Engenharia de Software

Implementação

Luís Morgado

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

Diagramas de Transição de Estado

- **Diferentes formas de implementação**
  - **Procedimental**
    - A especificação é traduzida de forma imperativa para a lógica de controlo das operações envolvidas na manutenção de estado
  - **Padrão de arquitectura *Estado***
    - A especificação é implementada com base no padrão de arquitectura *Estado*
  - **Máquinas de estados finitos**
    - A especificação é implementada com base numa tabela de transição de estado e numa tabela de activação de estado
  - **Máquinas de estados finitos hierárquicas**
    - A especificação é implementada com base numa máquina de estados hierárquica, com definição declarativa de estados e transições

Caso Prático

Torniquete de controlo de acessos

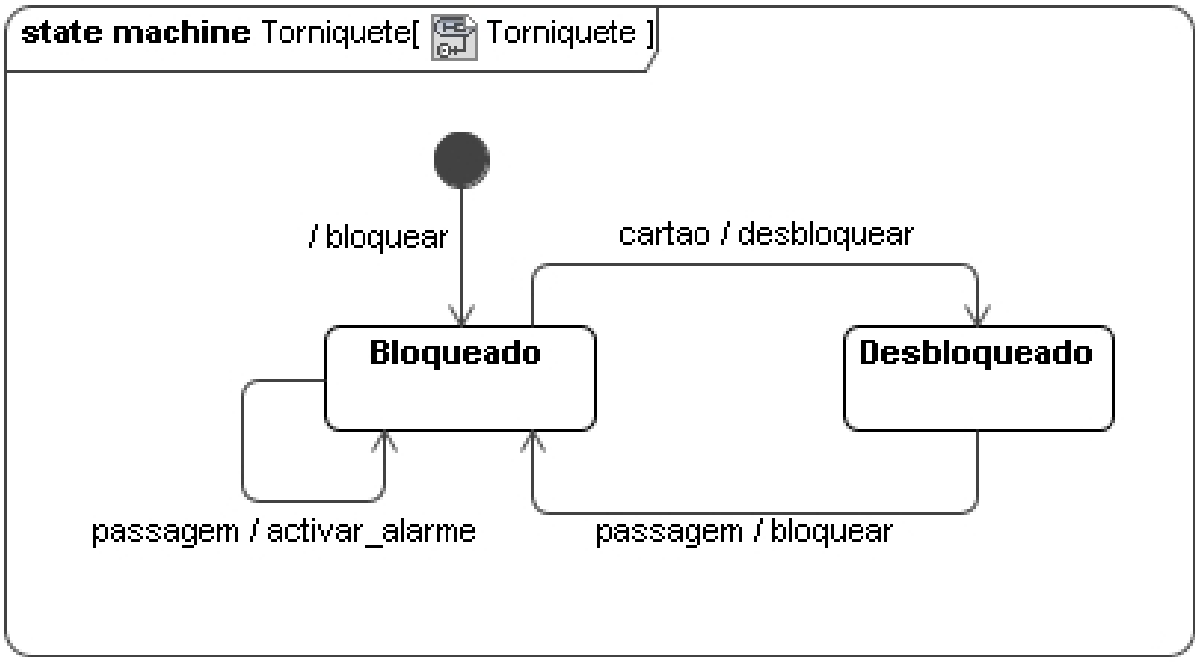
Pretende-se implementar o sistema de controlo de um torniquete de controlo de acessos.

O torniquete é composto por um detector que indica a presença de cartão válido e a ocorrência de passagem e por um actuador composto por um trinco que permite bloquear e desbloquear o acesso e por uma sirene de alarme que pode ser activada ou desactivada.

Por omissão o acesso está bloqueado. Quando é detectado um cartão válido o acesso é desbloqueado, voltando a ser bloqueado após a passagem. No caso da passagem ser forçada deve ser activada a sirene de alarme, que se manterá activa até o sistema ser reiniciado.

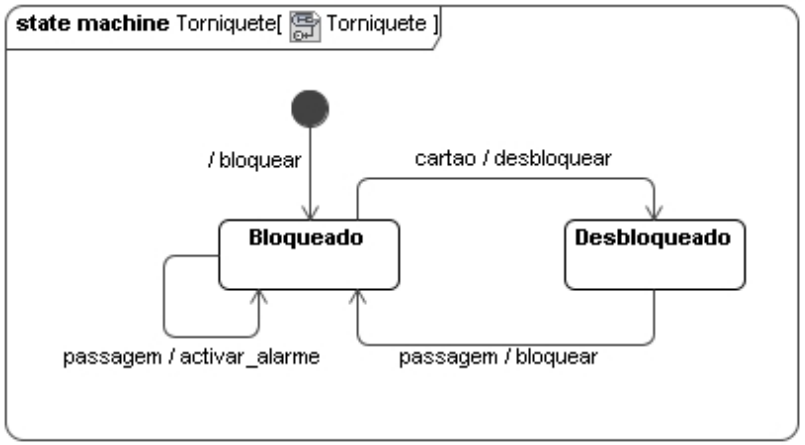
Diagramas de Transição de Estado

Exemplo: Torniquete (sem reiniciar)



Torniquete: Implementação

Implementação
Procedimental da Dinâmica




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

A especificação é traduzida de forma imperativa para a lógica de controlo das operações envolvidas na manutenção de estado, nomeadamente, sob a forma de decisões encadeadas que codificam as transições de estado possíveis

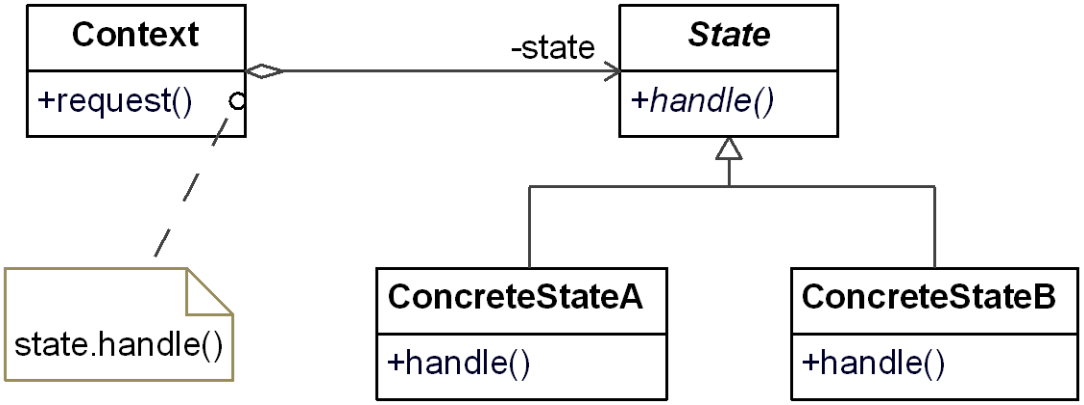
```
public void iniciar() {
    actuador.bloquear();
    setEstado(EstadoTorn.BLOQUEADO);
}
```

```
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;
        }
    }
}
```

Implementação de Dinâmica

- **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 *Estado* (“*State*”)



Contexto (classe *Context*)

- Define a funcionalidade disponibilizada aos clientes para manipulação do contexto
- Mantém uma instância de um estado concreto (subclasses *ConcreteState)* que define o estado atual

Estado (classe *State*)

- Define a funcionalidade geral para encapsular o comportamento associado ao estado do contexto

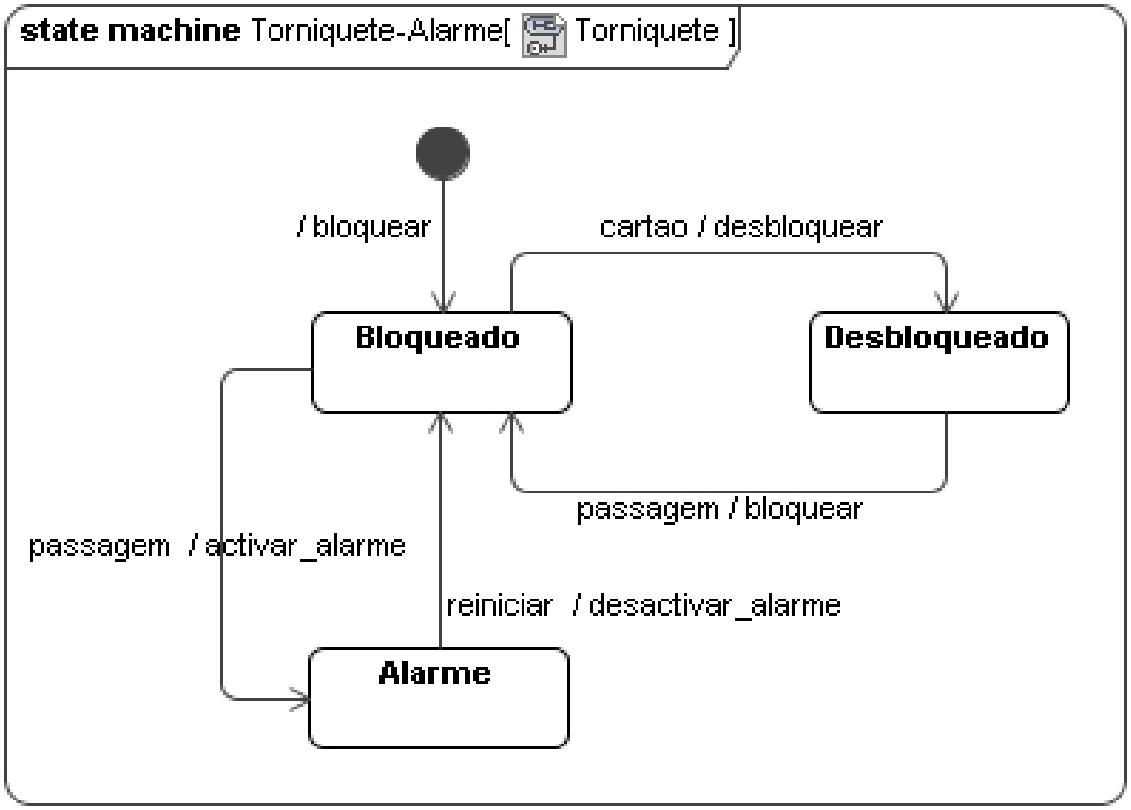
Estados concretos (subclasses *ConcreteState* da classe *State*)

- Cada subclasse implementa o comportamento associado a um estado específico do contexto

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

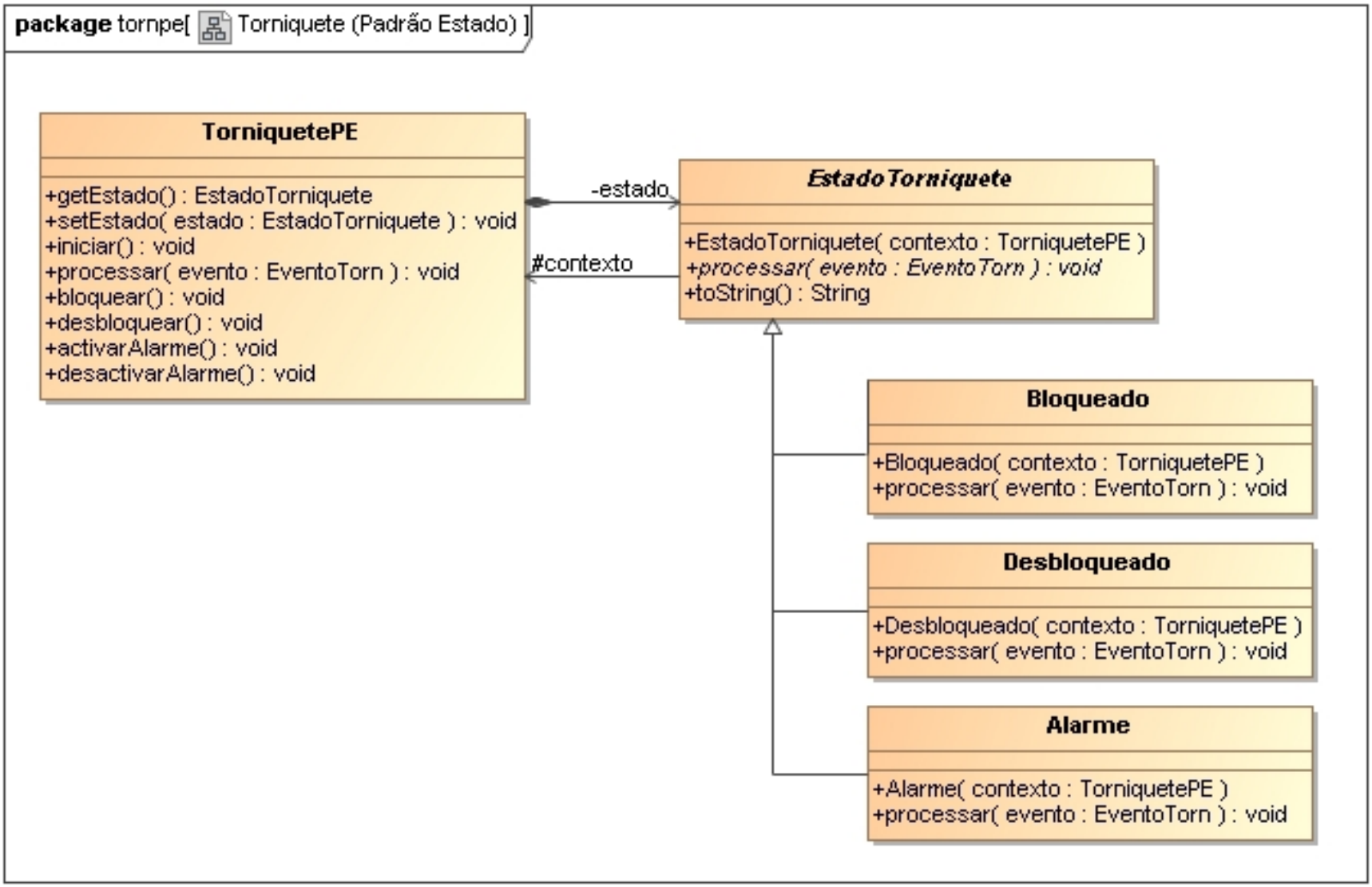
Modelo de Dinâmica

Exemplo: Torniquete



Implementação de Dinâmica

Torniquete com padrão *Estado*



Torniquete: Implementação

Exemplo: 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

Em comparação com a implementação procedimental, a implementação com o padrão *Estado* apresenta uma maior modularidade, pois o processamento associado a cada estado é definido em classes separadas em vez de estar organizado em decisões encadeadas, no entanto, tem como consequência a necessidade de múltiplas classes de representação de estado

```
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;
        }
    }
}
```

Implementação com padrão *Estado*

```
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;
```

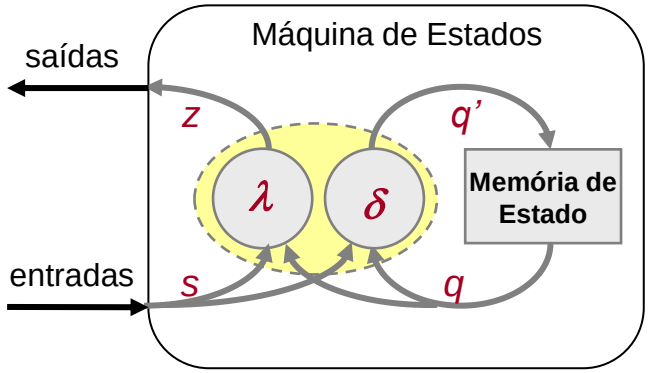
Implementação Procedimental

Padrão de Arquitectura *Estado* (“*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

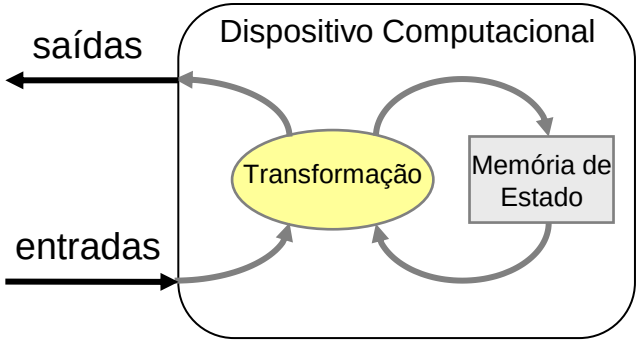
Modelo Formal de Máquina de Estados

- 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**:
    - $\lambda : Q \times \Sigma \rightarrow Z$

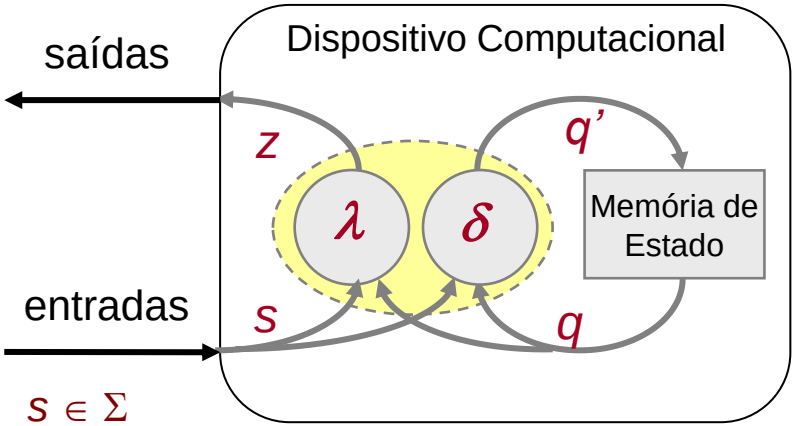


$s \in \Sigma$
$z \in Z$
$q, q' \in Q$

Tipos de Máquinas de Estados



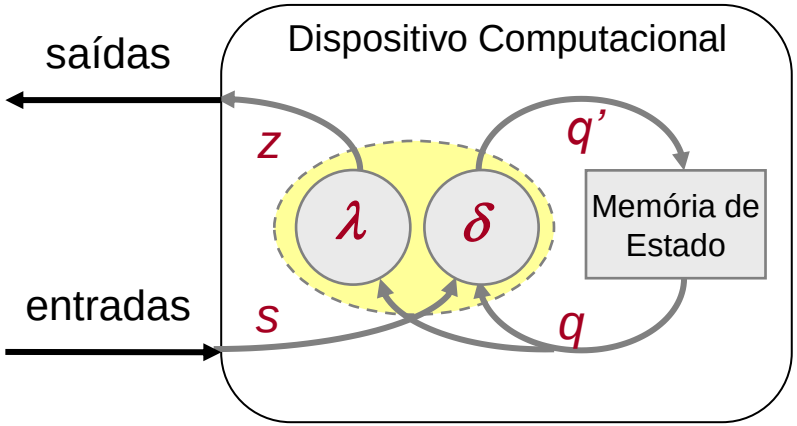
$\delta: Q \times \Sigma \rightarrow Q$
$\lambda: Q \times \Sigma \rightarrow Z$



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

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:

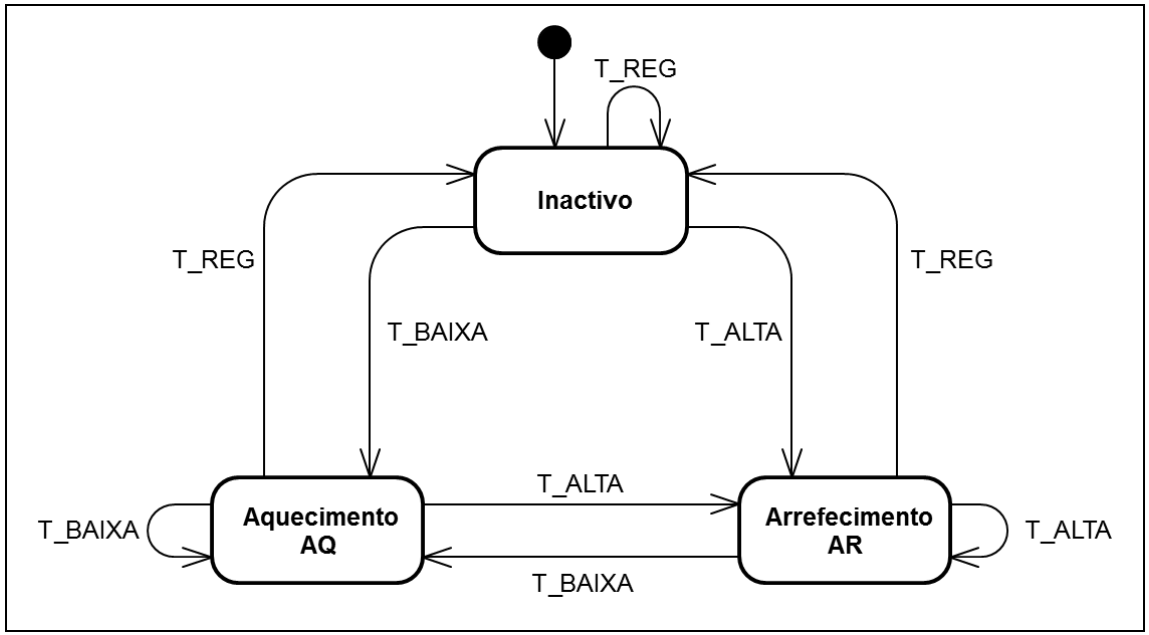
- AQ : sistema de aquecimento é activado;
- AR : sistema de arrefecimento é activado.

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:

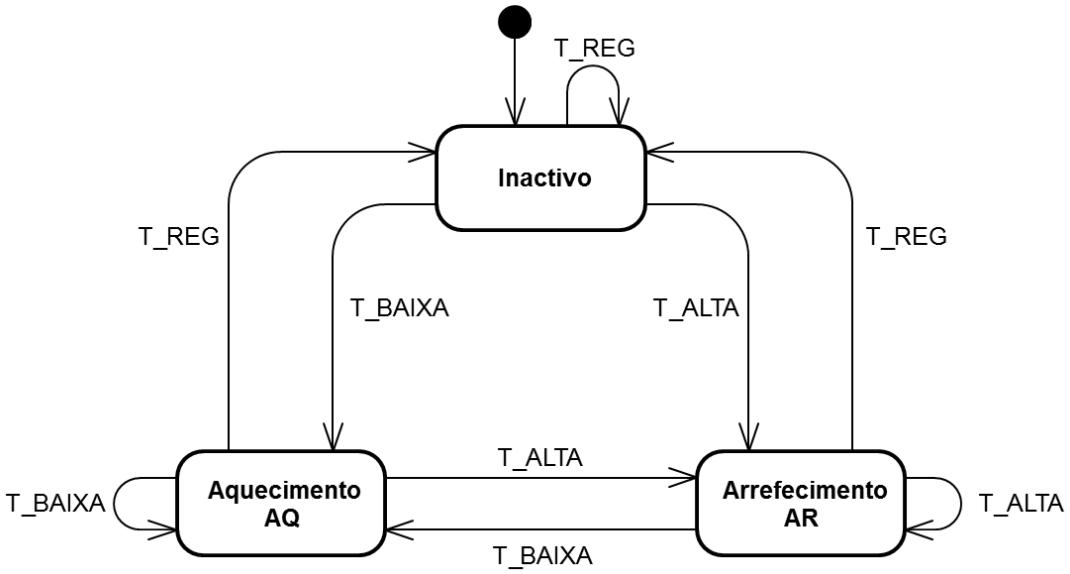
- Inactivo : os sistemas de aquecimento e de arrefecimento estão inactivos;
- Aquecimento : apenas o sistema de aquecimento está activo;
- Arrefecimento : apenas o sistema de arrefecimento está activo.

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 }



Exemplo: 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

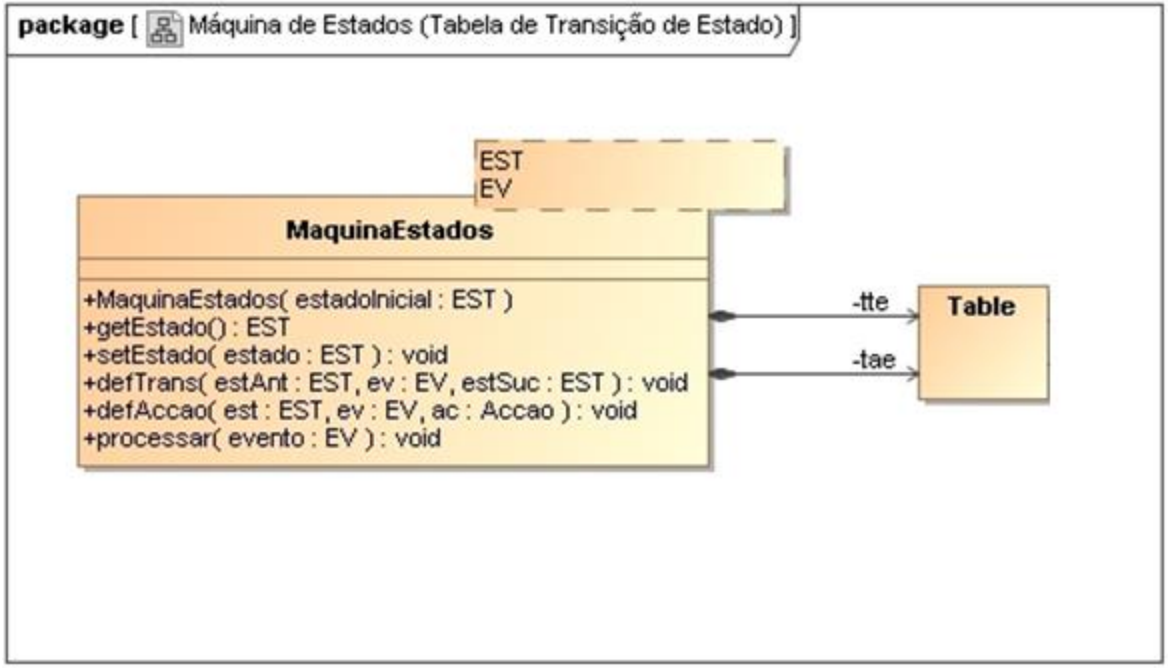
Implementação de Dinâmica

Implementação com máquinas de estados finitos

A especificação é implementada com base numa *tabela de transição de estado* e numa *tabela de activação de estado*

Os estados e os eventos podem assumir diferentes tipos consoante o contexto de utilização, podendo ser representados como tipos genéricos

As acções representam transformações a realizar sobre um contexto de acção, podendo ser definidas com base no padrão de arquitectura *Comando* de modo a abstrair a sua concretização específica

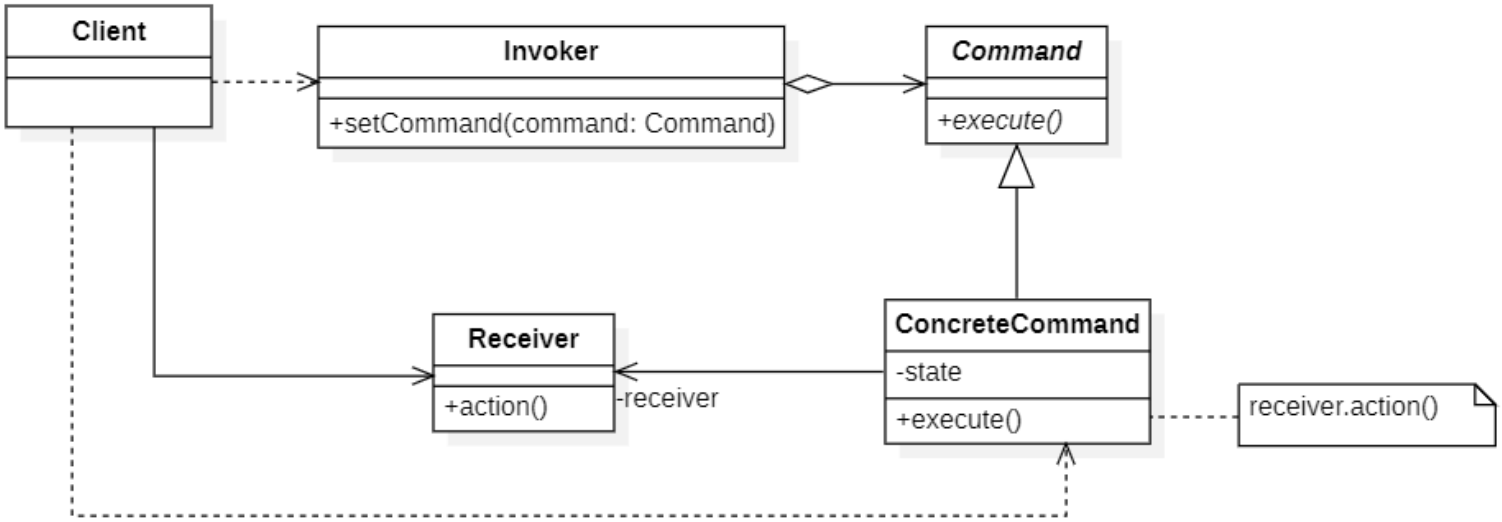


tte – *Tabela de Transição de Estado*

tae – *Tabela de Acção de Estado*

Padrão *Comando* (“*Command*”)

Estrutura



Participantes

Comando (*Command*)
- Declara a interface para execução de uma acção

Comando concreto (*ConcreteCommand*)
- Define a ligação entre receptor e comando concreto
- Implementa a execução da acção por activação da acção correspondente no receptor

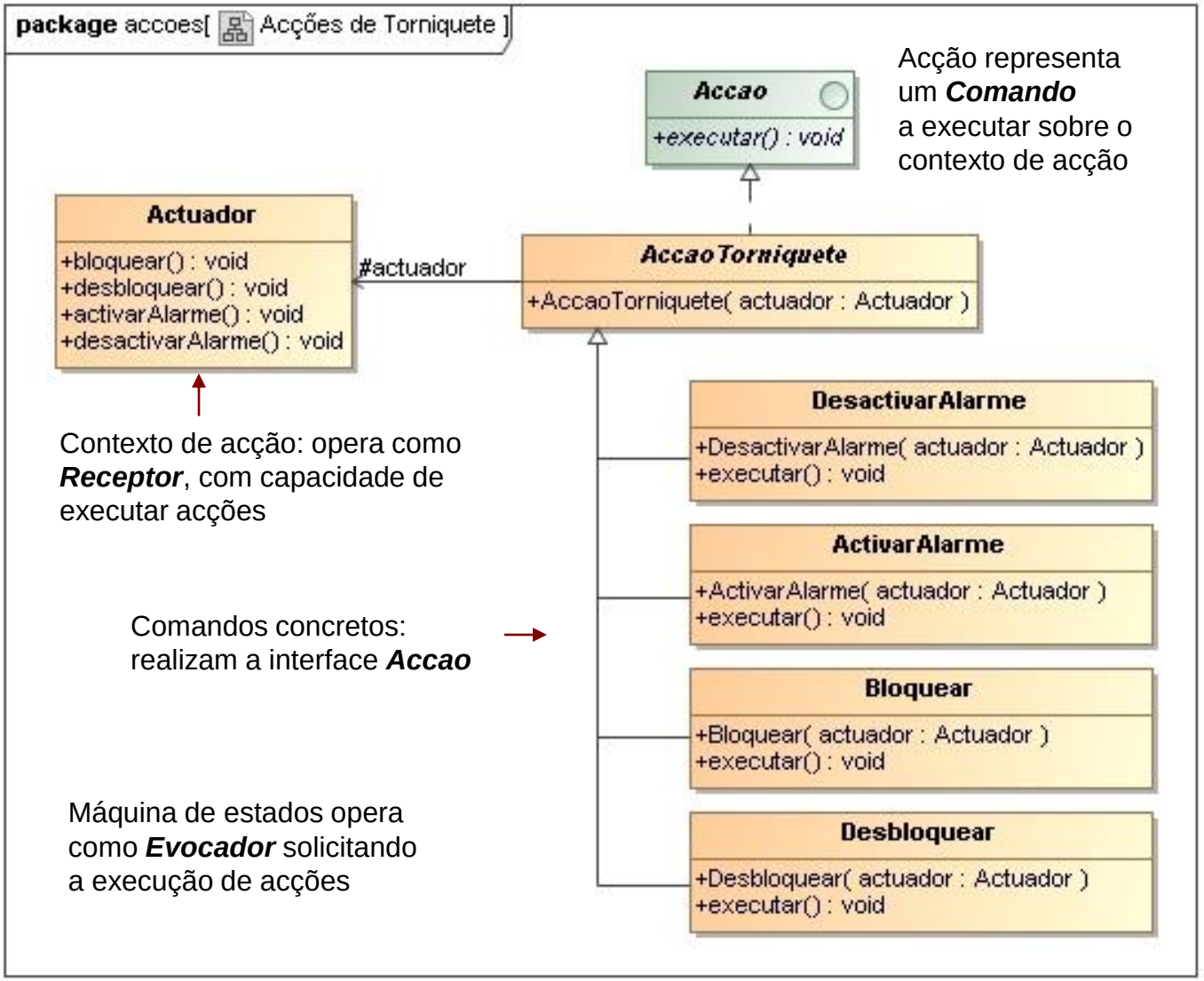
Cliente (*Client*)
- Cria o comando concreto e define o receptor

Evocador *(Invoker*)
- Solicita ao comando a execução da acção

Receptor (*Receiver*)
- Tem capacidade de executar a acção

Torniquete: Máquina de Estados

Implementação com Padrão *Comando*



Torniquete: Máquina de Estados

Definição de TTE e TAE

```
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;
```

Processamento geral de transições de estado

```
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;
}
```

Definição de transições de estado e de acções associadas para o contexto concreto

```
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));
}
```

Caso Prático

Torniquete de controlo de acessos com manutenção

Pretende-se implementar o sistema de controlo de um torniquete de controlo de acessos.

O torniquete é composto por um detector que indica a presença de cartão válido e a ocorrência de passagem e por um actuador compos por um trinco que permite bloquear e desbloquear o acesso e por um sirene de alarme que pode ser activada ou desactivada.

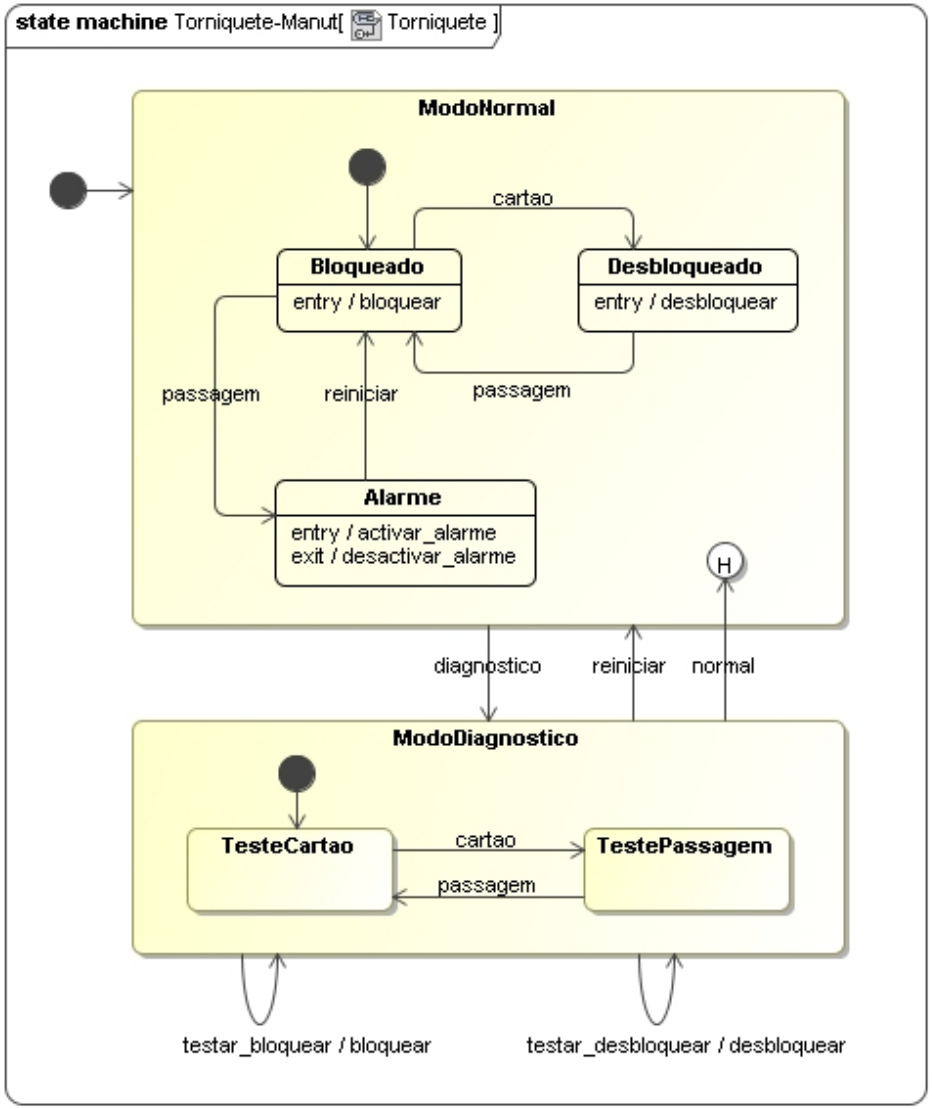
Por omissão o acesso está bloqueado. Quando é detectado um cartã válido o acesso é desbloqueado, voltando a ser bloqueado após a passagem. No caso da passagem ser forçada deve ser activada a sirene de alarme, que se manterá activa até o sistema ser reiniciado.

Para além do modo normal de operação, o torniquete suporta um modo de diagnóstico. Quando entra em modo de diagnóstico, fica à espera de um cartão válido para teste. Quando é detectado um cartão válido, o acesso é desbloqueado, ficando à espera de teste de passagem, ao detectar uma passagem retorna à situação de teste de cartão. Em ambas as situações é possível testar o bloqueio e desbloqueio do trinco, bem como reiniciar o sistema.

Quando é seleccionada a opção para regressar ao modo normal, o sistema regressa à situação em que se encontrava antes de entrar em diagnóstico.

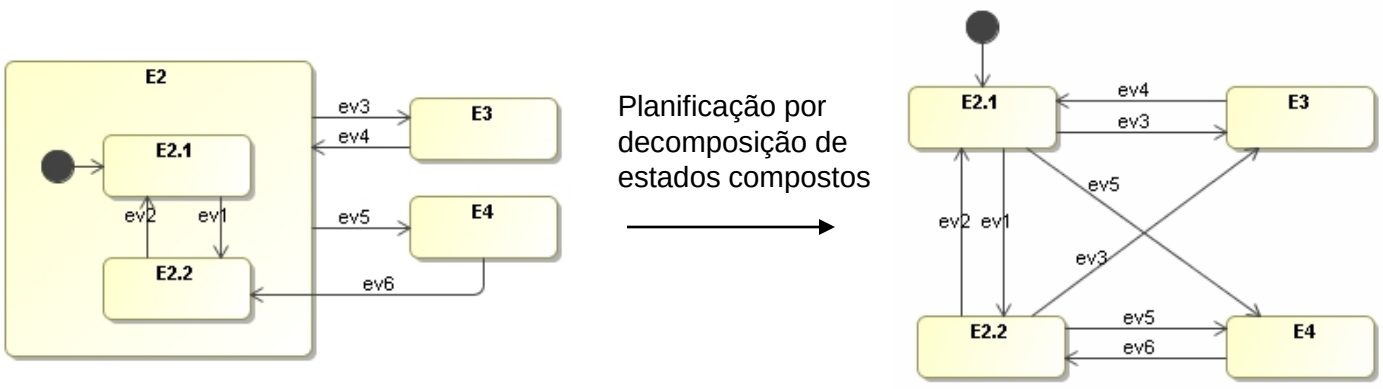
Máquinas de Estados Hierárquicas

Exemplo: Torniquete com manutenção



Máquinas de Estados Hierárquicas

- **Implementação**
  - **Máquina de estados hierárquica**
  - **Padrão de arquitectura *Estado* (“*State*”)**
    - Estados compostos representam sub-máquinas de estado
    - Estados compostos devem manter histórico
  - **Planificação**
    - Decomposição de estados compostos
    - Implementação com tabela de transição de estados
    - Requer mecanismo de manutenção de histórico (superficial/profundo)

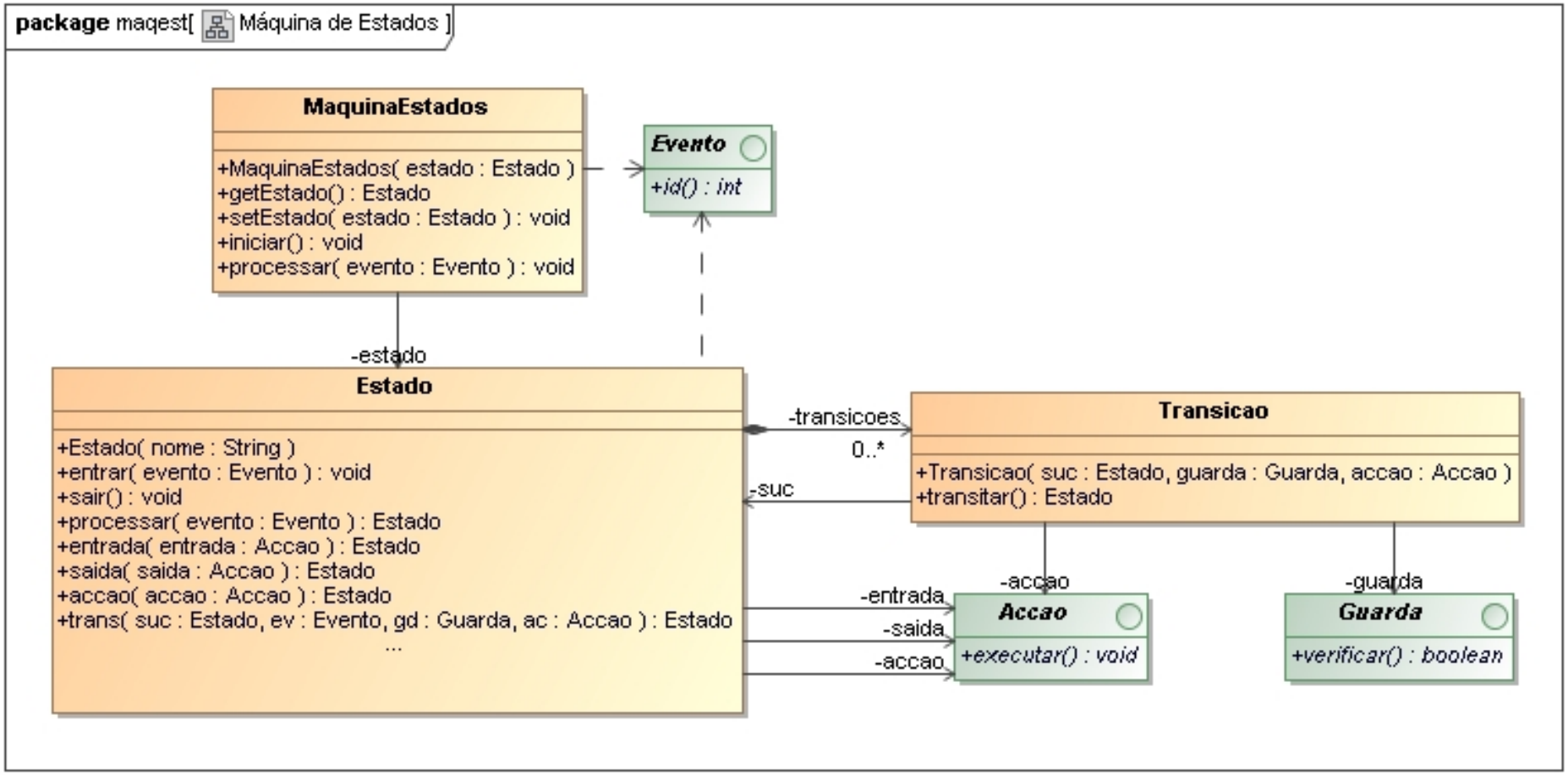


Máquina de Estados Hierárquica

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

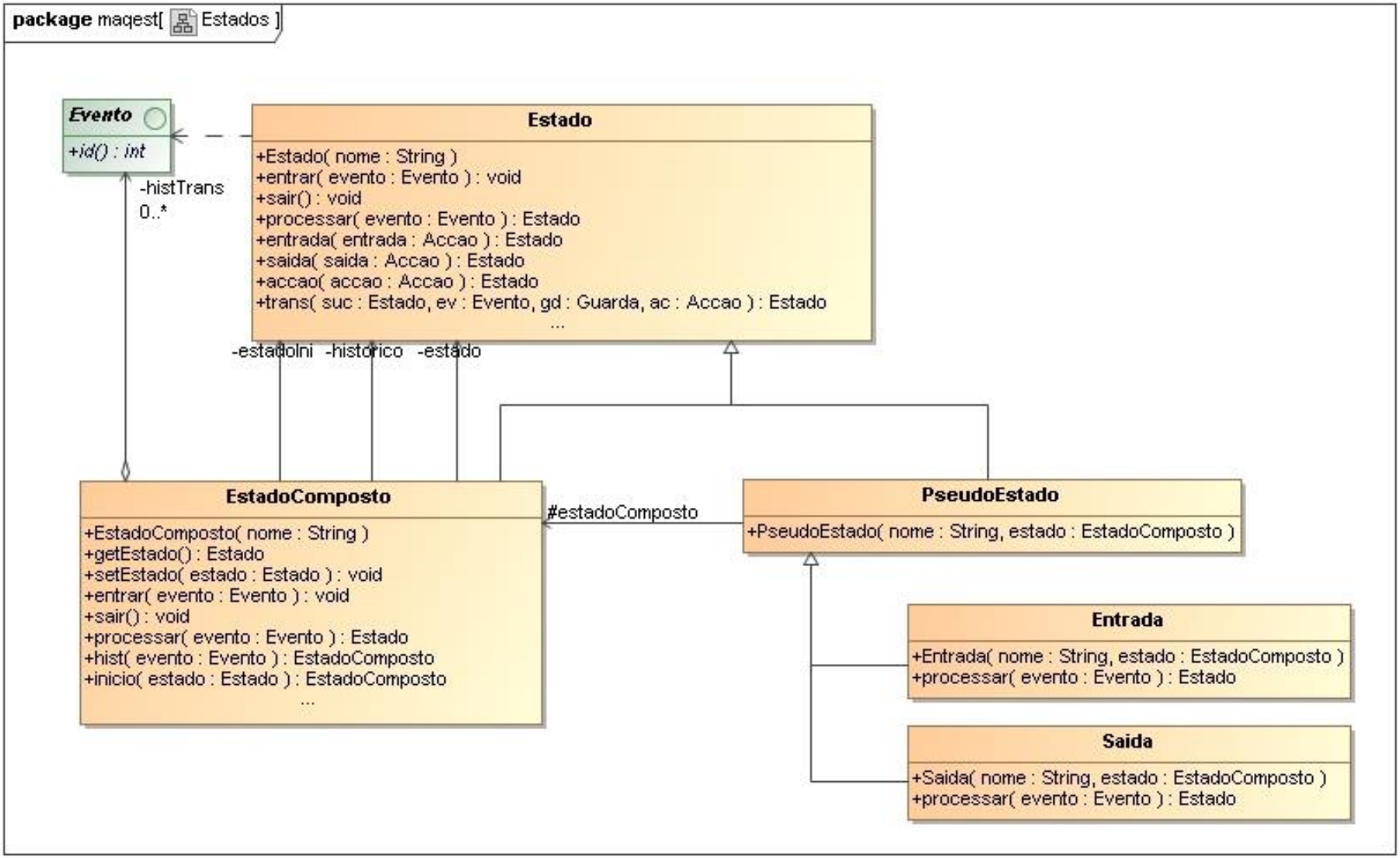
Máquina de Estados Hierárquica

Mecanismo de máquina de estados hierárquica



Máquina de Estados Hierárquica

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



Máquina de Estados Hierárquica

Exemplo de implementação

Processamento geral de transições de estado

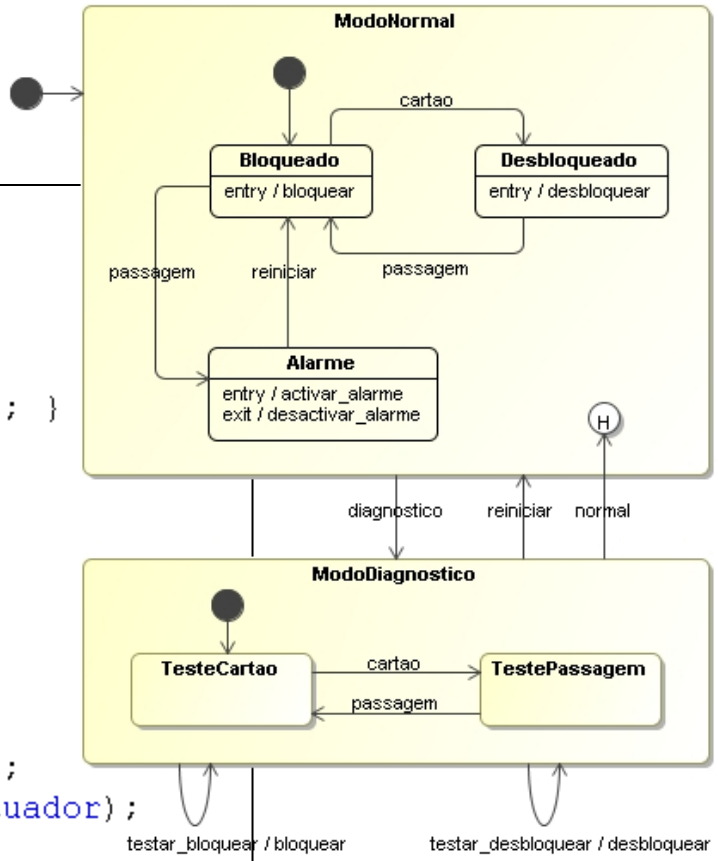
```
/**
 * Processar evento
 * @param evento Evento
 * @return       Novo estado
 */
public Estado processar(Evento evento) {
    if(accao != null) accao.executar();
    Transicao transicao = transicoes.get(evento);
    if(transicao != null) {
        sair();
        Estado novoEstado = transicao.transitar();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

Definição de transições de estado e de acções associadas

```
/**
 * 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;
}
```

Torniquete: Implementação

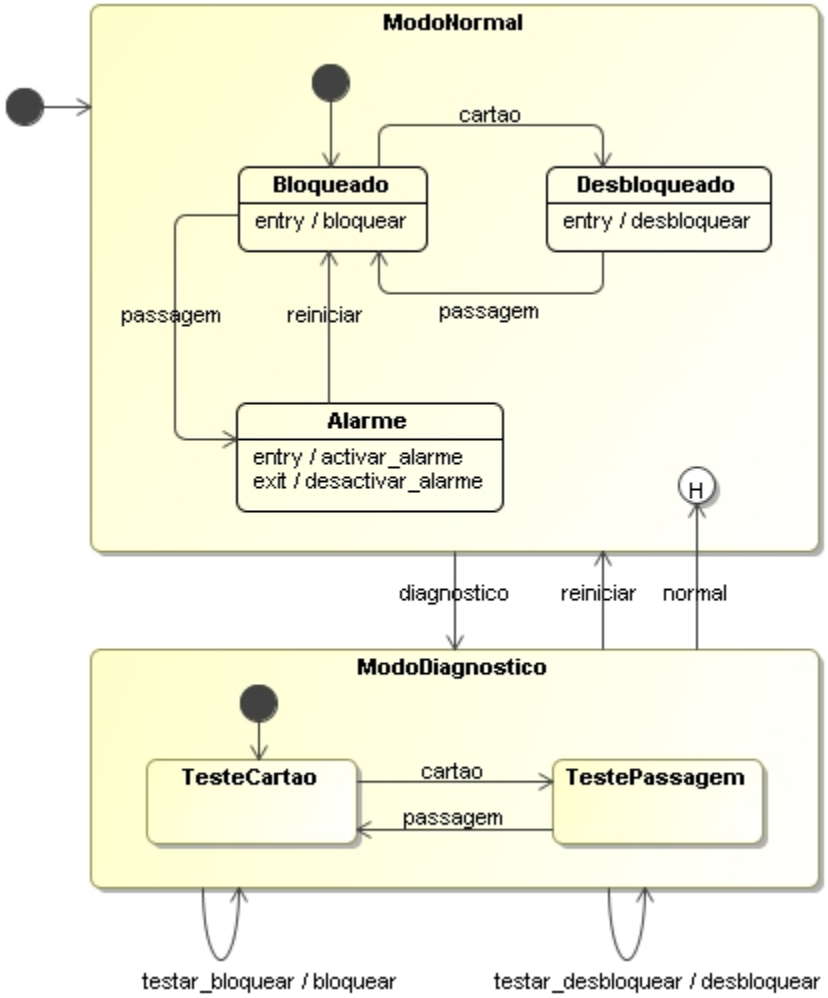
Com um mecanismo de máquina de estados hierárquica é possível a configuração declarativa da máquina de estados




```
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");
```

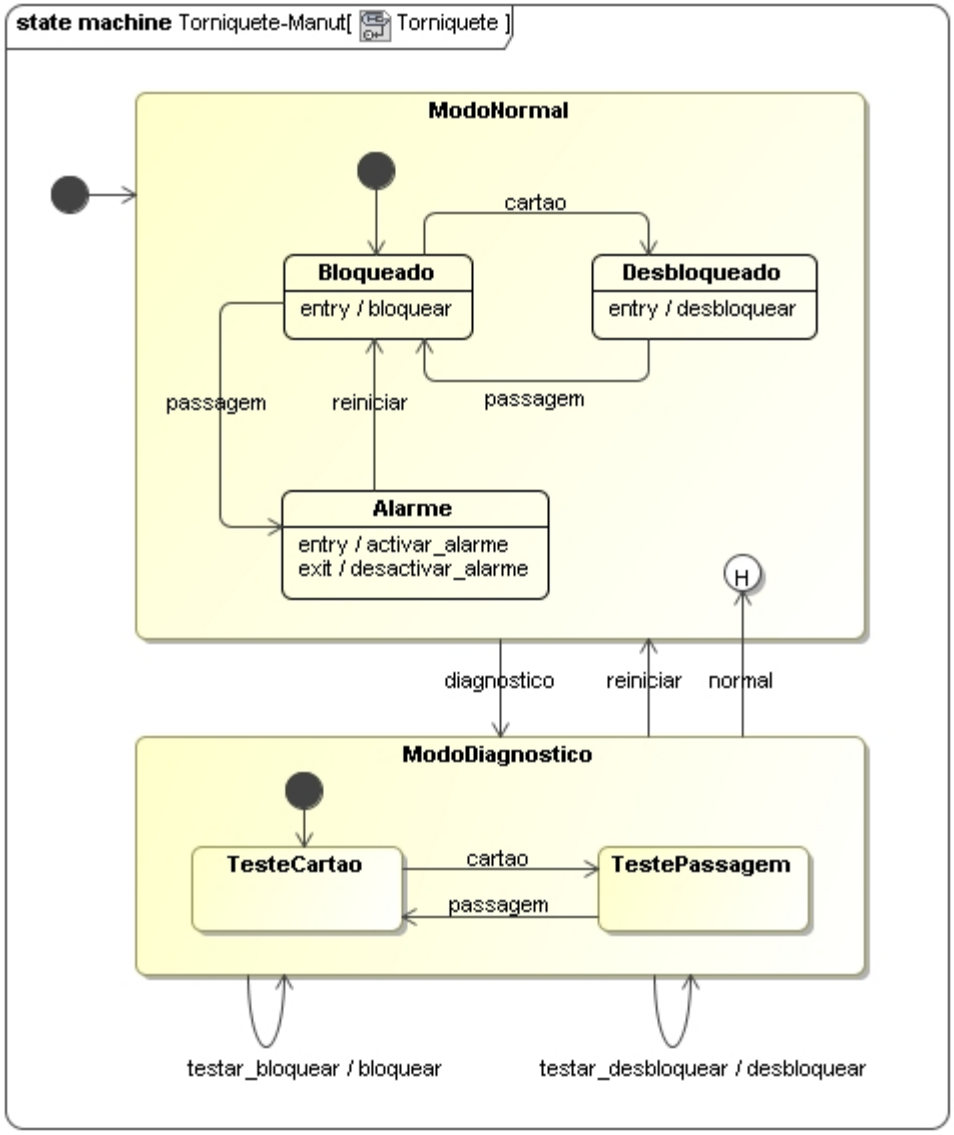
Torniquete: Implementação

```
// 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

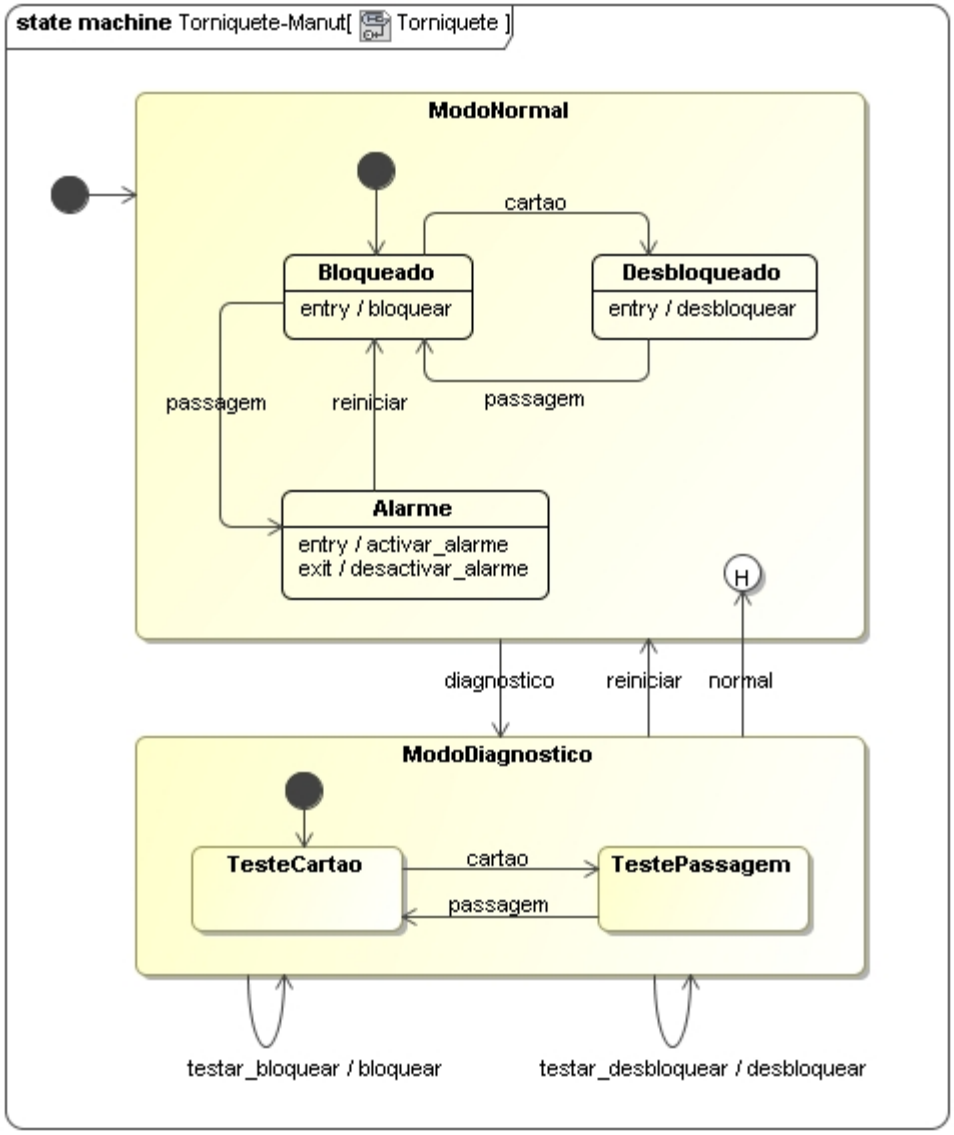
Exemplo de activação




```
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

Exemplo de activação (cont.)




```
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
```

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.