete: Implementação

Estado *Bloqueado*

```
public class Bloqueado extends EstadoTorniquete
{
    /**
     * Construtor
     * @param contexto Contexto de execução
     */
    public Bloqueado(TorniquetePE contexto) {
        super(contexto);
    }

    @Override
    public void processar(EventoTorn evento) {
        switch(evento) {
        case CARTAO:
            contexto.desbloquear();
            contexto.setEstado(new Desbloqueado(contexto));
            break;
        case PASSAGEM:
            contexto.activarAlarme();
            contexto.setEstado(new Alarme(contexto));
            break;
        default:
            break;
        }
    }
}
```

Torniquete: Implementação

Estado *Bloqueado*

```
public class Bloqueado extends EstadoTorniquete
{
    /**
     * Construtor
     * @param contexto
     */
    public Bloqueado(T
        super(contexto
    }

    @Override
    public void processar(EventoTorn evento) {
        switch(evento) {
        case CARTAO:
            contexto.desbloquear();
            contexto.setEstado(new Desbloqueado(contexto));
            break;
        case PASSAGEM:
            contexto.activarAlarme();
            contexto.setEstado(new Alarme(contexto));
            break;
        default:
            break;
        }
    }
}
```

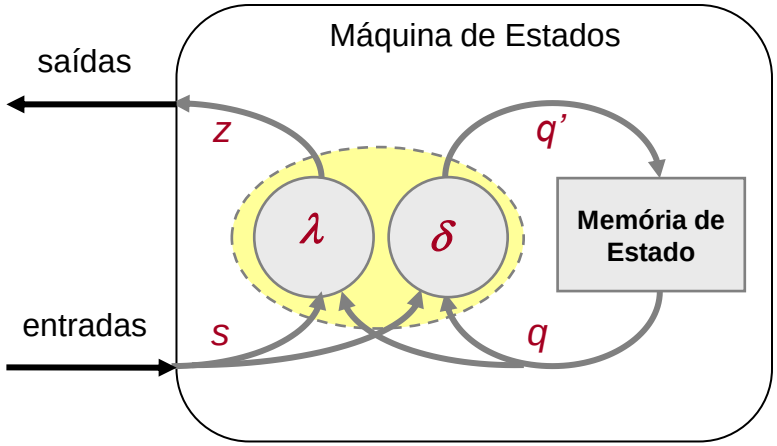
```
public void processar(EventoTorn evento) {
    switch(estado) {
    case BLOQUEADO:
        switch(evento) {
        case CARTAO:
            actuador.desbloquear();
            setEstado(EstadoTorniquete.DESBLOQUEADO);
            break;
```

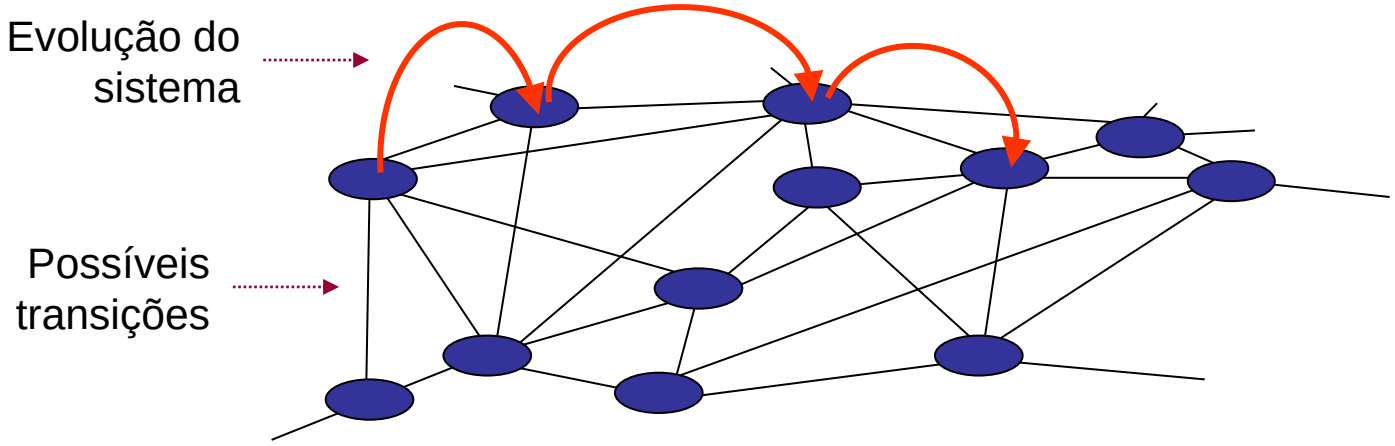
Implementação Procedimental

Padrão de Arquitectura "*State*"

- **Vantagens**
  - **Modularização do comportamento** associado a cada estado
    - Comportamento específico de cada estado definido na classe respectiva
    - Evita procedimentos longos de processamento de eventos
  - Facilidade de definição de novos estados e transições de estado
- **Problemas**
  - **Proliferação de classes** de representação de estado torna difícil a compreensão e revisão da dinâmica comportamental
    - **Complexidade**
  - Interface de **processamento de eventos pouco flexível** e interdependente entre contexto e estado
  - **Acoplamento bidireccional** entre contexto e estado

Máquinas de Estados

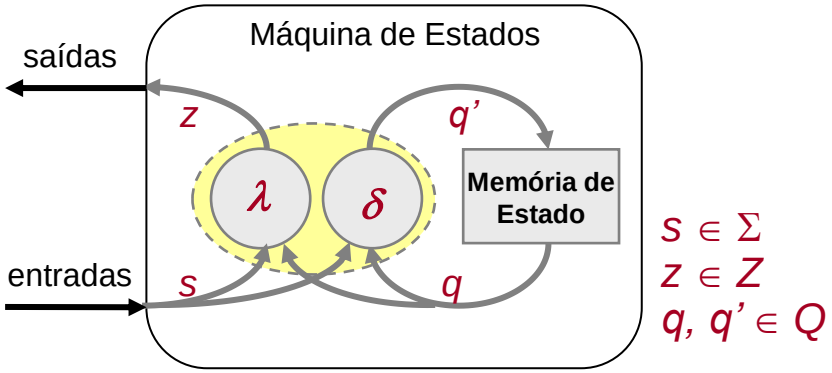




Espaço de estados

Modelo Formal de Máquina de Estados

- Entradas e saídas abstraídas em termos dos conjuntos de símbolos que nelas podem ocorrer:
  - Esses conjuntos de símbolos são designados *alfabetos*.
  - Consideremos um ***alfabeto de entrada*** Σ e um ***alfabeto de saída*** Z.
- Estado interno do sistema descrito em termos de um **conjunto de estados** possíveis:
  - Q
- Função de transformação do sistema descrita com base em duas funções distintas δ e λ:
  - **Função de transição de estado**:
    - $\delta : Q \times \Sigma \rightarrow Q$
  - **Função de saída**:
    - $\lambda : Q \times \Sigma \rightarrow Z$

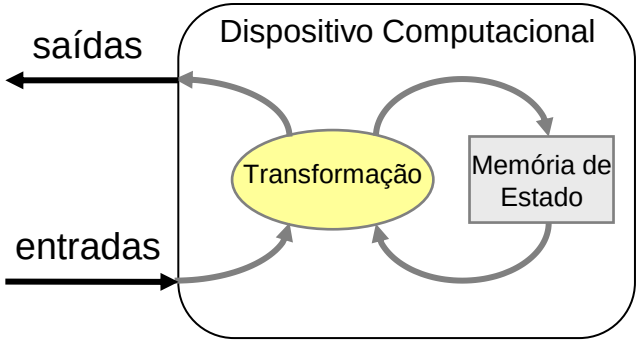


Modelo Formal de Computação

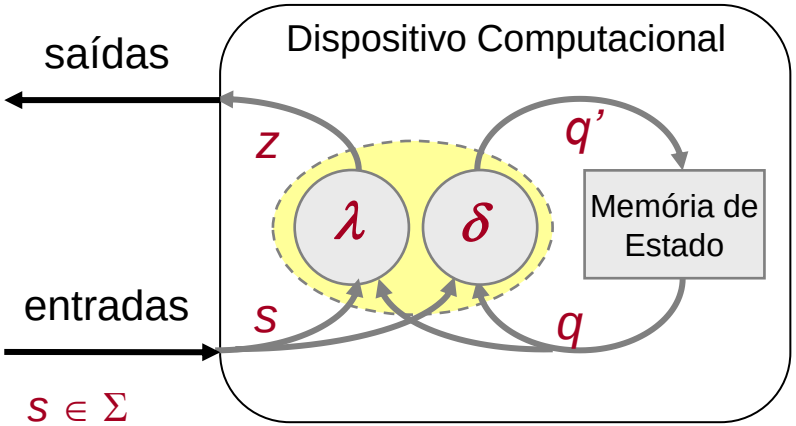
Um modelo formal de computação pode ser descrito como um quíntuplo $(Q, \Sigma, Z, \delta, \lambda)$, onde:

- Q é o **conjunto de estados** que caracterizam o sistema.
- Σ é o conjunto de símbolos de entrada (o **alfabeto de entrada**).
- Z é o conjunto de símbolos de saída (o ***alfabeto de saída***).
- $\delta : Q \times \Sigma \rightarrow Q$ é a **função de transição de estado**.
- $\lambda : Q \times \Sigma \rightarrow Z$ é a **função de saída**.

Tipos de Máquinas de Estados



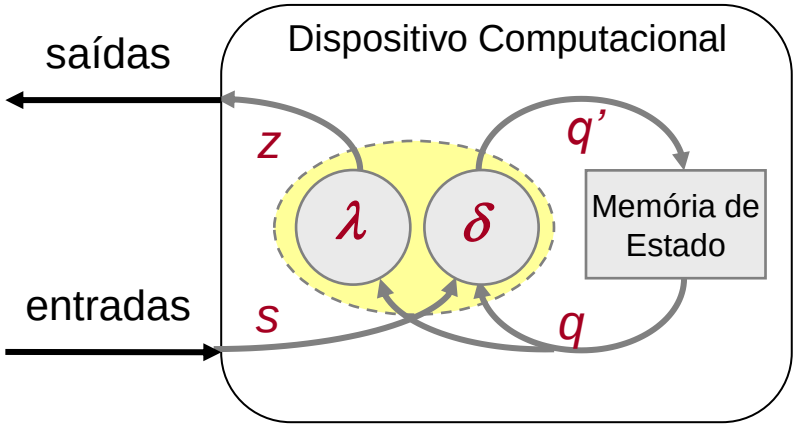
$\delta: Q \times \Sigma \rightarrow Q$
$\lambda: Q \times \Sigma \rightarrow Z$



$s \in \Sigma$
$z \in Z$
$q, q' \in Q$

Máquinas de Mealy

$\delta: Q \times \Sigma \rightarrow Q$
$\lambda: Q \rightarrow Z$



Máquinas de Moore

Exemplo

Sistema de Regulação Automática de Temperatura

O sistema de controlo recebe do exterior uma entrada que pode assumir valores com as seguintes representações simbólicas:

- T_REG : indica que a temperatura está dentro dos limites definidos;
- T_BAIXA : indica que a temperatura está abaixo do limite mínimo;
- T_ALTA : indica que a temperatura está acima do limite máximo.

Por sua vez, o sistema produz uma saída para controlo dos mecanismos de aquecimento e de arrefecimento, a qual pode assumir valores com as seguintes representações simbólicas:

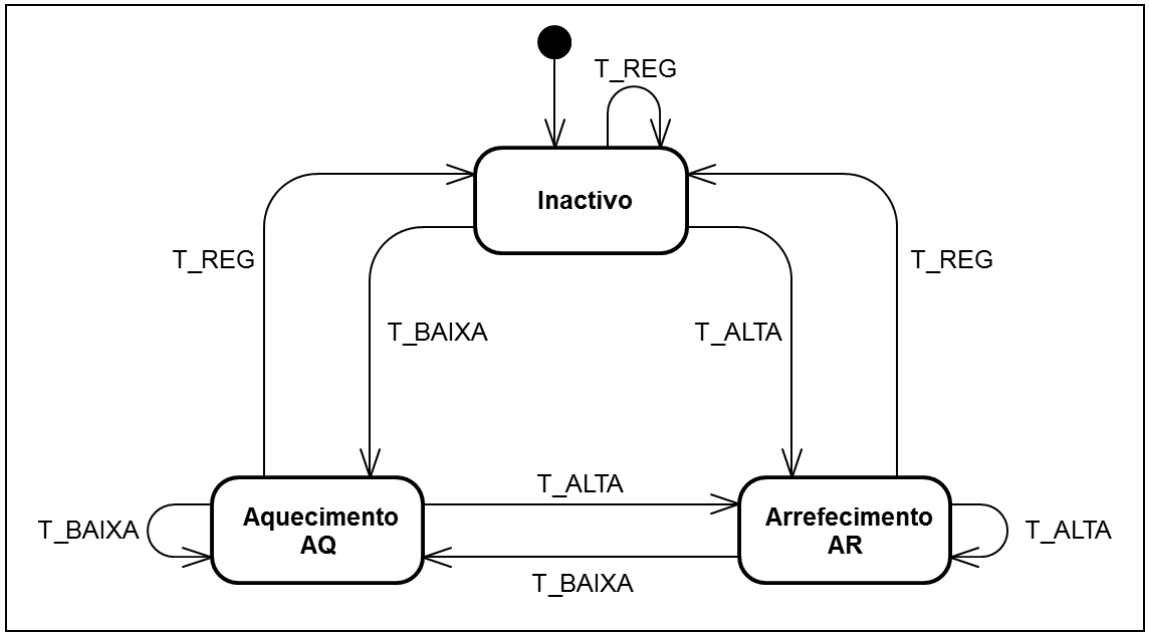
- AQ : sistema de aquecimento é activado;
- AR : sistema de arrefecimento é activado.

Na ausência dos valores AQ ou AR à saída do sistema de controlo, os mecanismos de aquecimento e de arrefecimento respectivos mantêm-se inactivos.
O sistema de controlo é caracterizado por três estados:

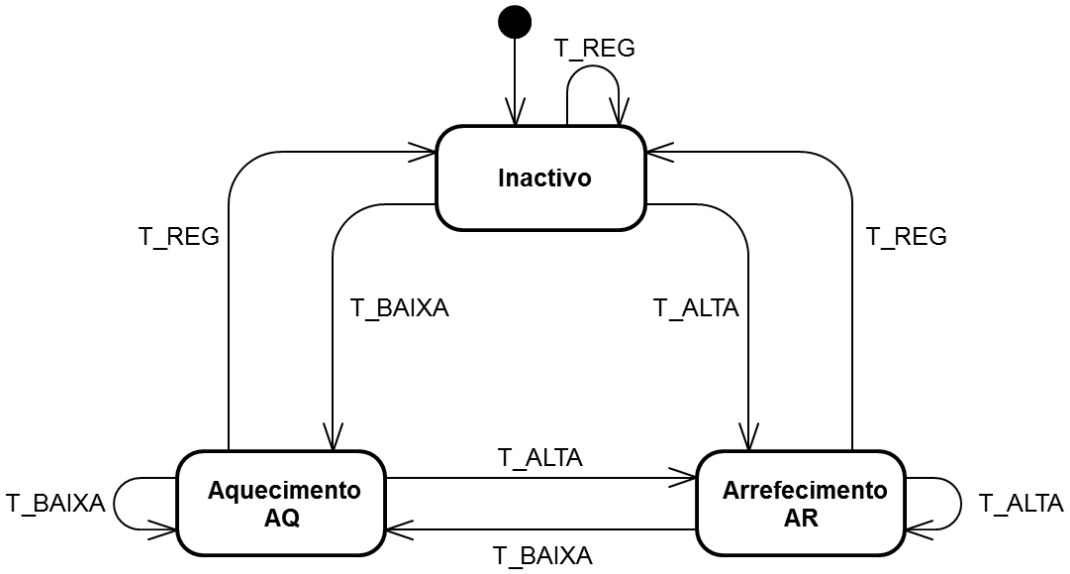
- Inactivo : os sistemas de aquecimento e de arrefecimento estão inactivos;
- Aquecimento : apenas o sistema de aquecimento está activo;
- Arrefecimento : apenas o sistema de arrefecimento está activo.

Exemplo

- Conjunto de símbolos de entrada (o alfabeto de entrada):
  - Σ = { T_REG, T_BAIXA, T_ALTA }
- Conjunto de símbolos de saída (o alfabeto de saída):
  - ***Z*** = { AQ, AR }
- Conjunto de estados que caracterizam o sistema de controlo:
  - ***Q*** = { Inactivo, Aquecimento, Arrefecimento }



Implementação



Função de transição de estado

$\delta : Q \times \Sigma \rightarrow Q$

Q \ Σ	T_REG	T_BAIXA	T_ALTA
$q_{inactivo}$	$q_{inactivo}$	$q_{aquecimento}$	$q_{arrefecimento}$
$q_{aquecimento}$	$q_{inactivo}$	$q_{aquecimento}$	$q_{arrefecimento}$
$q_{arrefecimento}$	$q_{inactivo}$	$q_{aquecimento}$	$q_{arrefecimento}$

Tabela de transição de estado

Função de saída

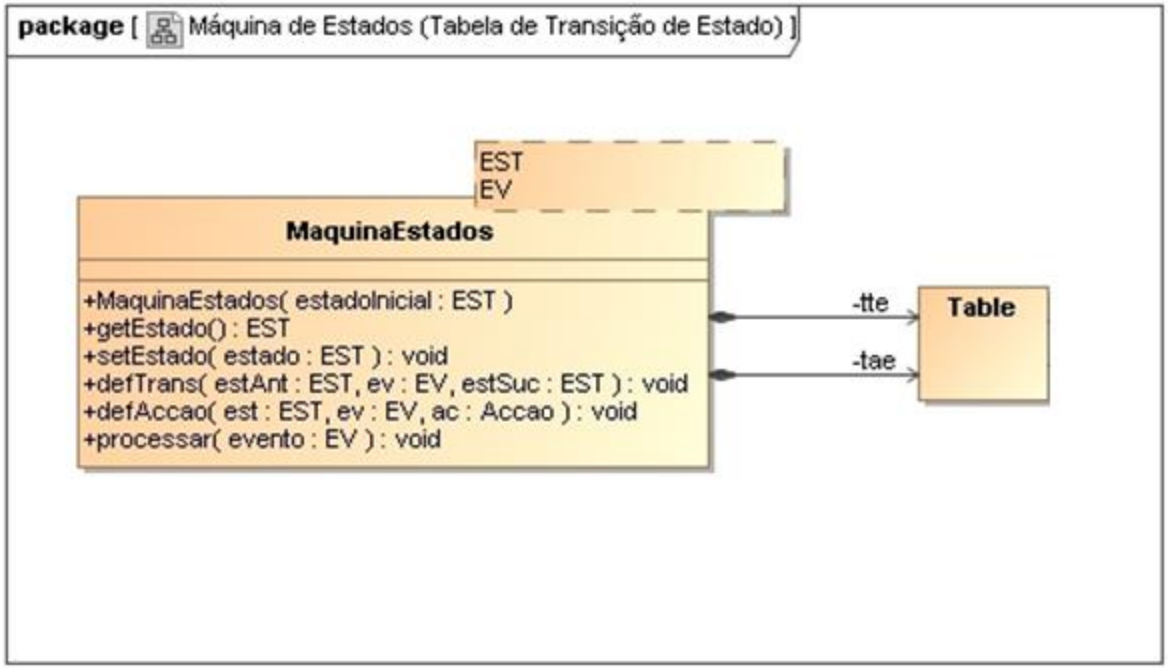
$\lambda : Q \rightarrow Z$

Q	Z
$q_{inactivo}$	—
$q_{aquecimento}$	AQ
$q_{arrefecimento}$	AR

Tabela de acção de estado

Máquina de Estados

Implementação com tabela de transição de estado



TTE – Tabela de Transição de Estado

TAE – Tabela de Acção de Estado

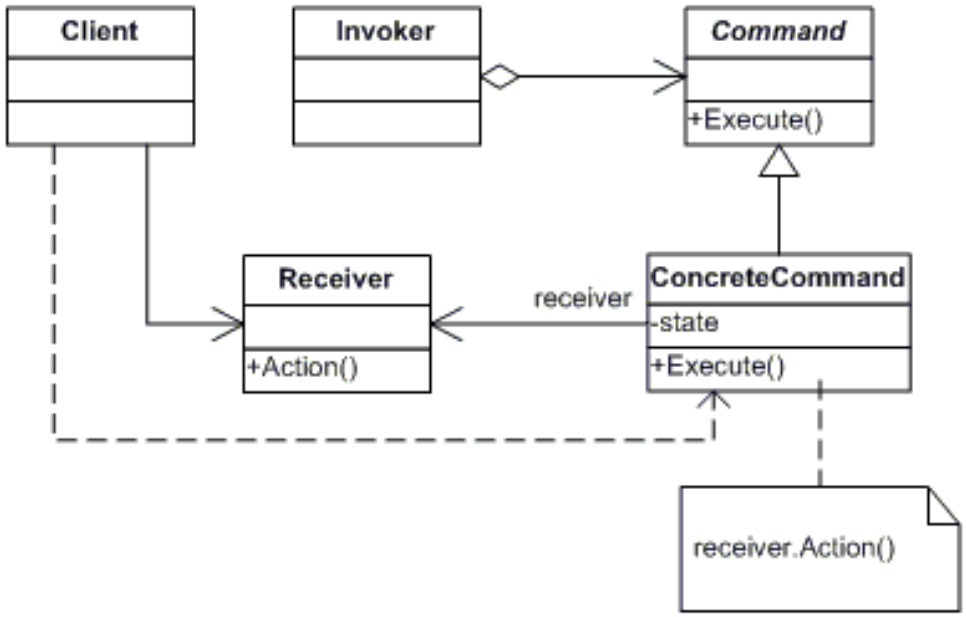
Implementação com Tabela de Transição de Estado

```
public class MaquinaEstados<EST, EV>
{
    /** Tabela de transição de estado (função delta) */
    private Table<EST, EV, EST> tte;

    /** Tabela de acção de estado (função lambda) */
    private Table<EST, EV, Accao> tae;
```

```
public EST processar(EV evento)
{
    // Executar acção de transição
    Accao accao = tae.get(estado, evento);
    if(accao != null)
        accao.executar();

    // Transitar de estado
    EST novoEstado = tte.get(estado, evento);
    if(novoEstado != null) {
        estado = novoEstado;
    }

    return novoEstado;
}
```

```
public Torniquete() {
    maqEst = new MaquinaEstados<EstadoTorn, EventoTorn>(EstadoTorn.BLOQUEADO);

    // Definição de transições
    maqEst.defTrans(EstadoTorn.BLOQUEADO, EventoTorn.CARTAO, EstadoTorn.DESBLOQUEADO);
    maqEst.defTrans(EstadoTorn.BLOQUEADO, EventoTorn.PASSAGEM, EstadoTorn.BLOQUEADO);
    maqEst.defTrans(EstadoTorn.DESBLOQUEADO, EventoTorn.CARTAO, EstadoTorn.DESBLOQUEADO);
    maqEst.defTrans(EstadoTorn.DESBLOQUEADO, EventoTorn.PASSAGEM, EstadoTorn.BLOQUEADO);

    // Definição de acções de transição de estado
    maqEst.defAccao(EstadoTorn.BLOQUEADO, EventoTorn.CARTAO, new Desbloquear(actuador));
    maqEst.defAccao(EstadoTorn.BLOQUEADO, EventoTorn.PASSAGEM, new ActivarAlarme(actuador));
    maqEst.defAccao(EstadoTorn.DESBLOQUEADO, EventoTorn.PASSAGEM, new Bloquear(actuador));
}
```

Padrão de Arquitectura "*Command*"



Command
- Declares an interface for executing an operation

ConcreteCommand
- Defines a binding between a *Receiver* object and an action
- Implements *Execute* by invoking the corresponding operation(s) on *Receiver*

Client
- Creates a *ConcreteCommand* object and sets its receiver

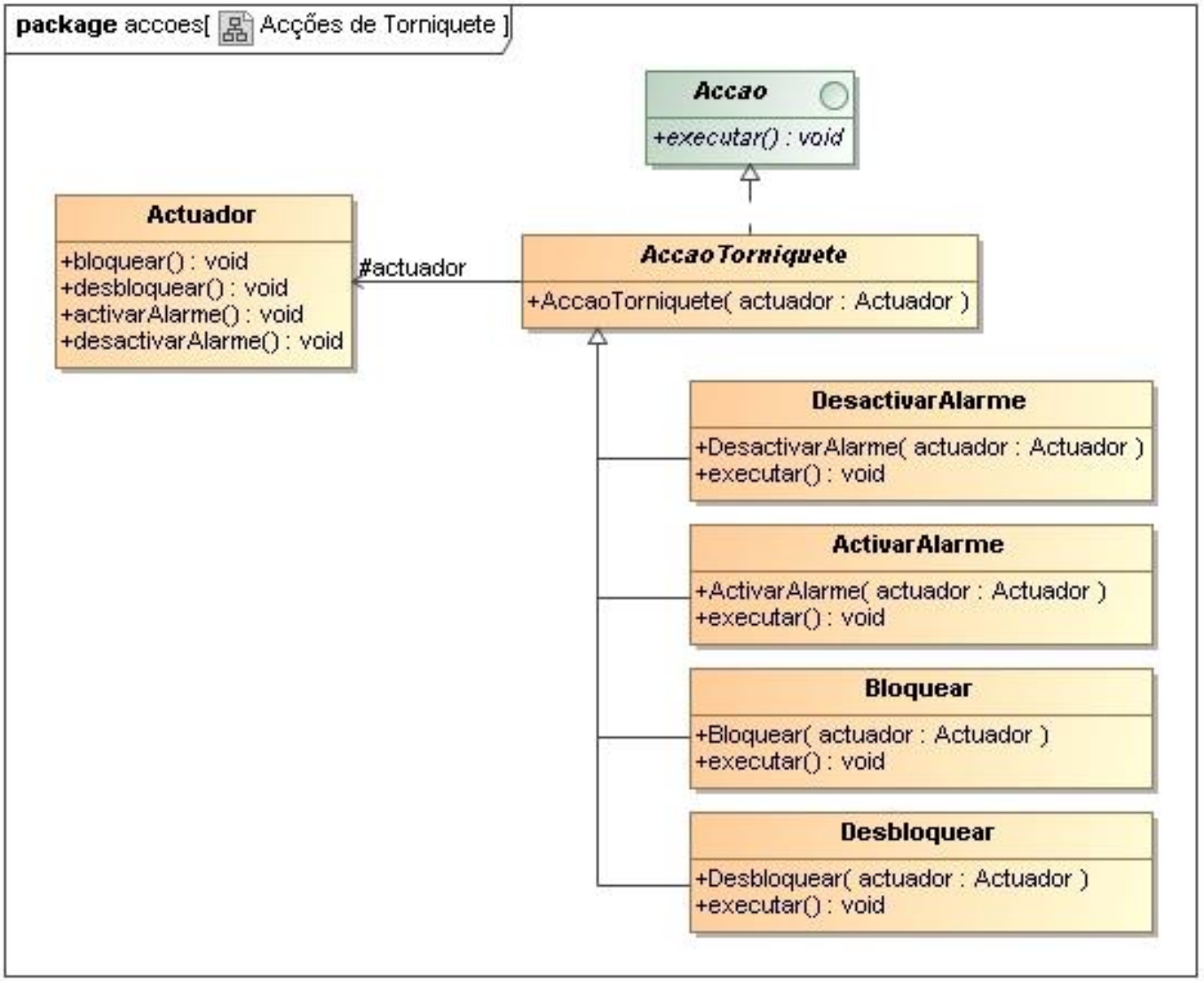
Invoker
- Asks the command to carry out the request

Receiver
- Knows how to perform the operations associated with carrying out a request

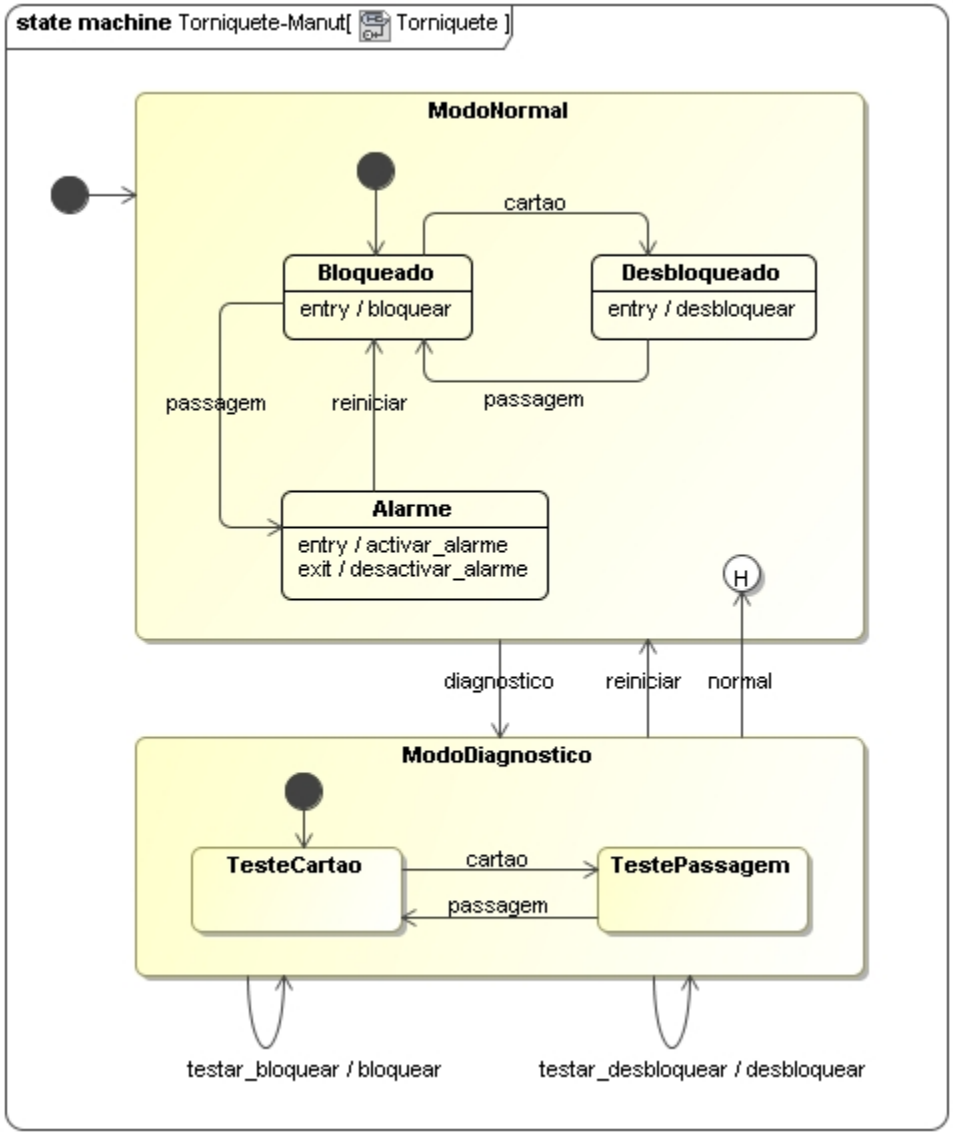
[Gamma *et al*., 1995]

Torniquete: Acções

Implementação com Padrão *Comando*

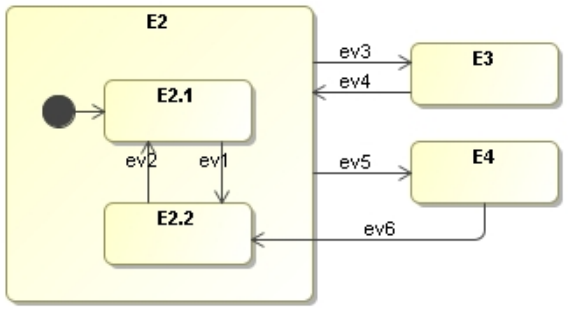


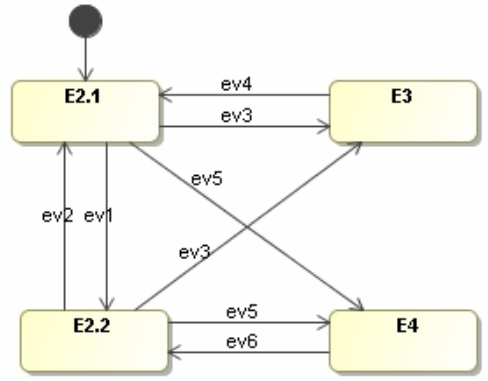
Torniquete com Manutenção



Máquinas de Estados Hierárquicas

- **Implementação**
  - **Máquina de estados hierárquica**
  - **Padrão de arquitectura "*State*"**
    - Sub-máquinas de estado
    - Manutenção de histórico
  - **Planificação**
    - Decomposição de estados compostos
    - Implementação com tabela de transição de estados
    - Manutenção de histórico (superficial/profundo)



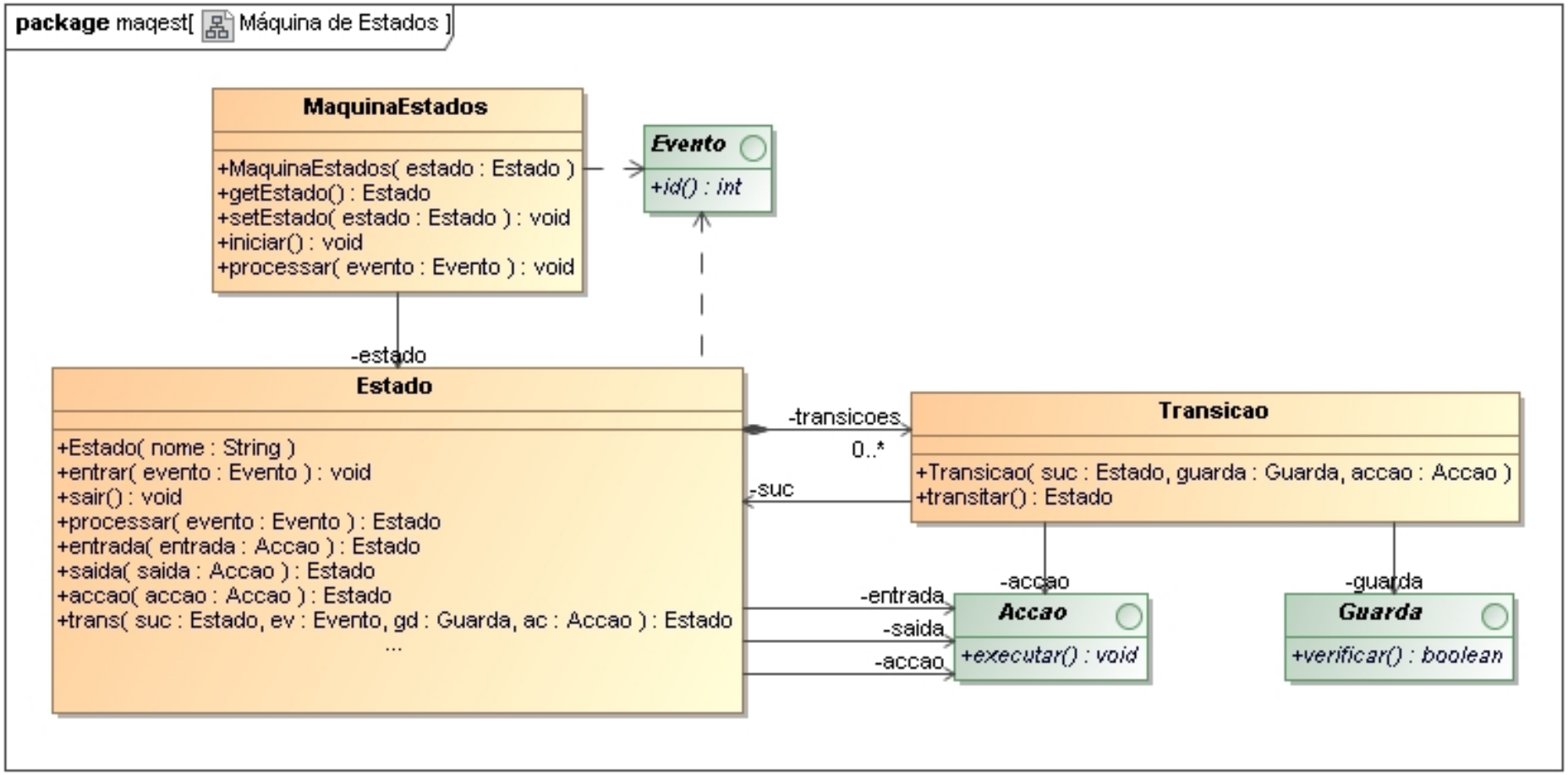


Máquina de Estados Hierárquica

- **Conceitos base**
  - Máquina de estados
  - Estado
  - Estado composto
  - Evento
  - Transição
  - Acção
  - Guarda
  - Pseudo-estado
  - Entrada
  - Saída

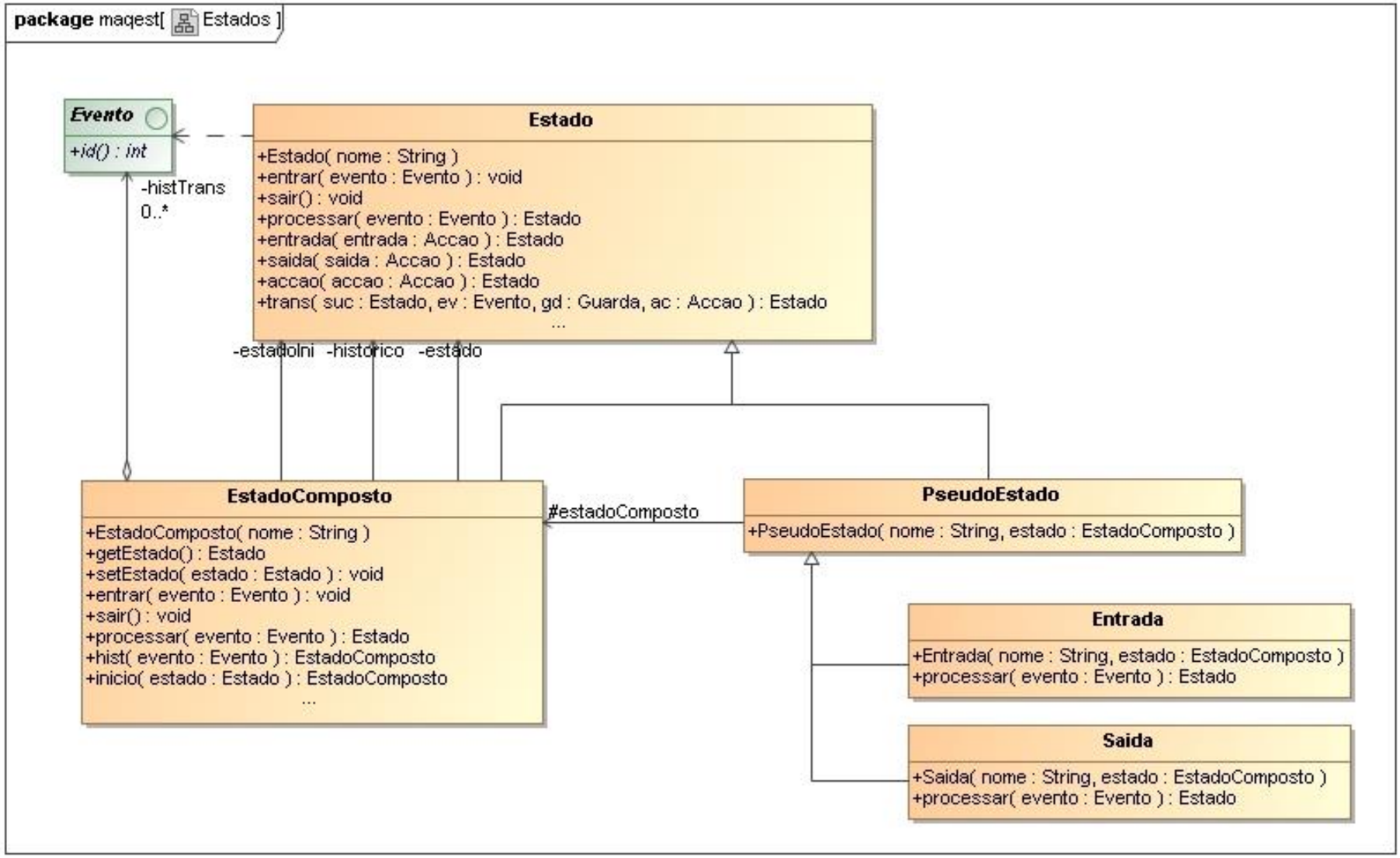
Máquina de Estados Hierárquica

Mecanismo de máquina de estados hierárquica



Máquina de Estados Hierárquica

Definição de *Estados Compostos* e *Pseudo-Estados*



Máquina de Estados Hierárquica

```java
/**
 * Processar evento
 * @param evento Evento
 * @return       Novo estado
 */
public Estado processar(Evento evento) {
    if(accao != null) accao.executar();
    Transicao transicao = transicoes.get(evento);
    if(transicao != null) {
        sair();
        Estado novoEstado = transicao.transitar();
        novoEstado.entrar(evento);
        return novoEstado;
    }
    else return null;
}
```

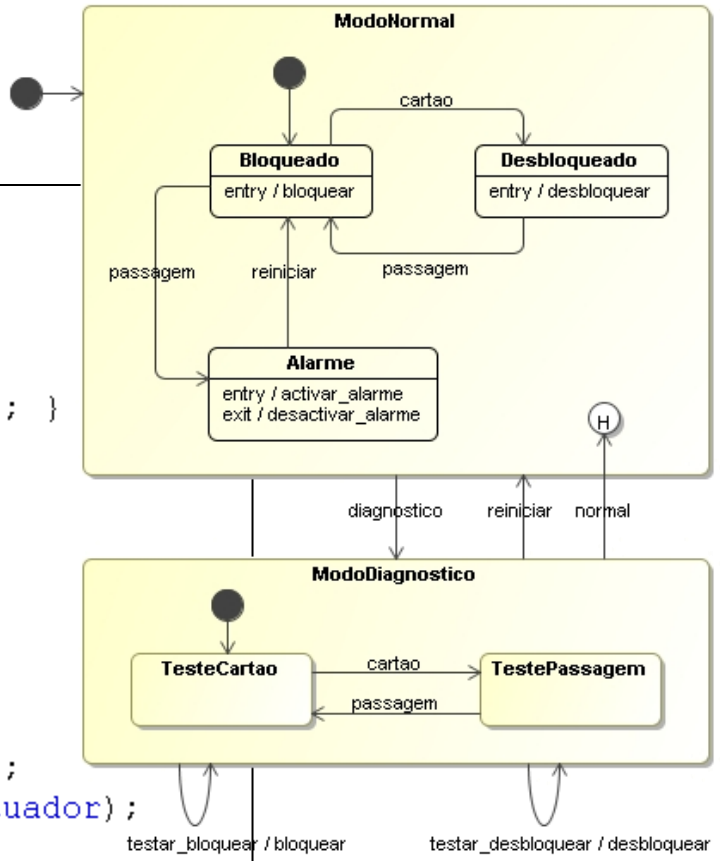
```java
/**
 * Definir transição
 * @param suc   Estado sucessor
 * @param ev    Evento
 * @param gd    Guarda
 * @param ac    Acção
 * @return      Estado
 */
public Estado trans(Estado suc, Evento ev, Guarda gd, Accao ac) {
    Transicao transicao = new Transicao(suc, gd, ac);
    transicoes.put(ev, transicao);
    return this;
}
```

Máquina de Estados Hierárquica

Implementação

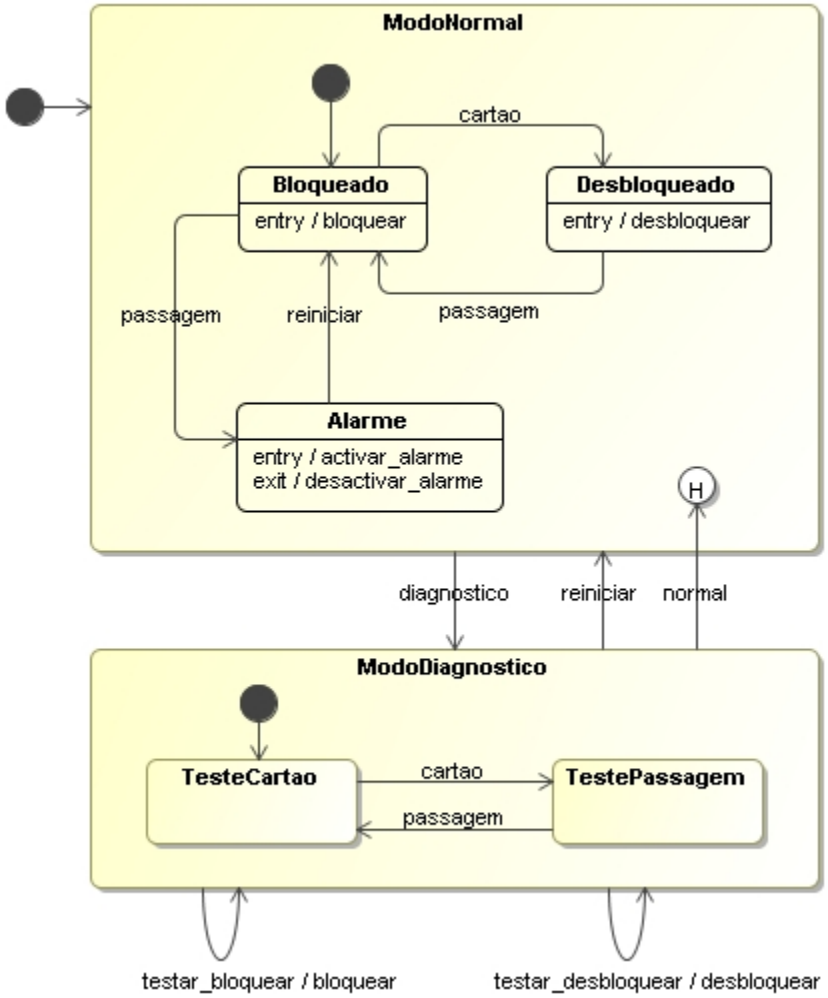
Exemplo: Torniquete

```
public class TorniqueteME extends Torniquete<Estado>
{
    private MaquinaEstados maqEst;

    @Override
    public Estado getEstado() { return maqEst.getEstado(); }

    /**
     * Construtor
     */
    public TorniqueteME() {
        // Acções
        Accao bloquear = new Bloquear(actuador);
        Accao desbloquear = new Desbloquear(actuador);
        Accao activarAlarme = new ActivarAlarme(actuador);
        Accao desactivarAlarme = new DesactivarAlarme(actuador);

        // Estados
        Estado bloqueado = new Estado("Bloqueado");
        Estado desbloqueado = new Estado("Desbloqueado");
        Estado alarme = new Estado("Alarme");
        Estado testeCartao = new Estado("TesteCartao");
        Estado testePassagem = new Estado("TestePassagem");
        EstadoComposto modoNormal = new EstadoComposto("ModoNormal");
        EstadoComposto modoDiag = new EstadoComposto("ModoDiag");
```

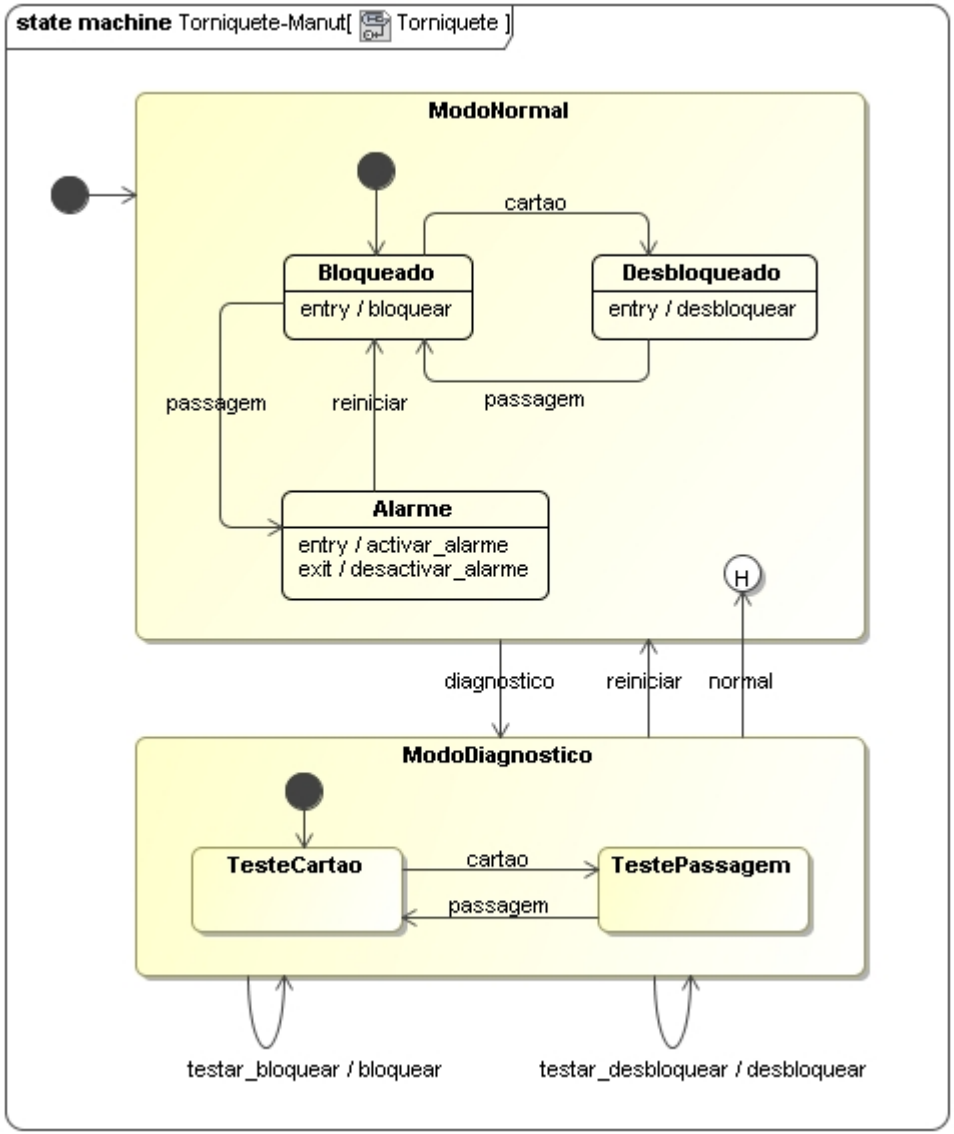


Torniquete: Implementação

```
// Definição da máquina de estados
bloqueado
     .entrada(bloquear)
     .trans(desbloqueado, EventoTorn.CARTAO)
     .trans(alarme, EventoTorn.PASSAGEM);

desbloqueado
     .entrada(desbloquear)
     .trans(bloqueado, EventoTorn.PASSAGEM);

alarme
     .entrada(activarAlarme)
     .saida(desactivarAlarme)
     .trans(bloqueado, EventoTorn.REINICIAR);

testeCartao
     .trans(testePassagem, EventoTorn.CARTAO);

testePassagem
     .trans(testeCartao, EventoTorn.PASSAGEM);

modoNormal
     .inicio(bloqueado)
     .hist(EventoTorn.NORMAL)
     .trans(modoDiag, EventoTorn.DIAGNOSTICO);

modoDiag
     .inicio(testeCartao)
     .trans(modoNormal, EventoTorn.NORMAL)
     .trans(modoNormal, EventoTorn.REINICIAR)
     .trans(modoDiag, EventoTorn.TESTAR_BLOQ, bloquear)
     .trans(modoDiag, EventoTorn.TESTAR_DESBLOQ, desbloquear);
```



Torniquete com Manutenção



Teste: TorniqueteME

ACÇÃO: Bloquear
ESTADO: ModoNormal : Bloqueado

EVENTO: CARTAO
ACÇÃO: Desbloquear
ESTADO: ModoNormal : Desbloqueado

EVENTO: CARTAO
ESTADO: ModoNormal : Desbloqueado

EVENTO: PASSAGEM
ACÇÃO: Bloquear
ESTADO: ModoNormal : Bloqueado

EVENTO: PASSAGEM
ACÇÃO: Activar alarme
ESTADO: ModoNormal : Alarme

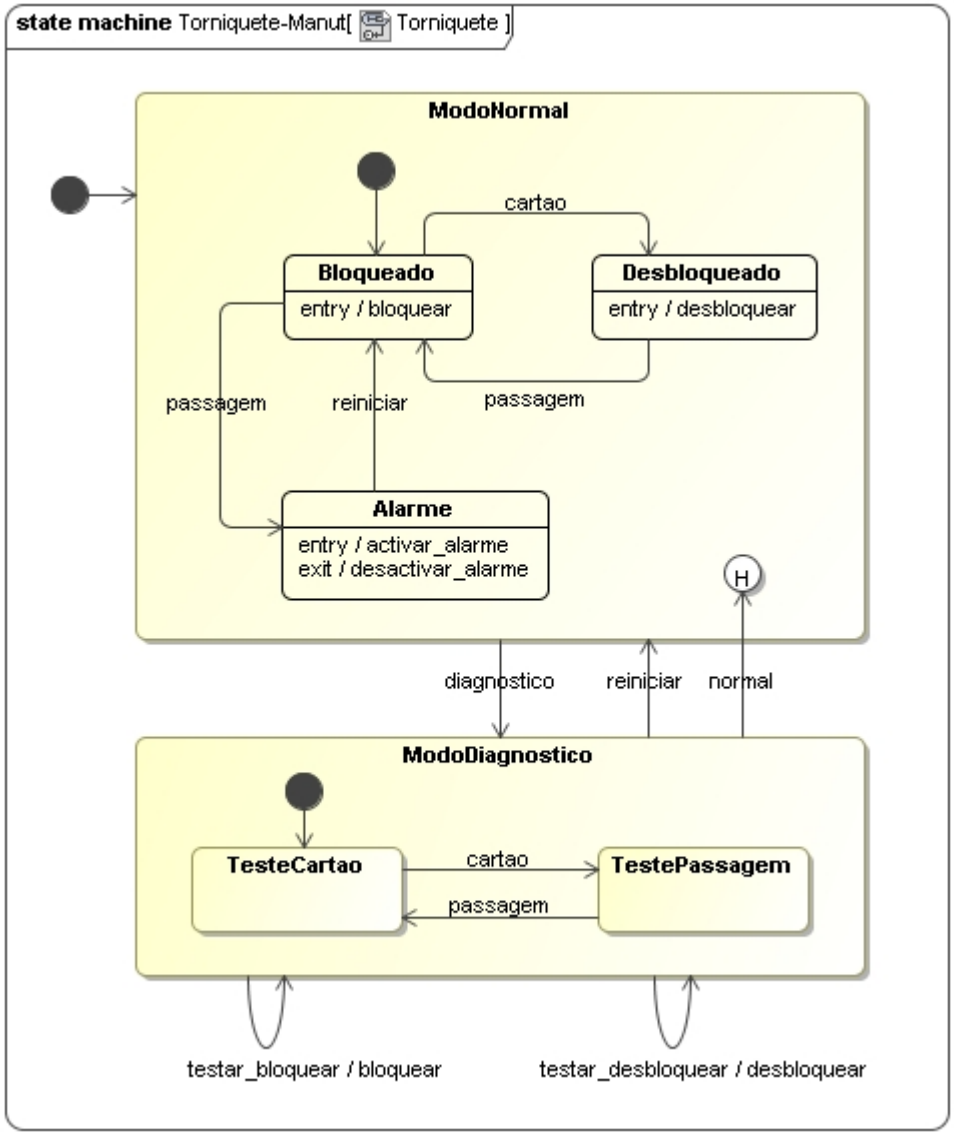
EVENTO: DIAGNOSTICO
ACÇÃO: Desactivar alarme
ESTADO: ModoDiag : TesteCartao

EVENTO: CARTAO
ESTADO: ModoDiag : TestePassagem

EVENTO: TESTAR_BLOQ
ACÇÃO: Bloquear
ESTADO: ModoDiag : TesteCartao

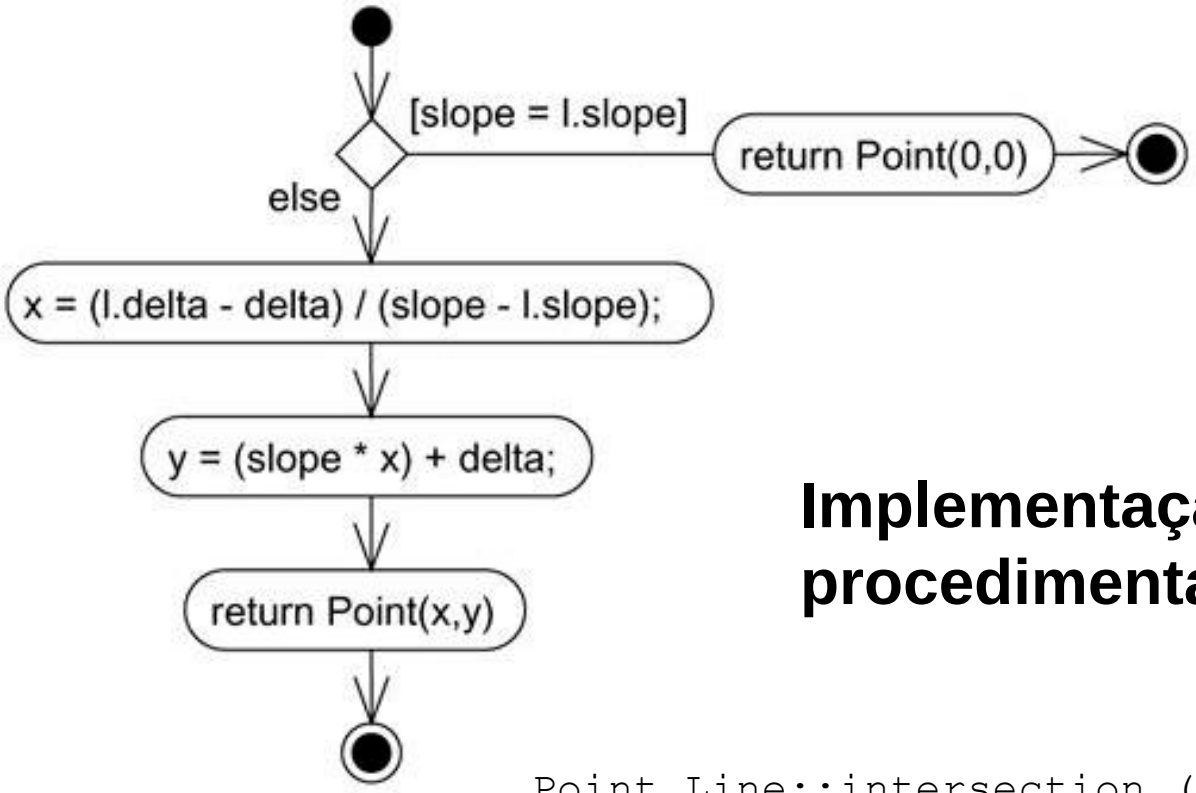
EVENTO: TESTAR_DESBLOQ
ACÇÃO: Desbloquear
ESTADO: ModoDiag : TesteCartao

Torniquete com Manutenção




```
EVENTO: NORMAL
ACÇÃO: Activar alarme
ESTADO: ModoNormal : Alarme

EVENTO: CARTAO
ESTADO: ModoNormal : Alarme

EVENTO: REINICIAR
ACÇÃO: Desactivar alarme
ACÇÃO: Bloquear
ESTADO: ModoNormal : Bloqueado

EVENTO: PASSAGEM
ACÇÃO: Activar alarme
ESTADO: ModoNormal : Alarme

EVENTO: DIAGNOSTICO
ACÇÃO: Desactivar alarme
ESTADO: ModoDiag : TesteCartao

EVENTO: CARTAO
ESTADO: ModoDiag : TestePassagem

EVENTO: REINICIAR
ACÇÃO: Bloquear
ESTADO: ModoNormal : Bloqueado

EVENTO: CARTAO
ACÇÃO: Desbloquear
ESTADO: ModoNormal : Desbloqueado

EVENTO: PASSAGEM
ACÇÃO: Bloquear
ESTADO: ModoNormal : Bloqueado
```

Diagramas de Actividade



Implementação procedimental imperativa

```
Point Line::intersection (l : Line) {
   if (slope == l.slope) return Point(0,0);
   int x = (l.delta - delta) / (slope - l.slope);
   int y = (slope * x) + delta;
   return Point(x, y);
}
```

[UML User Guide, Booch *et al.* 1998]

Bibliografia

[Pressman, 2003]

R. Pressman, *Software Engineering: a Practitioner's Approach*, McGraw-Hill, 2003.

[Gamma et al., 1995]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Shaw & Garlan, 1996]

M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[Vernon, 2013]

V. Vernon, *Implementing Domain Driven Design*, Addison-Wesley, 2013.

[Parnas, 1972]

D. Parnas, *On the Criteria to Be Used in Decomposing Systems into Modules*, Communications of the ACM 15-12, 1968.

[Kruchten, 1995]

F. Kruchten, *Architectural Blueprints - The "4+1" View Model of Software Architecture*, IEEE Software, 12-6, 1995.

[Booch *et al*., 1998]

G. Booch, J. Rumbaugh, I. Jacobson, *UML User Guide*, Addison-Wesley, 1998.

[Booch, 2004]

G. Booch, *Software Architecture*, IBM, 2004.