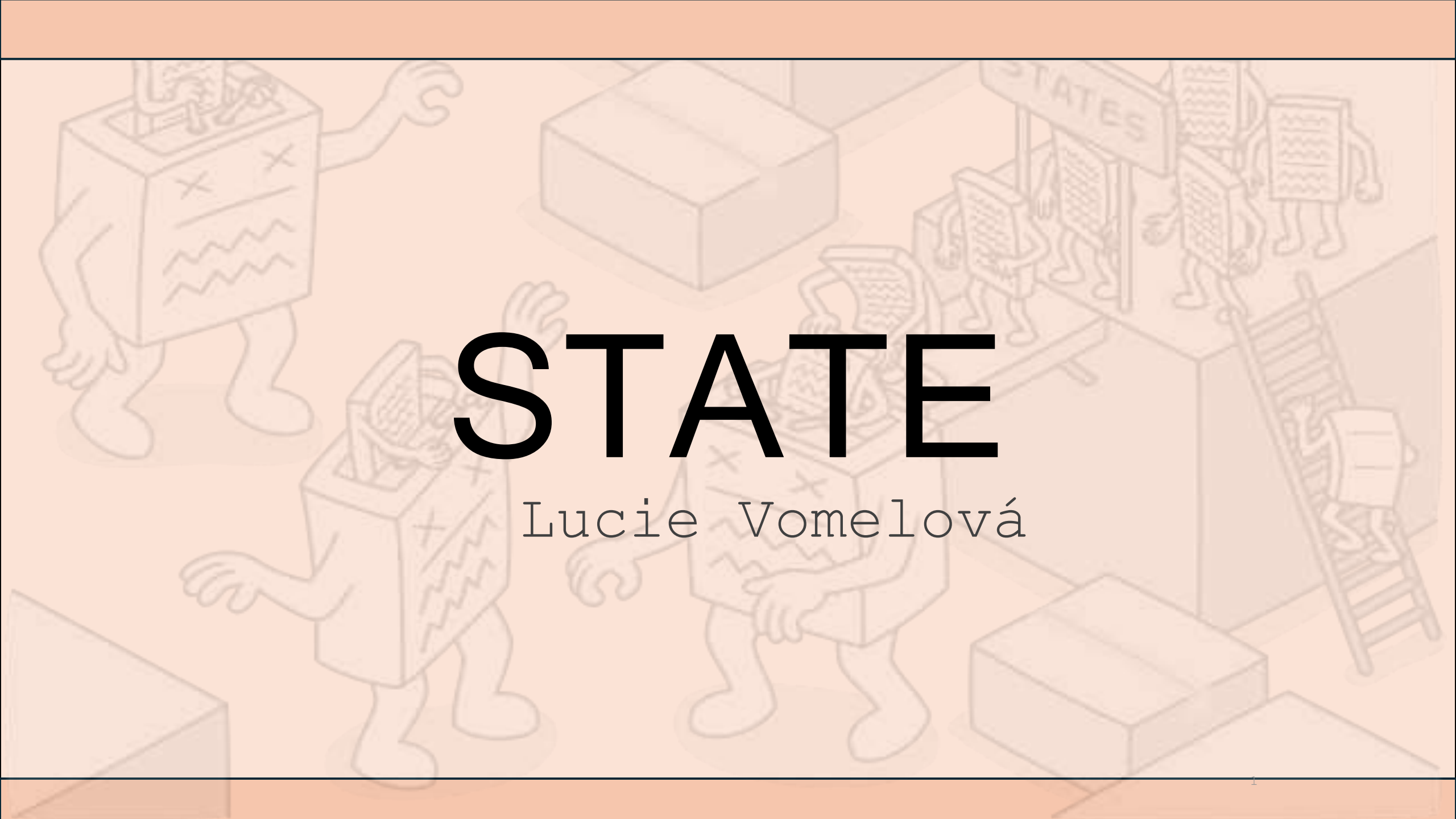
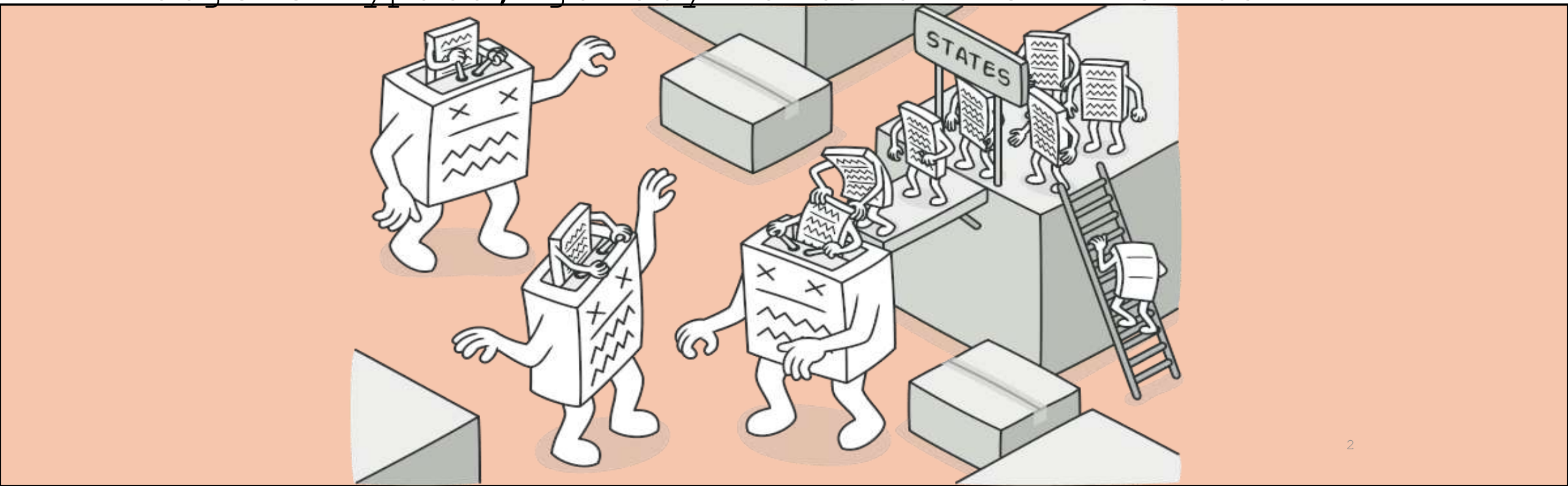


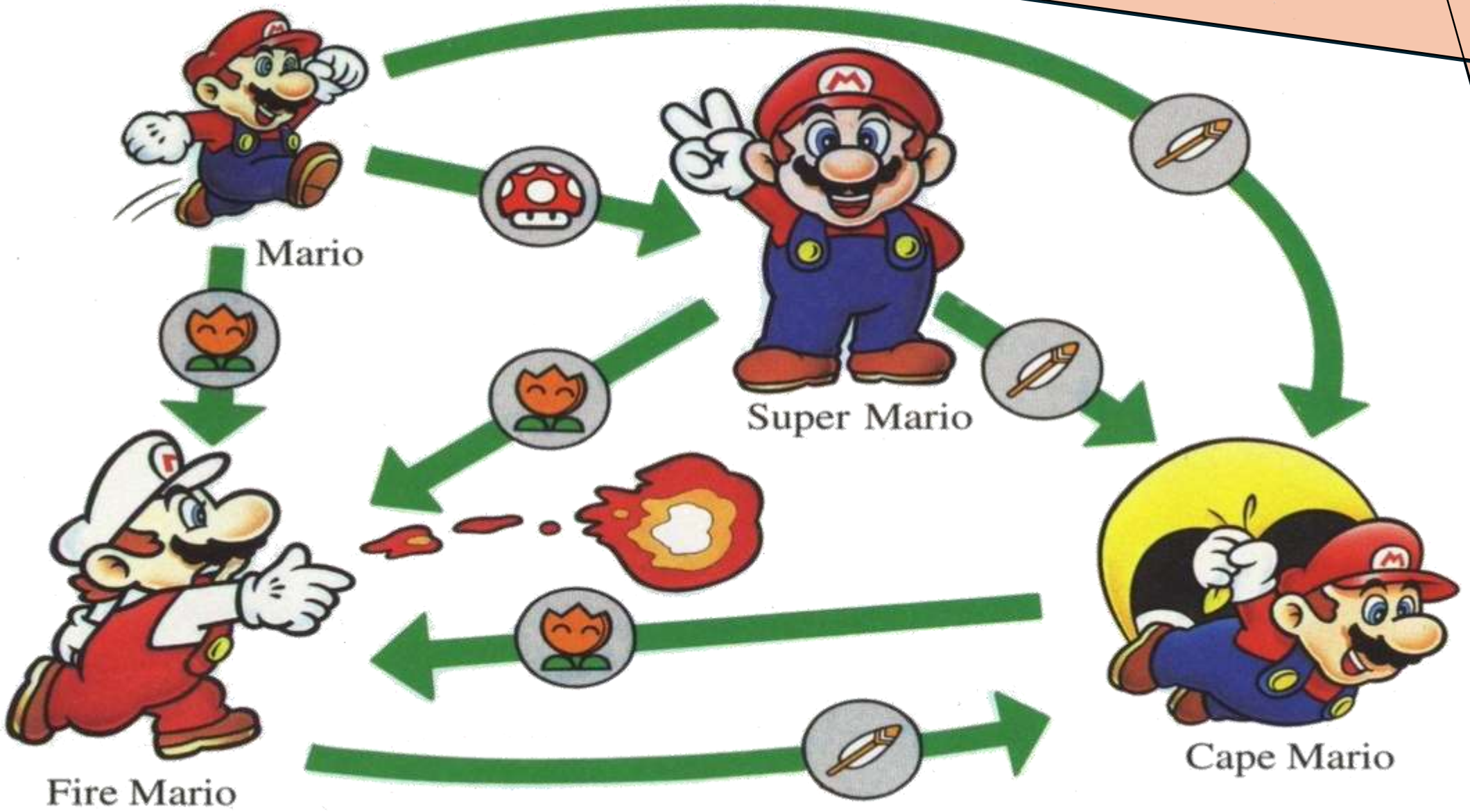
STATE

Lucie Vomelová

STATE

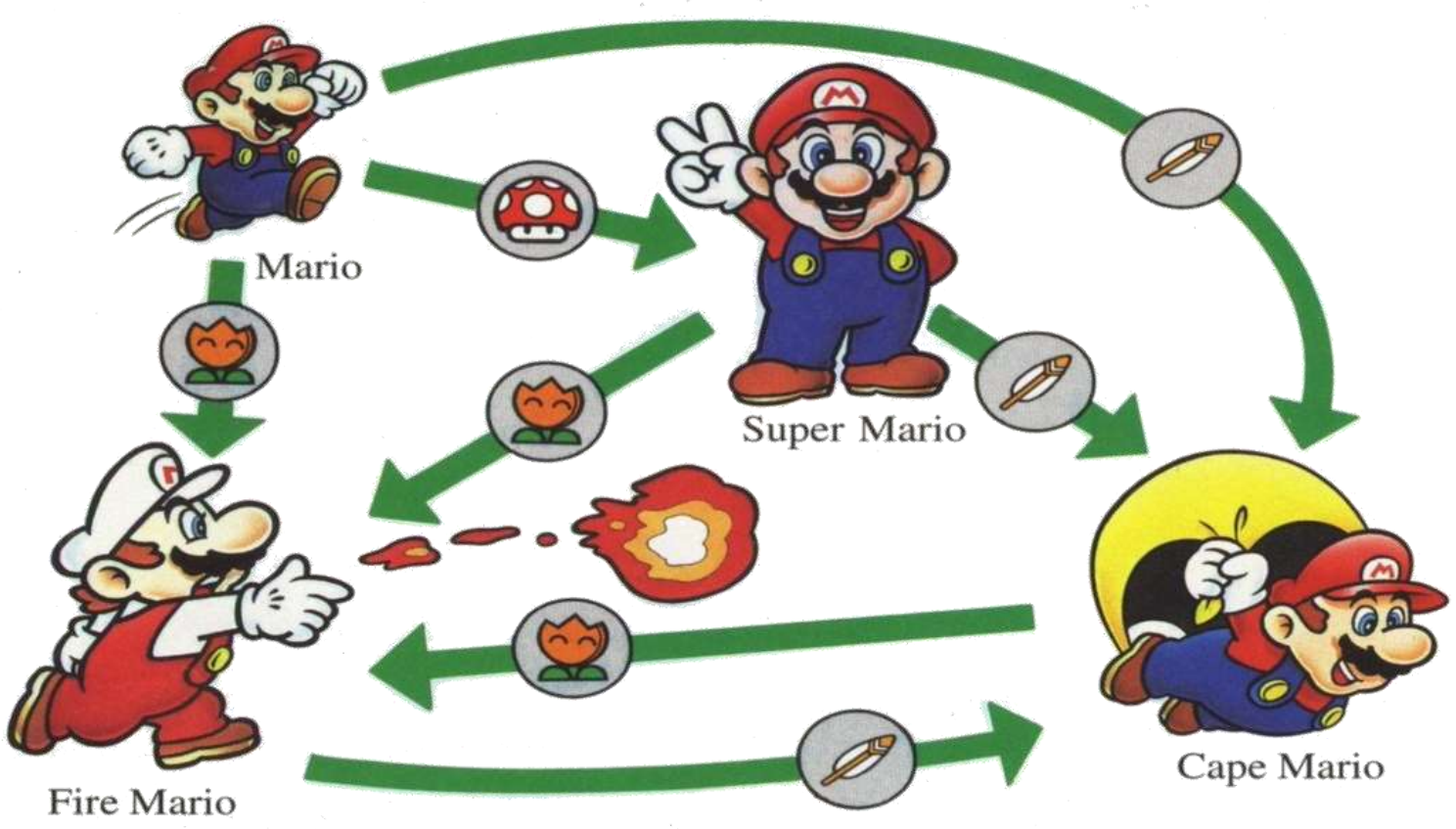
- Umožňuje měnit chování objektu na základě jeho vnitřního stavu
- Objekt vypadá, jakoby za běhu změnil třídu




Mario
Super Mario
Fire Mario
Cape Mario

STATE

- Souvisí s **konečným stavovým automatem**
  - Stavy
  - Přechody



MARIO – NAIVNÍ IMPLEMENTACE

- **Enum stavů**
- **Metoda pro každý event**

MARIO – NAIVNÍ IMPLEMENTACE

- **Enum stavů**

```
public class Mario {
    enum internalState {
        SmallMario,
        SuperMario,
        FireMario,
        CapeMario
    }

private internalState State {get; set;}

public Mario() {
    State = internalState.SmallMario;
}
```

- **Metoda pro každý event**

MARIO – NAIVNÍ IMPLEMENTACE

- **Enum stavů**

```
public class Mario {
    enum internalState {
        SmallMario,
        SuperMario,
        FireMario,
        CapeMario
    }

private internalState State {get; set;}

public Mario() {
    State = internalState.SmallMario;
}
```

- **Metoda pro každý event**

```
public void GotMushroom() {
    if (State == internalState.SmallMario)
        State = internalState.SuperMario;
}
```

```
public void GotFireFlower() {
    State = internalState.FireMario;
}
```

```
public void GotFeather() {
    State = internalState.CapeMario;
}
}
```

ÚČASTNÍCI

Context

State

ConcreteState

ÚČASTNÍCI

Context

- Má referenci na ConcreteState
  - Deleguje na něj state-specific work

State

ConcreteState

ÚČASTNÍCI

Context

- Má referenci na ConcreteState
  - Deleguje na něj state-specific work

State

- Deklaruje state-specific metody
- Pouze metody, které dávají smysl pro všechny stavy

ConcreteState

ÚČASTNÍCI

Context

- Má referenci na ConcreteState
  - Deleguje na něj state-specific work

State

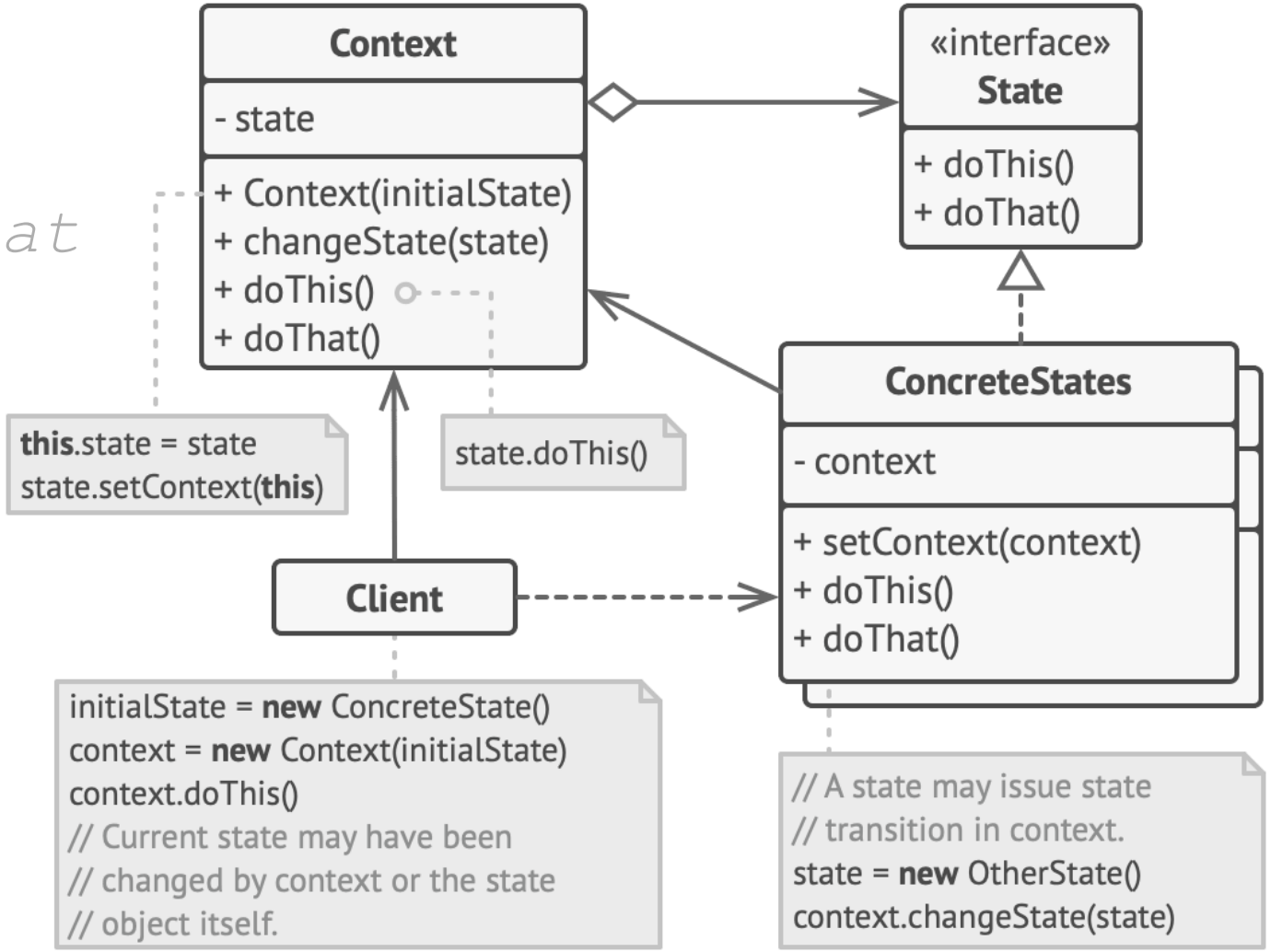
- Deklaruje state-specific metody
- Pouze metody, které dávají smysl pro všechny stavy

ConcreteState

- Vlastní implementace state-specific metod

ÚČASTNÍCI

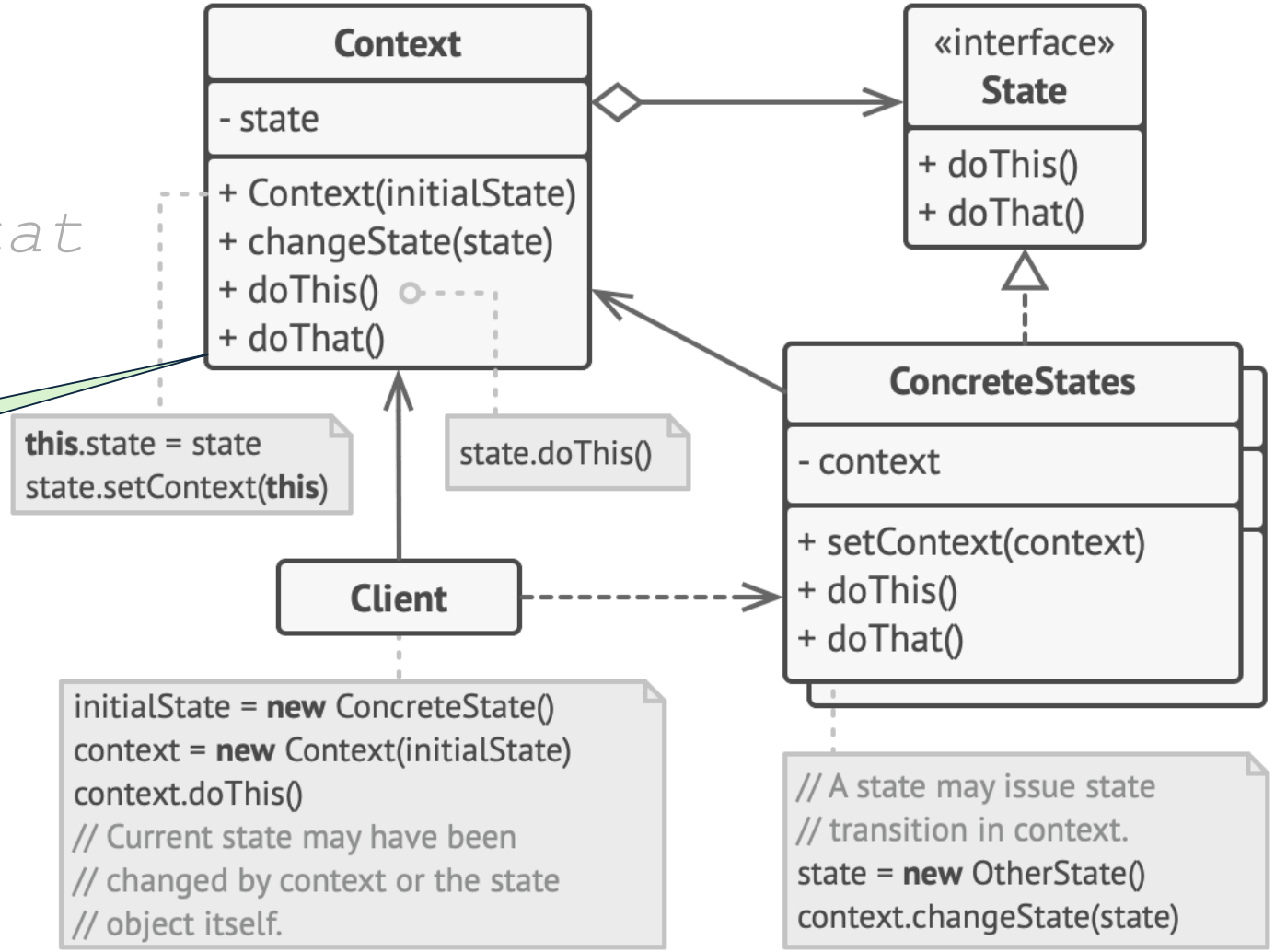
- *Context*
- *State*
- *ConcreteStat*
  e



ÚČASTNÍCI

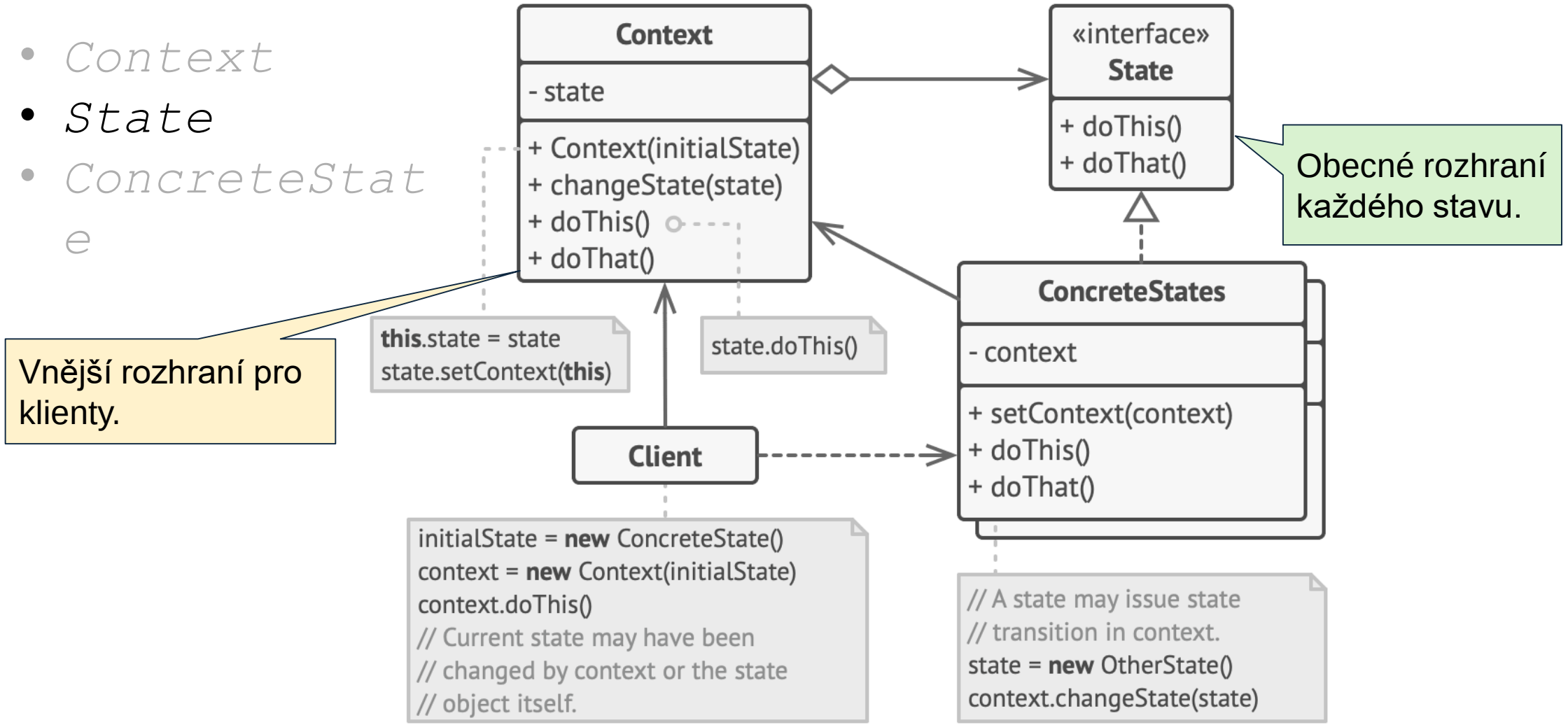
- *Context*
- *State*
- *ConcreteStat e*

Vnější rozhraní pro klienty.



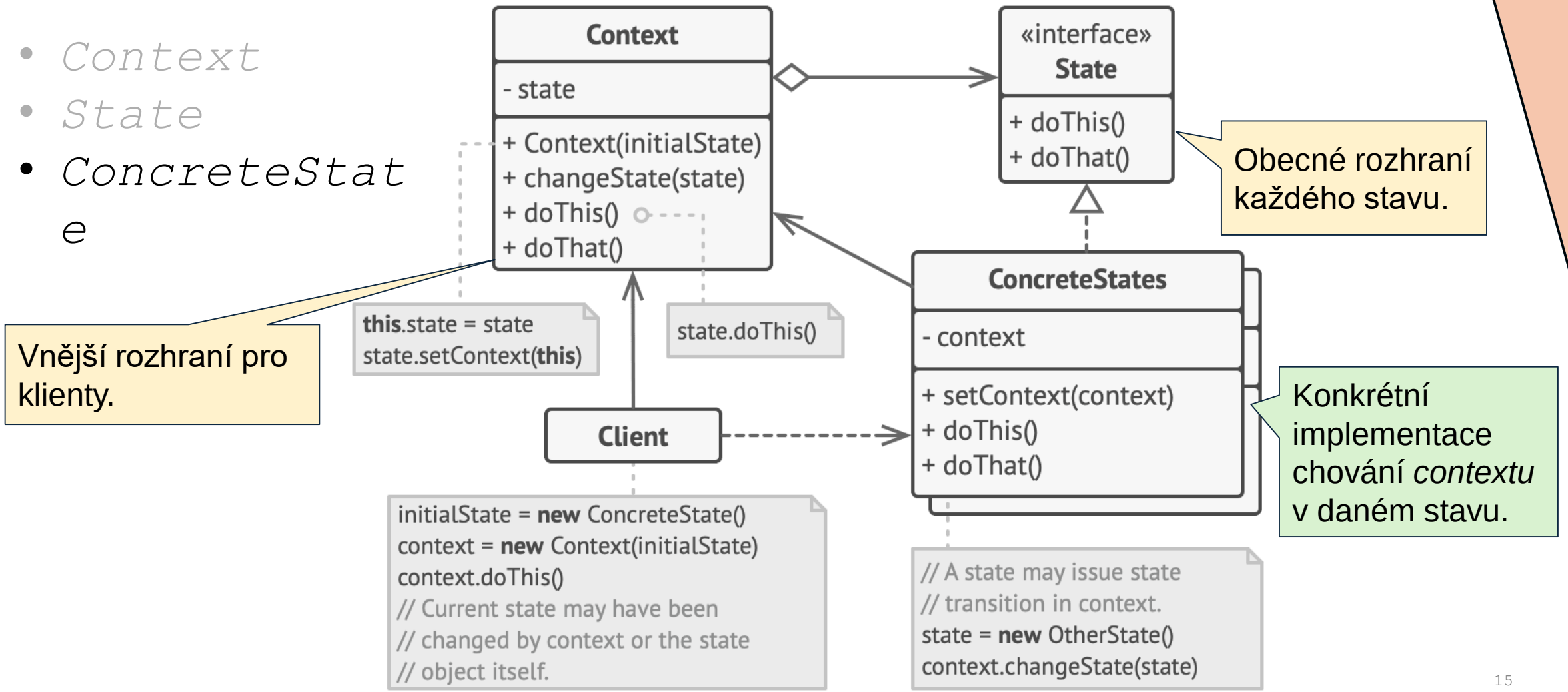
ÚČASTNÍCI

- *Context*
- *State*
- *ConcreteStat e*



ÚČASTNÍCI

- *Context*
- *State*
- *ConcreteStat e*



MARIO – VYUŽITÍ STATE

- **Interface IState**
- **Třída pro každý stav**

MARIO – VYUŽITÍ STATE

- **Interface IState**

```
public interface IState {
    void GotMushroom();
    void GotFireFlower();
    void GotFeather();
};
```

- **Třída pro každý stav**

MARIO – VYUŽITÍ STATE

- **Interface IState**

```
public interface IState {
    void GotMushroom();
    void GotFireFlower();
    void GotFeather();
};
```

- **Třída pro každý stav**

```
public class SmallMario : IState {
    private Mario mario;
    public SmallMario(Mario mario) {
        this.mario = mario;
    }
    public void GotMushroom() {
        mario.state = mario.GetState("superMario");
    }
    public void GotFireFlower() {
        mario.state = mario.GetState("fireMario");
    }
    public void GotFeather() {
        mario.state = mario.GetState("capeMario");
    }
}
```

MARIO – VYUŽITÍ STATE

- **Hlavní třída (Context)**

```
public class Mario {
    public IState state;
    private SmallMario smallMario;
    private SuperMario superMario;
    private FireMario fireMario;
    private CapeMario capeMario;

   public Mario() {
        smallMario = new SmallMario(this);
        superMario = new SuperMario(this);
        fireMario = new FireMario(this);
        capeMario = new CapeMario(this);
        state = smallMario;
    }
```

```
    public void GotMushroom() {
        state.GotMushroom();
    }
```

```
    public void GotFireFlower() {
        state.GotFireFlower();
    }
```

```
    public void GotFeather() {
        state.GotFeather();
    }
}
```

KDY POUŽÍT STATE

- Spousta **if-else** nebo **switch**

KDY POUŽÍT STATE

- Spousta **if-else** nebo **switch**
- Objekt se spoustou stavů a častými přechody

KDY POUŽÍT STATE

- Spousta `if-else` nebo `switch`
- Objekt se spoustou stavů a častými přechody
- Duplikátní kód mezi stavy
  - Jde vyřešit abstraktními base classes
  - Hierarchie stavů

HIERARCHIE STATE TŘÍD

- Pokud nějaké stavy sdílí stejné vlastnosti – duplicitní kód
- Řešení → base classes
- Můžeme vytvořit hierarchii stavů

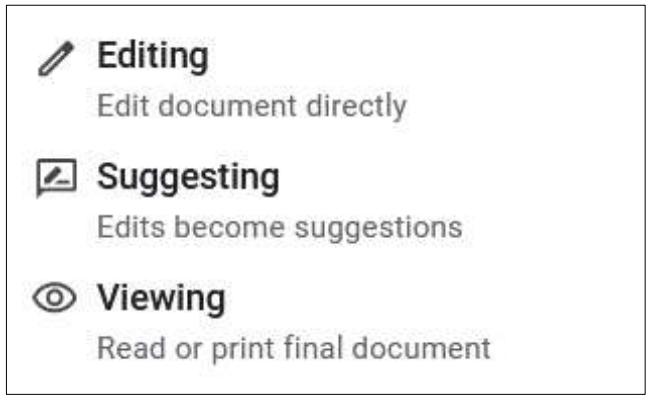
- Výhody:
  - Odstranění duplicitního kódu
  - Hierarchická struktura – přehledná vizualizace

REÁLNÉ POUŽITÍ

- **Editační nástroje**
  - Textové editory
  - Grafické programy

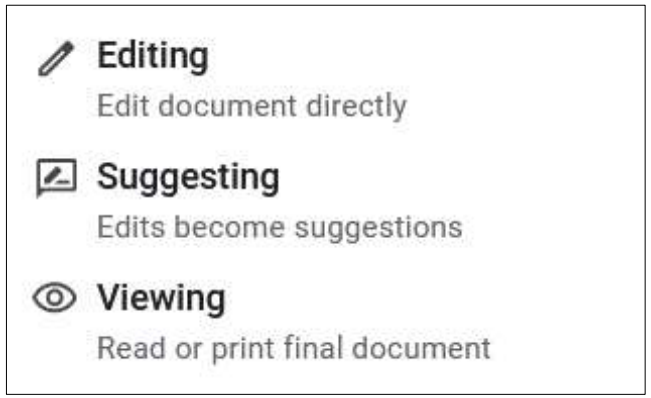
REÁLNÉ POUŽITÍ

- **Editační nástroje**
  - Textové editory
  - Grafické programy



REÁLNÉ POUŽITÍ

- **Editační nástroje**
  - Textové editory
  - Grafické programy

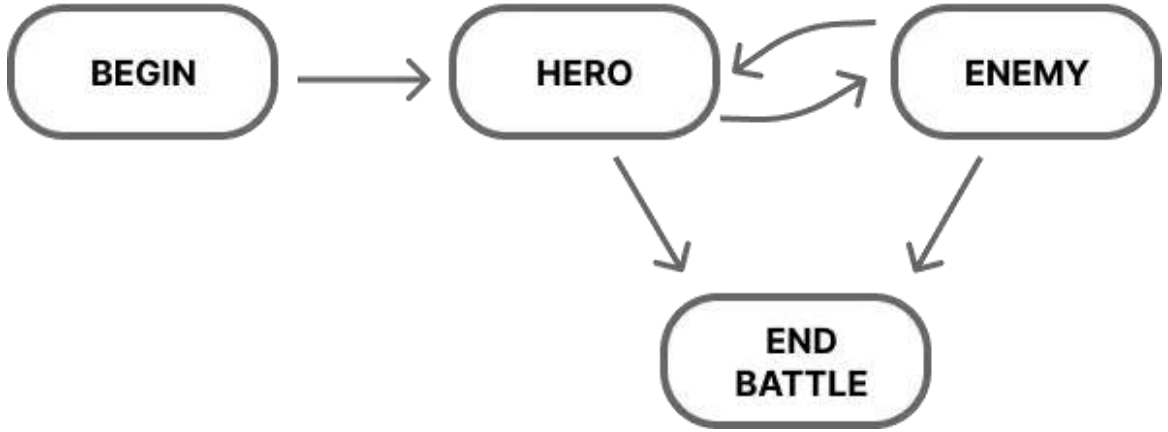


REÁLNÉ POUŽITÍ

- **Editační nástroje**
  - Textové editory
  - Grafické programy
- **Počítačové hry**
  - Alive / Dead
  - Idle / Run / Battle
  - Turn-based combat – Begin / My turn / Enemy turn / End

REÁLNÉ POUŽITÍ

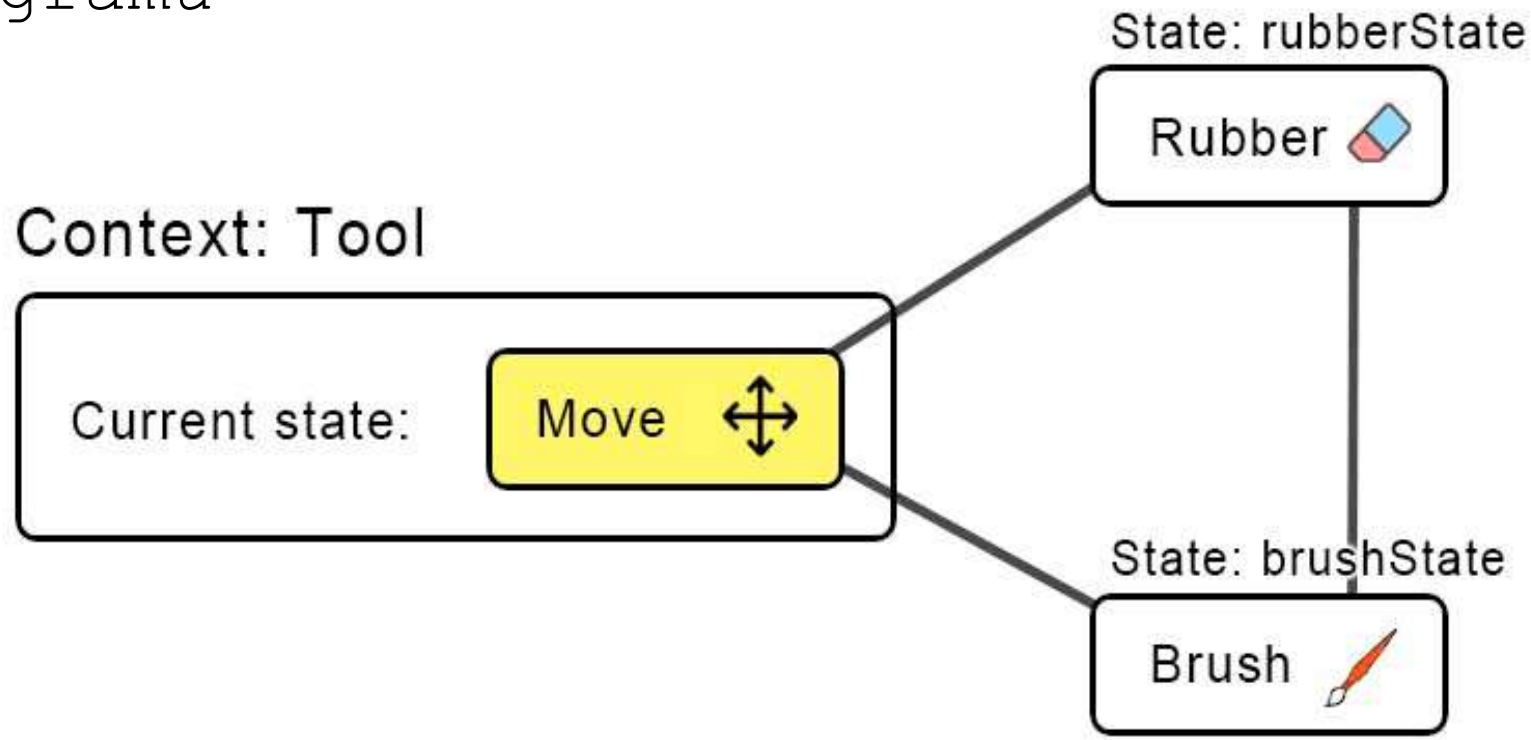
- **Editační nástroje**
  - Textové editory
  - Grafické programy



- **Počítačové hry**
  - Alive / Dead
  - Idle / Run / Battle
  - Turn-based combat – Begin / My turn / Enemy turn / End

PŘÍKLAD – GRAFICKÝ EDITOR

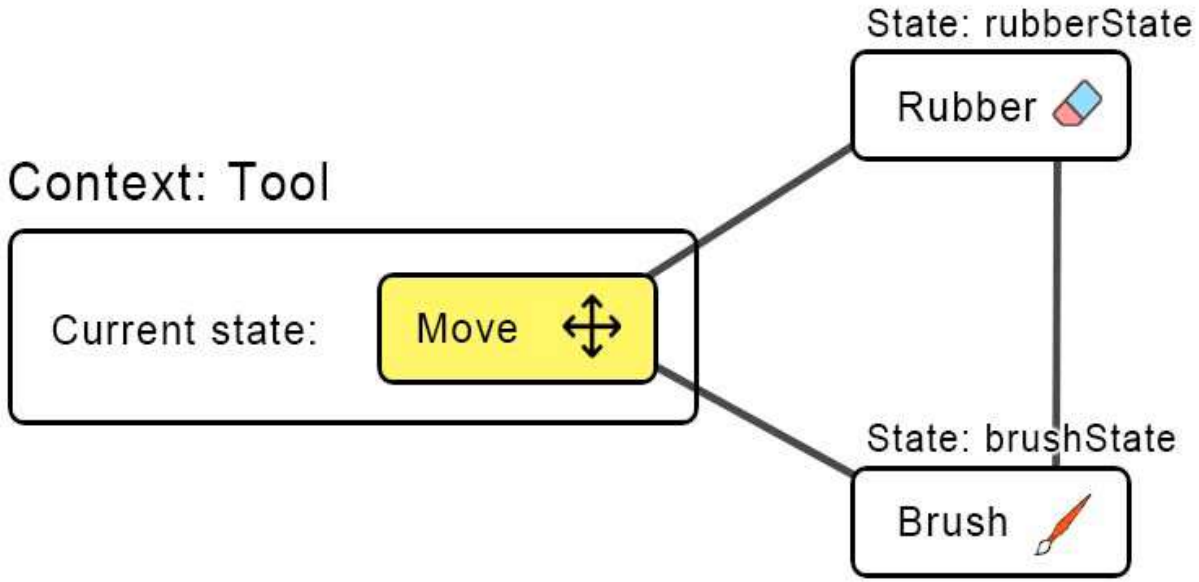
- Chceme měnit chování myši za běhu programu



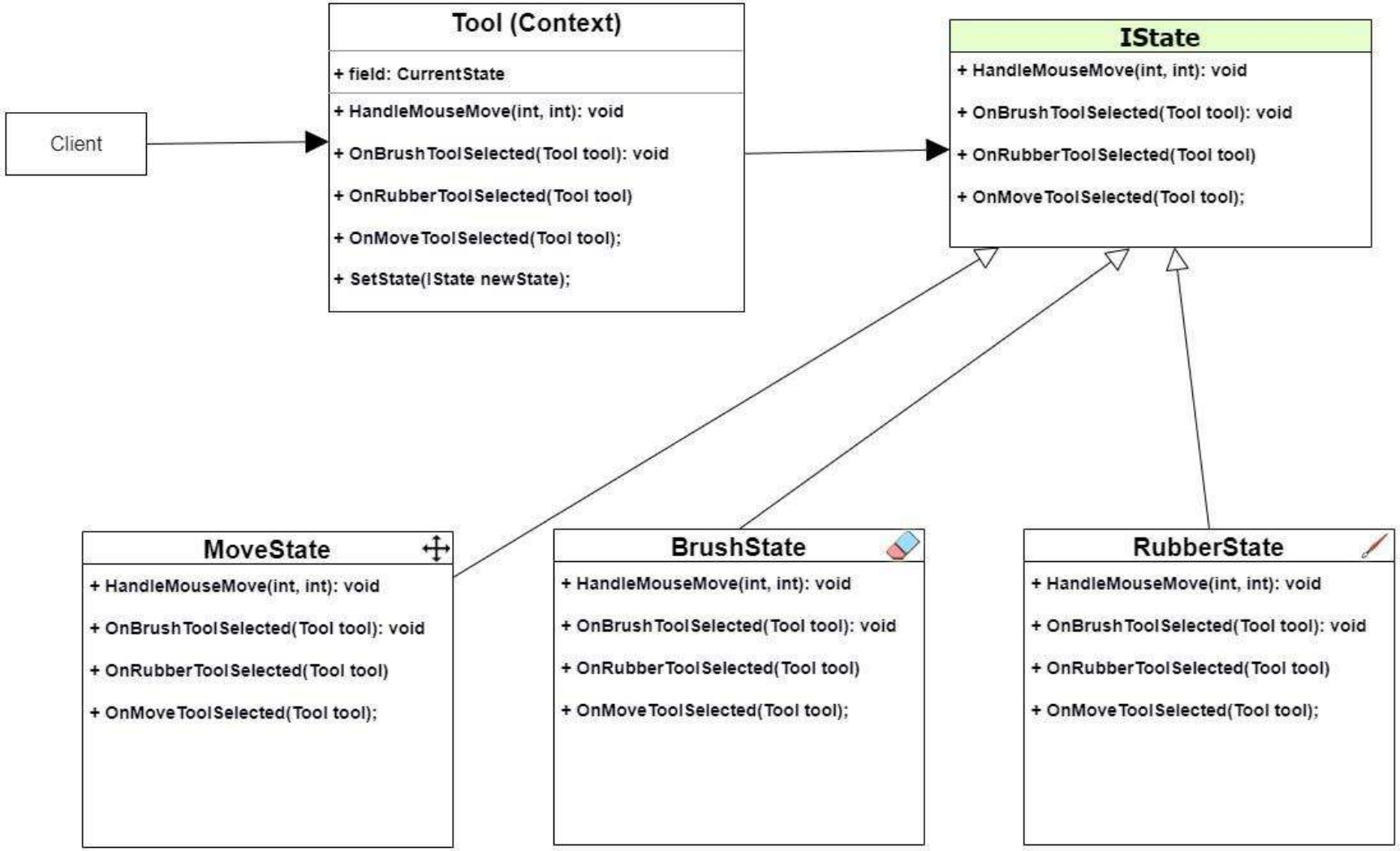
PŘÍKLAD – GRAFICKÝ EDITOR

- Context - obecný nástroj
  - Chování podle současného stavu

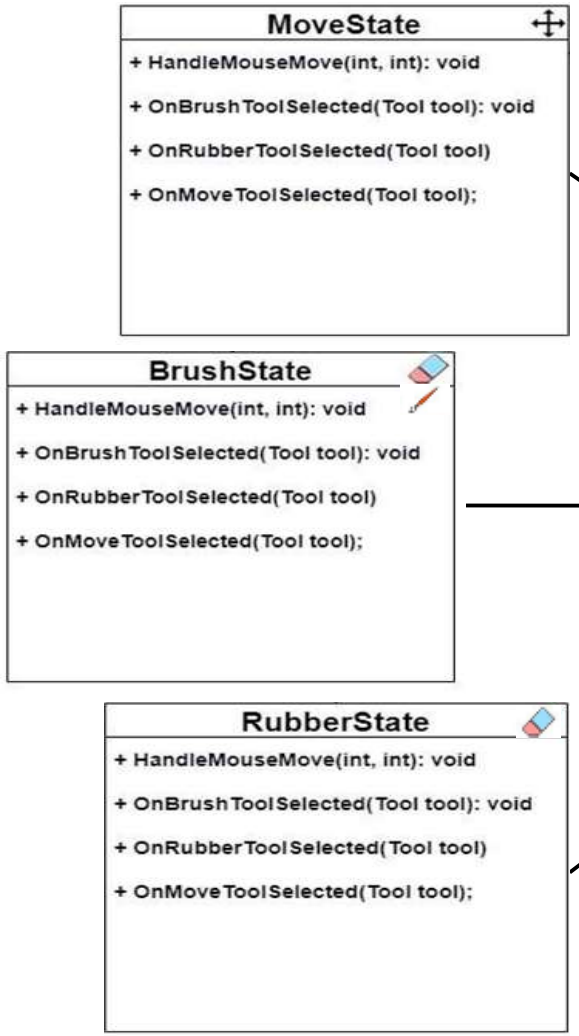
- Stavy:
  - Brush
  - Rubber
  - Move



GRAFICKÝ EDITOR



GRAFICKÝ EDITOR - ISTATE




```
public interface IState
{
    public void HandleMouseMove(int mouseX, int mouseY);

    // switch to brush
    public void OnBrushToolSelected(Tool tool);

    // switch to rubber
    public void OnRubberToolSelected(Tool tool);

    // switch to move tool
    public void OnMoveToolSelected(Tool tool);
}
```

GRAFICKÝ EDITOR – CONTEXT

```
public class Tool
{
    private IState _state;
    public Tool() => SetState(new MoveState()); // default state

    public void SetState(IState newState) {
        _state = newState; // sets new state
    }
    public void MouseMoved(int mouseX, int mouseY) {
        _state.HandleMouseMove(mouseX, mouseY); //on mouse move
    }
    public void OnBrushToolSelected() {
        _state.OnBrushToolSelected(this); //switch to brush
    }
    public void OnRubberToolSelected() {
        _state.OnRubberToolSelected(this); //switch to rubber
    }
    public void OnMoveToolSelected() {
        _state.OnMoveToolSelected(this); // switch to move tool
    }
}
```

GRAFICKÝ EDITOR – CONCRETE STATE

```
public class BrushState : IState
{
    public void HandleMouseMove(int mouseX, int mouseY) {
        // implementation of what to do on move operation
    }

    public void OnBrushToolSelected(Tool tool) {
        return; // already in brush state
    }

    public void OnMoveToolSelected(Tool tool) {
        tool.SetState(new MoveState());
    }

    public void OnRubberToolSelected(Tool tool) {
        tool.SetState(new RubberState());
    }
}
```

GRAFICKÝ EDITOR – NOVÝ TOOL

- *SelectState*

```
public class SelectState : Istate
```

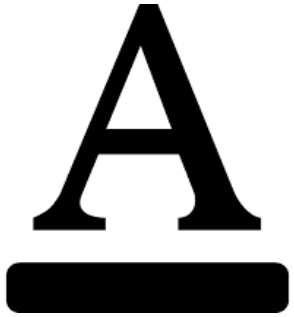
- Upravení IState:

```
public interface IState
{
    public void HandleMouseMove(int mouseX, int mouseY);
    public void OnBrushToolSelected(Tool tool);
    public void OnRubberToolSelected(Tool tool);
    public void OnMoveToolSelected(Tool tool);

    // add select tool
    public void OnSelectToolSelected(Tool tool);
}
```

GRAFICKÝ EDITOR – NOVÝ TOOL

- Přidání nových toolů je snadné



PŘECHOD MEZI STAVY

Kdo mění stavy? - Context nebo ConcreteState

- **Context**

- **ConcreteState**

PŘECHOD MEZI STAVY

Kdo mění stavy? - Context nebo ConcreteState

- **Context**
  - Musí znát logiku a podmínky změn
- **ConcreteState**

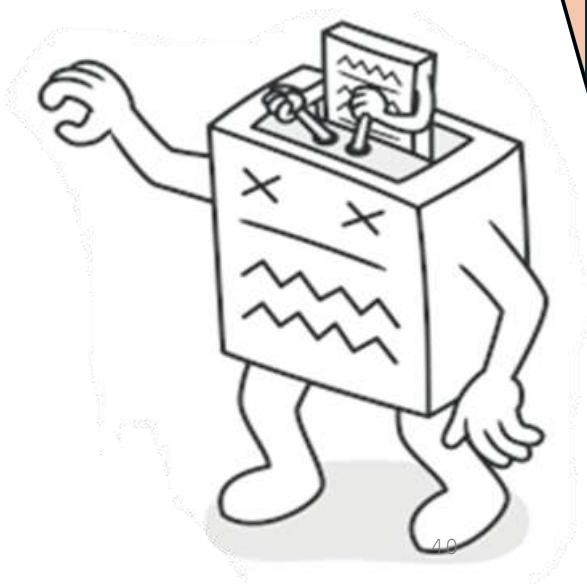
PŘECHOD MEZI STAVY

Kdo mění stavy? - Context nebo ConcreteState

- **Context**
  - Musí znát logiku a podmínky změn
- **ConcreteState**
  - Ví, do jakých stavů se může změnit
  - Flexibilnější, přehlednější

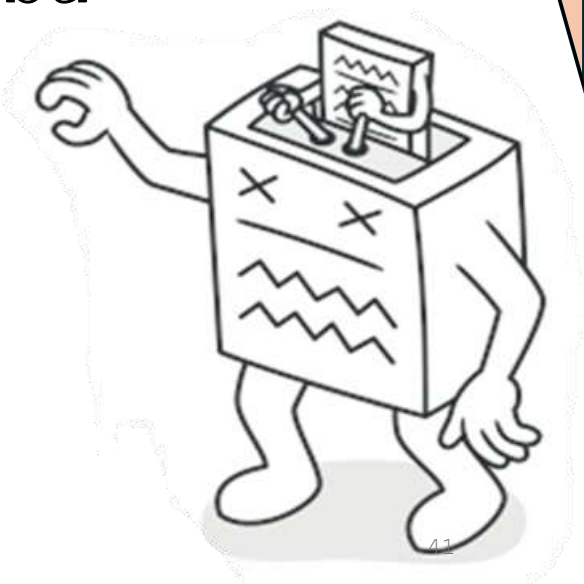
VÝHODY

- ✔ Single Responsibility Principle (**S**OLID)
  - Každý stav má svoji třídu



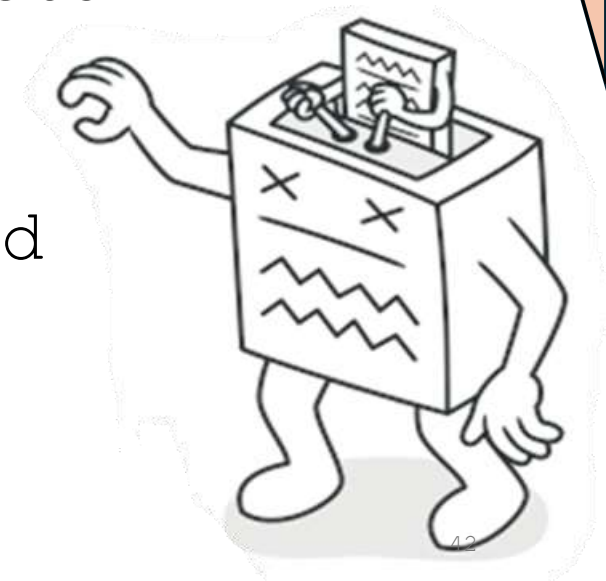
VÝHODY

- ✔ Single Responsibility Principle (**S**OLID)
  - Každý stav má svoji třídu
- ✔ Open/Closed Principle (S**O**LID)
  - Pro přidání nových stavů není potřeba měnit už existující



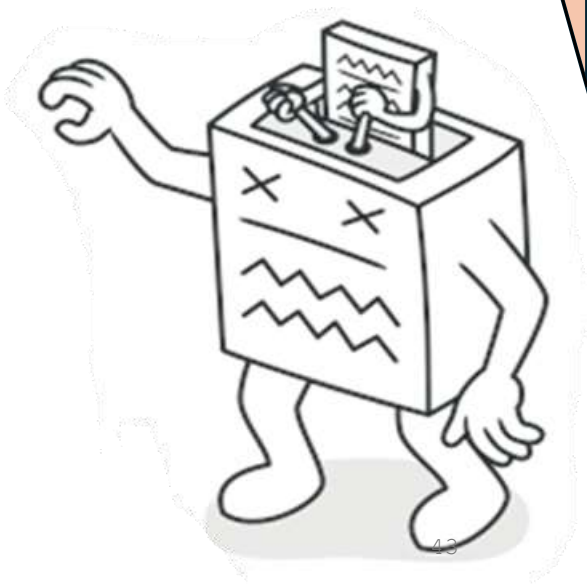
VÝHODY

- ✓ Single Responsibility Principle (**S**OLID)
  - Každý stav má svoji třídu
- ✓ Open/Closed Principle (S**O**LID)
  - Pro přidání nových stavů není potřeba měnit už existující
- ✓ Context – jednodušší a přehlednější kód



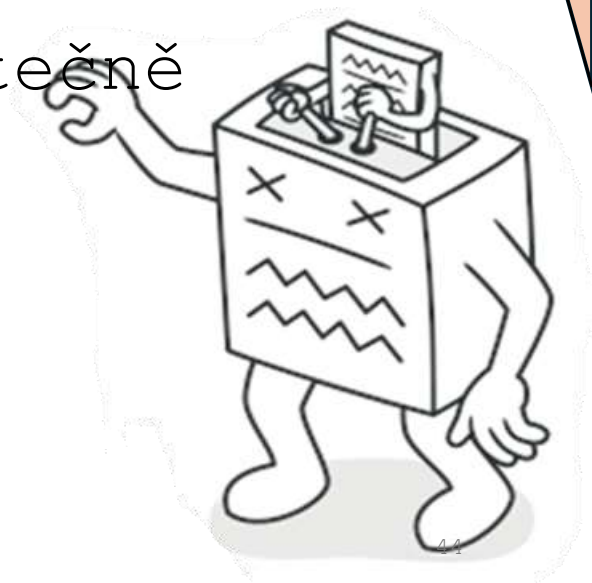
NEVÝHODY

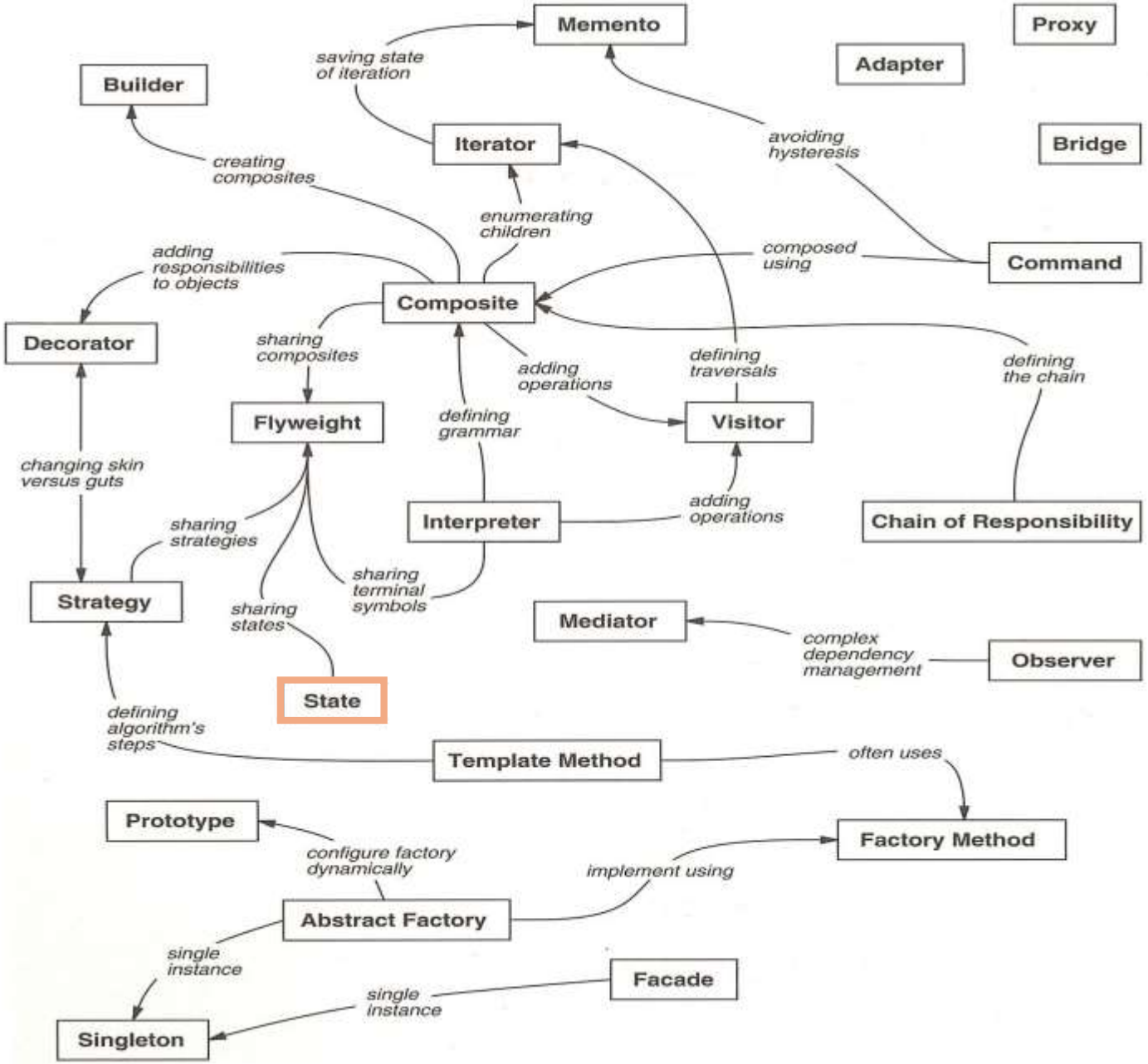
- ✕ Málo stavů → overkill
  - Třída pro každý stav, i když by stačil jeden switch



NEVÝHODY

- ✕ Málo stavů → overkill
  - Třída pro každý stav, i když by stačil jeden switch
- ✕ Výkon
  - Vytváření objektů může zabírat zbytečně moc času




Memento
Proxy
saving state of iteration
Builder
Adapter
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
Strategy
sharing terminal symbols
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

SOUVISEJÍCÍ VZORY

- **Singleton pattern**
- **Flyweight pattern**
- **Strategy pattern**

SOUVISEJÍCÍ VZORY

- **Singleton pattern**
  - Stavy mohou být singleton
- **Flyweight pattern**
- **Strategy pattern**

SOUVISEJÍCÍ VZORY

- **Singleton pattern**
  - Stavy mohou být singleton
- **Flyweight pattern**
  - Stavy
- **Strategy pattern**

SOUVISEJÍCÍ VZORY

- **Singleton pattern**
  - Stavy mohou být singleton
- **Flyweight pattern**
  - Stavy
- **Strategy pattern**
  - U obou context deleguje práci na pomocné objekty
  - Strategy – objekty úplně nezávislé (neví o sobě)
  - State – nestanovuje, jaké mají být závislosti mezi stavy

Děkuji za pozornost

Lucie Vomelová