

# Programação Avançada

## 9

### Padrão de Desenho State

Bruno Silva, Patrícia Macedo

# Sumário

- Padrão **State**
  - Enquadramento
  - Motivação
  - Solução Proposta (pelo padrão)
  - Prós e contras
  - Exemplo de Aplicação
    - Exercícios
  - Prós e contras

# Enquadramento

- O padrão **state** é um padrão de desenho *comportamental*.


























## Padrão Comportamental

Estes padrões incidem sobre algoritmos e na atribuição/divisão de responsabilidades entre classes.

# State

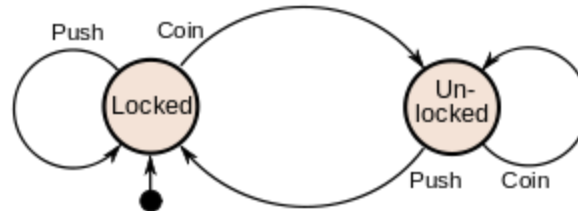
- Proposto em *Design Patterns: Elements of Reusable Object-Oriented Software (1994)*

## THE 23 GANG OF FOUR DESIGN PATTERNS

 Abstract Factory	 Facade	 Proxy
 Adapter	 Factory Method	 Observer
 Bridge	 Flyweight	 Singleton
 Builder	 Interpreter	 State
 Chain of Responsibility	 Iterator	 Strategy
 Command	 Mediator	 Template Method
 Composite	 Memento	 Visitor
 Decorator	 Prototype	

# State

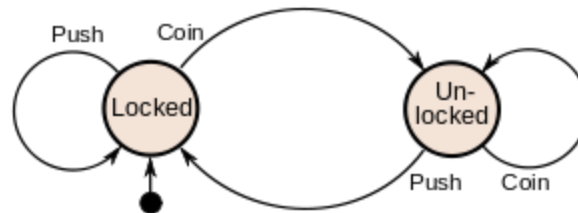
Está diretamente relacionado com *máquinas de estados finitos*.



## Motivação

Permite que um objeto altere o seu comportamento, mediante o estado em que se encontra.

# State



- Existe um número **finito** de *estados* nos quais um *objeto* se pode encontrar;
- Dentro de cada *estado* o objeto comporta-se de forma diferente;
- Certos *inputs* podem fazer o objeto *transitar* de *estado*, ou não;
- As *regras de transição* são também *finitas* e *pre-determinadas*.

# State

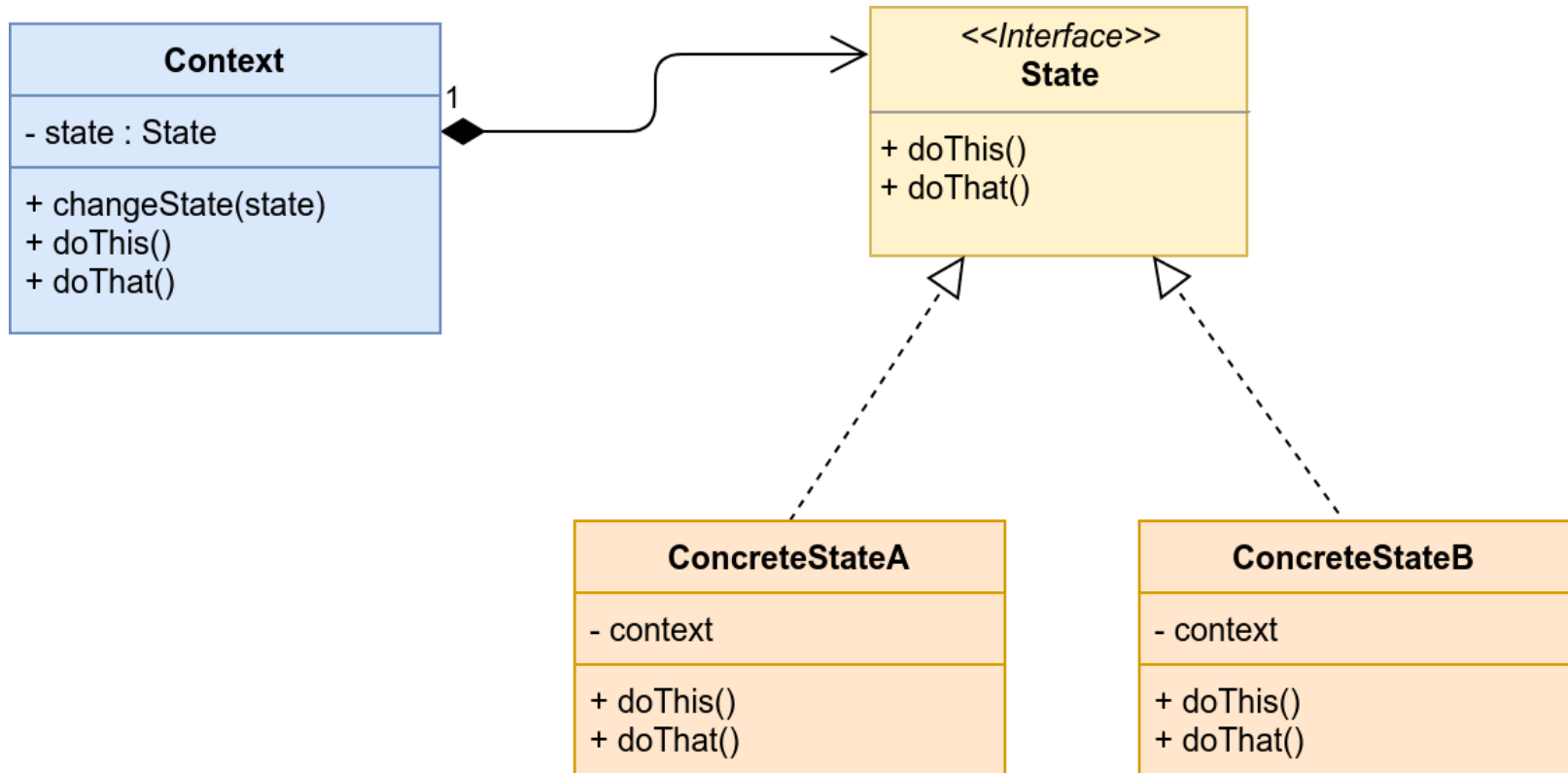
**Permite evitar** estruturas condicionais difíceis de manter em máquina de estados mais complexas. Para o exemplo anterior (simples) teríamos, e.g.,:

```
public class Machine {  
    enum State {LOCKED, UNLOCKED};  
    private State state;  
    public Machine() { state = State.LOCKED; }  
    public void insertCoin() {  
        if(state == State.LOCKED) {  
            //accept coin and change state  
            state = state.UNLOCKED;  
        } else if(state == State.UNLOCKED) {  
            //return coin and maintain state  
        }  
    }  
    //...  
}
```

💡 Se forem adicionados novos *estados* e *regras de transição* será difícil manter as estruturas condicionais.

# State

Diagrama de classes da **solução proposta** pelo padrão:





# State

Participantes e responsabilidades:

- **Context:** Guarda uma referência para um dos estados concretos e delega nele todo o comportamento específico do estado atual. Permite que o seu estado seja alterado.
- **State:** Declara os métodos específicos de um estado. Estes métodos serão "comuns" a todos os estados concretos, variando a sua implementação.
- **Concrete states:** Representam um estado possível e implementam os métodos de *State*, refletindo o comportamento do objeto para esse estado.

# Problema real

O seguinte repositório contém a aplicação do padrão **state** a um *music player*, baseado em linha de comandos.

[https://github.com/brunomnsilva/JavaPatterns\\_State](https://github.com/brunomnsilva/JavaPatterns_State)

---

Podemos interagir com o *music player* através dos seguintes comandos (métodos públicos, são invocados na *CLI*):

-  `play()`
-  `prev()`
-  `next()`
-  `stop()`
-  `status()`

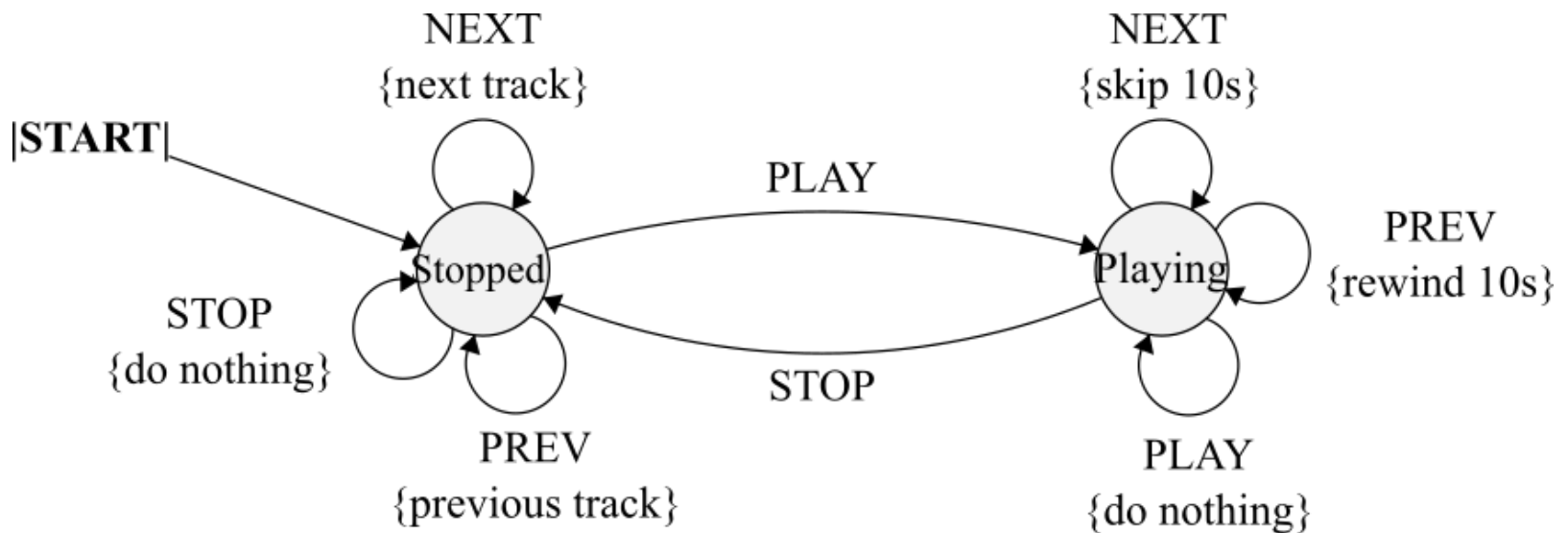
# State

Atente no código da classe `MusicPlayer` :

- Possui um atributo do tipo `MusicPlayerState` (classe abstrata);
- Os estados possíveis atualmente são as concretizações desta classe: `StoppedState` ou `PlayingState` ;
- O comportamento dos métodos `play()` , `prev()` , `next()` , `stop()` e `status()` é "delegado" para o *estado* atual.
- O estado inicial (construtor) é o estado `StoppedState` .

# State

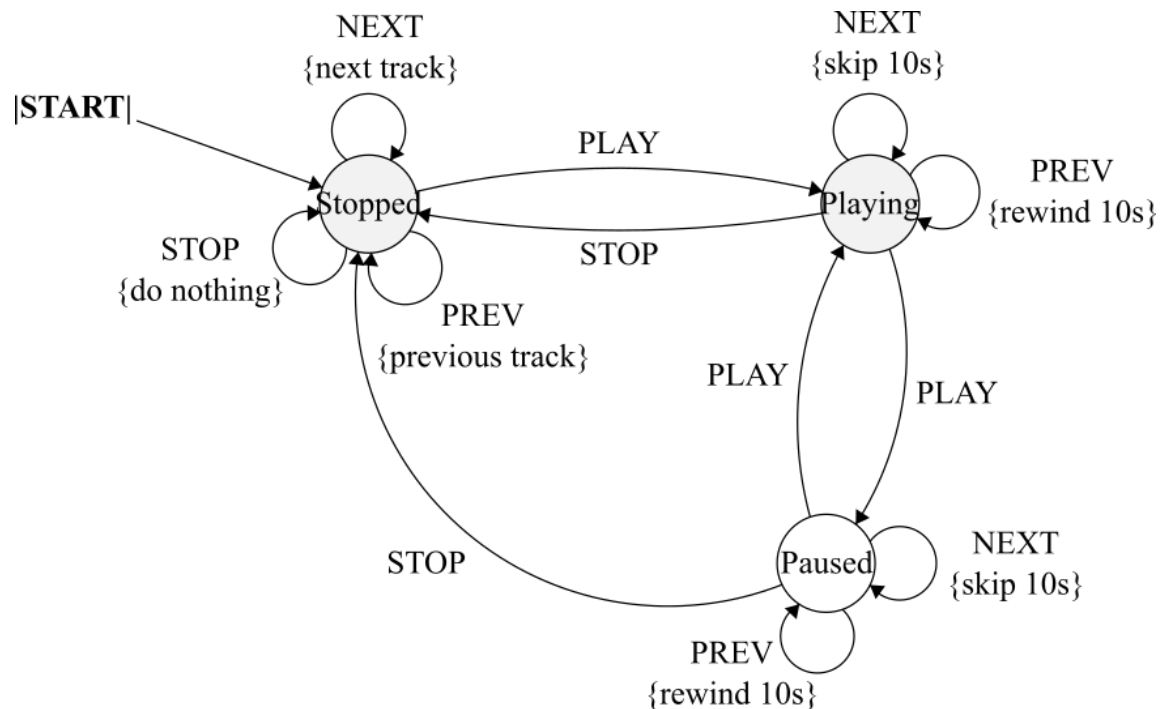
A *máquina de estados finitos* atualmente implementada é a seguinte:



💡 Teste a aplicação verificando que o comportamento do *music player* obedece a este diagrama.

# Exercícios

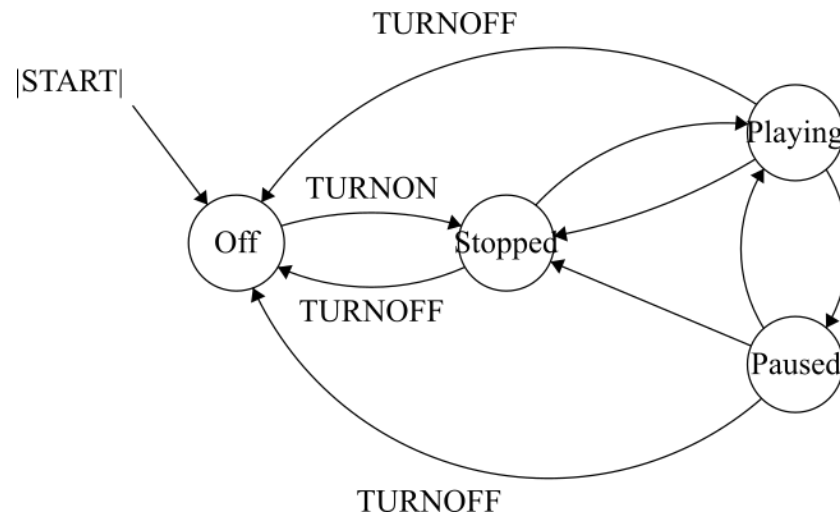
1. Crie um novo *estado* `PausedState` e modifique os existentes por forma a implementar a seguinte *máquina de estados*:



⚠ não serão necessárias alterações à classe `CLI`.

# Exercícios

2. Adicione os métodos `turnOn()` e `turnOff()` a `MusicPlayer`. Estes serão "delegados" para o `MusicPlayerState` atual (requer adicionar métodos à "interface"); Crie um novo *estado* `offState` e modifique os existentes por forma a implementar a seguinte *máquina de estados* (regras de transição no slide seguinte):



⚠ serão necessárias alterações à classe `CLI` para estes comandos.

# Exercícios

## 2. (regras de transição):

- As regras de transição existentes anteriormente mantêm-se (estão omitidas no diagrama);
- O estado inicial é o estado `offState` ;
  - Apenas transita de estado mediante o "comando" *turn on*; ignora todos os outros (mantendo o estado).
  - Ao transitar para `StoppedState` deverá carregar a primeira música que se encontra *playlist*.
- Todos os restantes estados "ignoram" o comando *turn on*.
- Sempre que, de outro estado, se transitar para `StoppedState` , o *playback* deverá terminar e os recursos correspondentes libertados.

## Prós e contras

- ✓ *Single Responsibility Principle*. Organização do código respetivo a cada estado particular em classes separadas.
- ✓ *Open/Closed Principle*. É possível introduzir novos estados com alterações mínimas nas existentes e, por vezes, sem alterar o *contexto*.
- ✓ Simplificar o código do *contexto* eliminando estruturas condicionais difíceis de manter.
- ✗ A aplicação do padrão pode ser um exagero no caso de máquinas de estado muito simples ou que raramente mudam de estado.



## Bibliografia

- *Design Patterns: Elements of Reusable Object-Oriented Software* (1994)
- <https://refactoring.guru/design-patterns/state>