Software Design Patterns

FSM – *Finite-State Machine*

Complexidade do problema

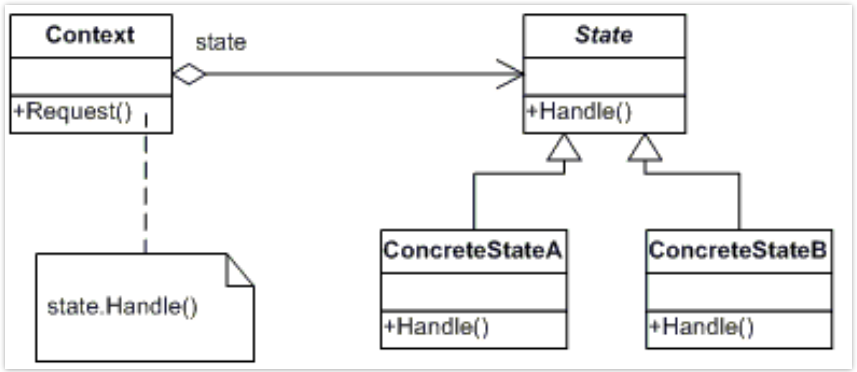
- Quando um problema a resolver ou um programa a desenvolver se torna demasiado complicado, recorrem-se a técnicas de organização e/ou redução da complexidade para permitir a sua resolução

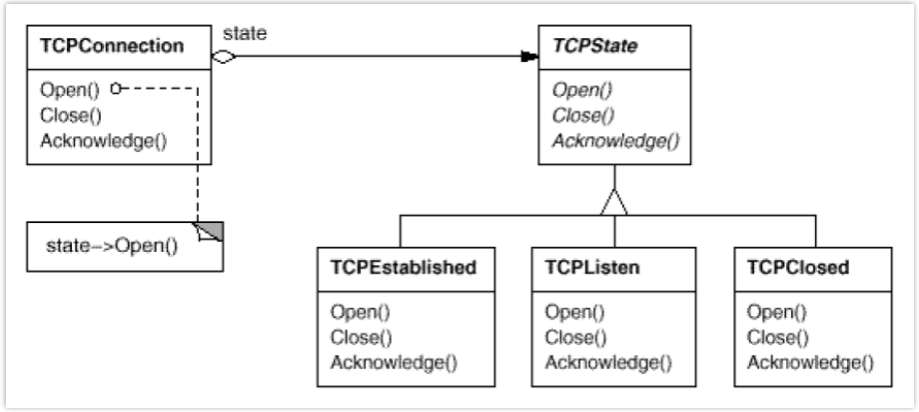
- Nos problemas em que é possível identificar fluxos de evolução dos programas entre uma sucessão de situações bem identificadas, normalmente designadas por "estados", recorre-se ao padrão de desenvolvimento designado por Máquina de Estados
  - **FSM – *Finite-State Machine***

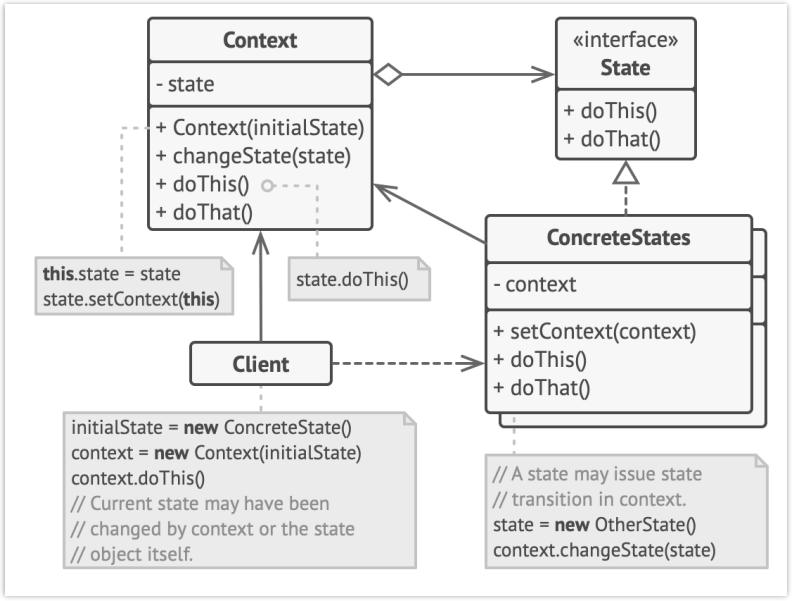
FSM – *Finite-State Machine*

- Usado quando uma determinada entidade é alvo de um processo de evolução entre estados sucessivos, no contexto dos quais pode ser alvo de eventos ou ações adequadas a cada estado e que lhe permitem evoluir/adequar os comportamentos ou os próprios dados
  - A entidade pode ser um objeto, um grupo de objetos ou toda a aplicação
  - Em cada estado da sua evolução existem operações associadas, eventualmente diferentes das existentes noutros estados

State design pattern



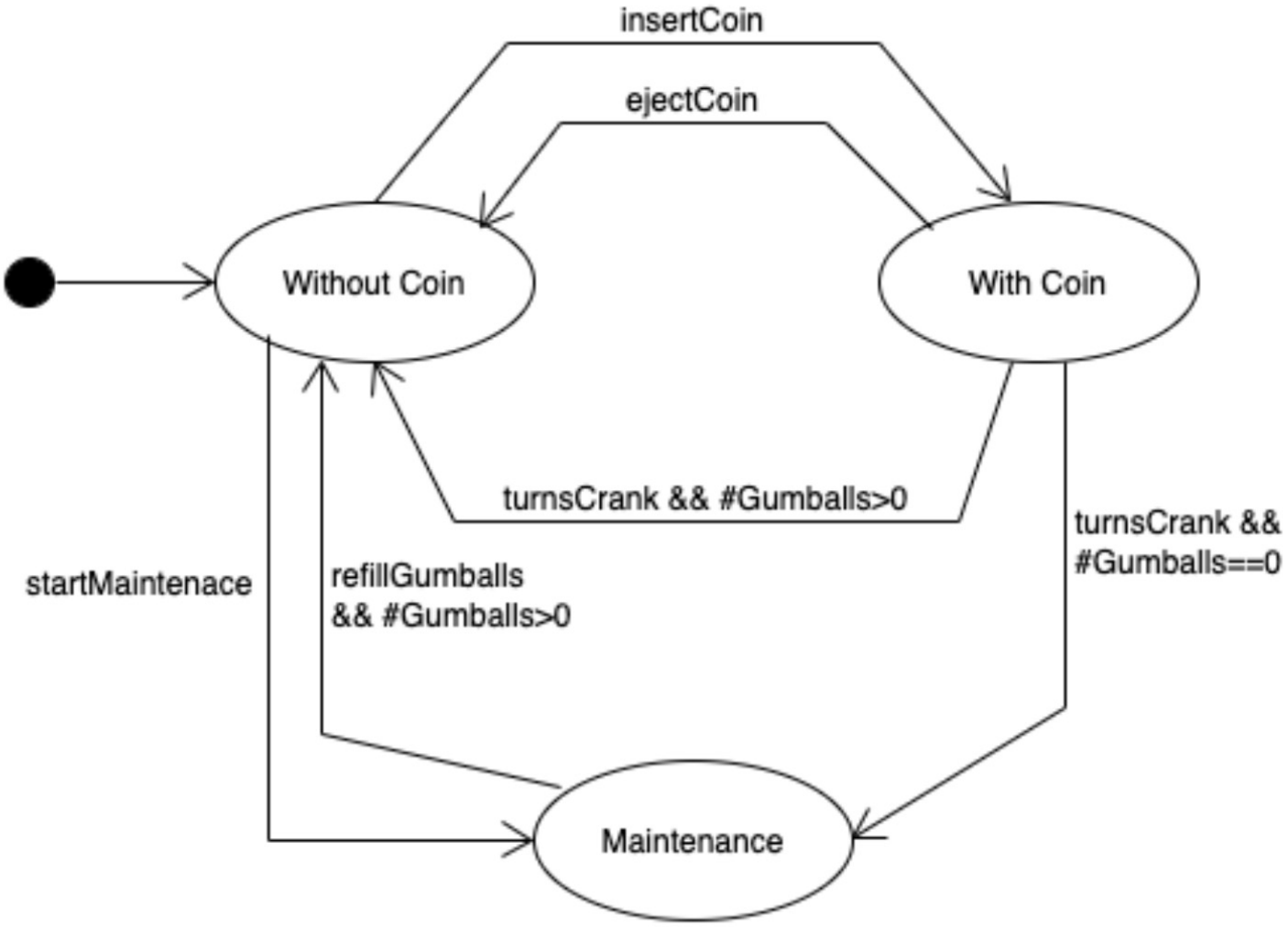




Exemplo – "Máquina de gomas"

- Vamos supor que se pretende implementar um simulador de uma máquina de gomas...
- A máquina de gomas permite disponibilizar gomas quando é introduzida uma moeda e rodado o manípulo
- Ações (todas as ações permitidas independentemente do estado):
  - Introduzir moeda
  - Retirar moeda
  - Rodar o manípulo
  - Colocar em manutenção
  - Abastecer
- Estados
  - Sem moeda introduzida
  - Com moeda introduzida
  - Em manutenção

Diagrama de estados



Máquina de estados

- A reação aos diversos eventos depende da situação (estado) em que a “máquina de gomas” se encontra

- A reação da máquina a um evento pode ser:
  - ignorar o evento
    - O evento pode não fazer sentido para o estado em que atualmente se encontra
  - alterar o seu estado
    - executar ações, podendo as ações alterar o estado em que se encontra

Criação da máquina de estados

- Depois de ter sido estudado o problema a resolver e ter sido criado o diagrama de estados...
- Criação das classes que suportam o modelo de dados
  - Não dependente da forma como as suas ações e transformações dos dados será realizada
    - Pode ser através de uma máquina de estado, mas o modelo de dados deve estar preparado de forma genérica para se adequar a qualquer modelo
    - Deve garantir que todas as regras/lógica do modelo ("regras do negócio") são cumpridas
- Criação de um conjunto de constantes ou *enum* que permita identificar todos os estados
- Criação de uma interface com métodos que permitam representar todas as transições
  - Incluir também método *getState* que permita retornar o estado atual (constante definida pelo *enum*)

Criação da máquina de estados

- Criação de uma classe abstrata *StateAdapter* que permita
  - disponibilizar implementações por omissão para todas as transições
  - gerir referências para a classe base que representa o modelo de dados e para a classe que representa o contexto geral da máquina de estados
  - fornecer método para alterar o estado atual no contexto

- Criação das classes que representam cada estado (derivadas a partir da classe *StateAdapter*)

Criação da máquina de estados

- Criação da classe *Context*
  - Referência para o estado atual, que poderá ser iniciado no seu construtor
  - Referência para o modelo de dados
  - Método público que permita obter o estado atual
  - Método *package-private* que permita alterar o estado atual
  - Métodos que reencaminhem as ações/eventos para o estado ativo
  - Conjunto de métodos que permita obter os dados necessários à interação com o utilizador ou com os restantes módulos do programa
- A classe *Context* deve garantir que as alterações ao modelo de dados apenas ocorrem no contexto dos métodos referentes a transições de estado
  - Não deve disponibilizar métodos ou retornar referências para objetos que permitam alteração direta dos dados

GumballMachineData

```
public class GumballMachineData {
    private int count = 0;

    public GumballMachineData(int count) { this.count = count; }

    public int getCount() { return count; }

    public void refillGumballs(int count) {
        if (count>0)
            this.count += count;
    }

    public boolean getGumball() {
        if (count>0) {
            count--;
            return true;
        }
        return false;
    }

    @Override
    public String toString() {
        return String.format("Gumball Machine with %d gumball(s)",count);
    }
}
```

IState e enum State

```
public enum State {
    MAINTENANCE, WITH_COIN, WITHOUT_COIN
}

public interface IState {
    boolean insertCoin();
    boolean ejectCoin();
    boolean turnsCrank();
    boolean startMaintenance();
    boolean refillGumballs(int count);

    State getState();
}
```

StateAdapter

```
abstract class StateAdapter implements IState {
    protected Context context;
    protected GumballMachineData data;

    protected StateAdapter(Context context, GumballMachineData data) {
        this.context = context;
        this.data = data;
    }

    protected void changeState(IState newState) { context.changeState(newState); }

    @Override
    public boolean insertCoin() { return false; }

    @Override
    public boolean ejectCoin() { return false; }

    @Override
    public boolean turnsCrank() { return false; }

    @Override
    public boolean startMaintenance() { return false; }

    @Override
    public boolean refillGumballs(int count)  { return false; }
}
```

WithoutCoinState

```
class WithoutCoinState extends StateAdapter {
    WithoutCoinState(Context context, GumballMachineData data) {
        super(context,data);
    }

    @Override
    public boolean insertCoin() {
        changeState(new WithCoinState(context,data));
        return true;
    }

    @Override
    public boolean startMaintenance() {
        changeState(new MaintenanceState(context,data));
        return true;
    }

    @Override
    public State getState() { return State.WITHOUT_COIN; }
}
```

WithCoin

```
class WithCoinState extends StateAdapter {
    WithCoinState(Context context, GumballMachineData data) {
        super(context,data);
    }

    @Override
    public boolean ejectCoin() {
        changeState(new WithoutCoinState(context, data));
        return true;
    }

    @Override
    public boolean turnsCrank() {
        if (data.getGumball() && data.getCount()>0) {
            changeState(new WithoutCoinState(context, data));
            return true;
        }
        changeState(new MaintenanceState(context,data));
        return false;
    }

    @Override
    public State getState() { return State.WITH_COIN; }
}
```

MaintenanceState

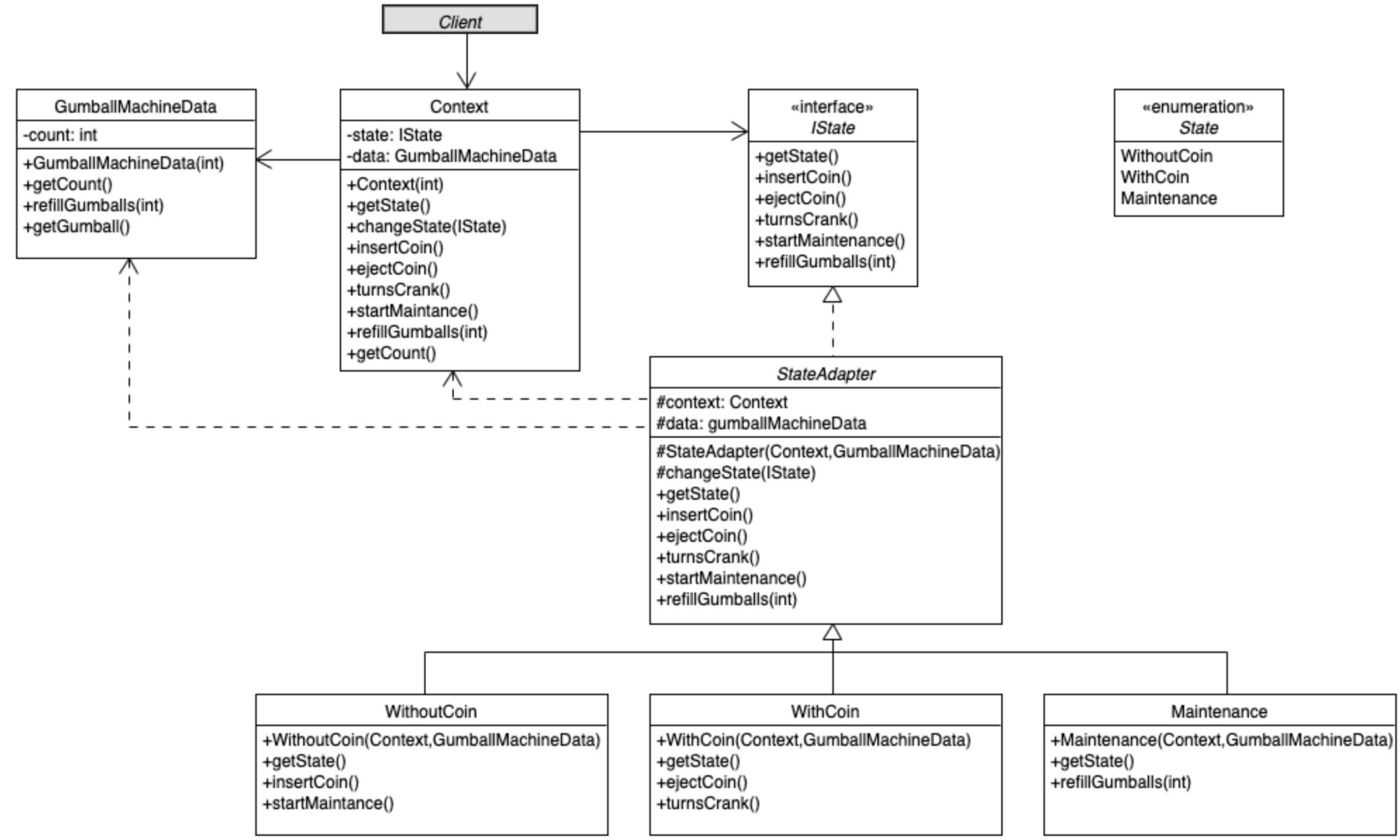
```
class MaintenanceState extends StateAdapter {
    MaintenanceState(Context context, GumballMachineData data) {
        super(context,data);
    }

    @Override
    public boolean refillGumballs(int count) {
        data.refillGumballs(count);
        if (data.getCount()>0) {
            changeState(new WithoutCoinState(context, data));
            return true;
        }
        return false;
    }

    @Override
    public State getState() {
        return State.MAINTENANCE;
    }
}
```

Context – Máquina de Estados

```
public class Context {
    private GumballMachineData data;
    private IState state;

    public Context(int count) {
      data = new GumballMachineData(count);
      state = new WithoutCoinState(this,data);
    }

    public State getState() {
        return state.getState();
    }

    void changeState(IState newState) {
        this.state = newState;
    }

 // ---- get data ----

    public int getCount() {
        return data.getCount();
    }

// ======================>
```

```
    public boolean insertCoin() {
        return state.insertCoin();
    }

    public boolean ejectCoin() {
        return state.ejectCoin();
    }

    public boolean turnsCrank() {
        return state.turnsCrank();
    }

    public boolean startMaintenance() {
        return state.startMaintenance();
    }

    public boolean refillGumballs(int count) {
        return state.refillGumballs(count);
    }

}
```

"Máquina de gomas"



Main

```
public class Main {
    public static void main(String[] args) {
        Context fsm = new Context(100);
        GumballMachineUI ui = new GumballMachineUI(fsm);
        ui.start();
    }
}
```

GumballMachineUI (1/2)

```
public class GumballMachineUI {
    private Context fsm;
    public GumballMachineUI(Context fsm) { this.fsm = fsm; }

    public boolean start() {
        while (switch (fsm.getState()) {
                  case MAINTENANCE  -> maintenance();
                  case WITH_COIN    -> withCoin();
                  case WITHOUT_COIN -> withoutCoin();
               }) {
            System.out.printf("\nCurrent state: %s\n\n",fsm.getState()); // (only for debug)
        }
        return false;
    }

    public boolean withoutCoin() {
        System.out.printf("\nGumball Machine with %d gumballs\n",fsm.getCount());
        switch (PAInput.chooseOption("Machine without coin","Insert coin","Start maintenance","Stop machine")) {
            case 1 -> fsm.insertCoin();
            case 2 -> fsm.startMaintance();
            default -> {
                return false;
            }
        }
        return true;
    }   // ========= next page ======>
```

GumballMachineUI (2/2)

```
    // <=============== previous page =======
    public boolean withCoin() {
        System.out.printf("\nGumball Machine with %d gumballs\n",fsm.getCount());
        switch (PAInput.chooseOption("Machine with a coin","Eject coin","Turns crank","Stop Machine")) {
            case 1 -> fsm.ejectCoin();
            case 2 -> fsm.turnsCrank();
            default -> {
                return false;
            }
        }
        return true;
    }

    public boolean maintenance() {
        switch (PAInput.chooseOption("Maintenance/Machine without gumballs","Refill gumballs","Stop machine")) {
            case 1 -> {
                int count = PAInput.readInt("Number of Gumballs: ");
                fsm.refillGumballs(count);
            }
            default -> {
                return false;
            }
        }
        return true;
    }
}
```