Das State-Pattern

NACH ROBERT NYSTROMS „GAME PROGRAMMING PATTERNS“

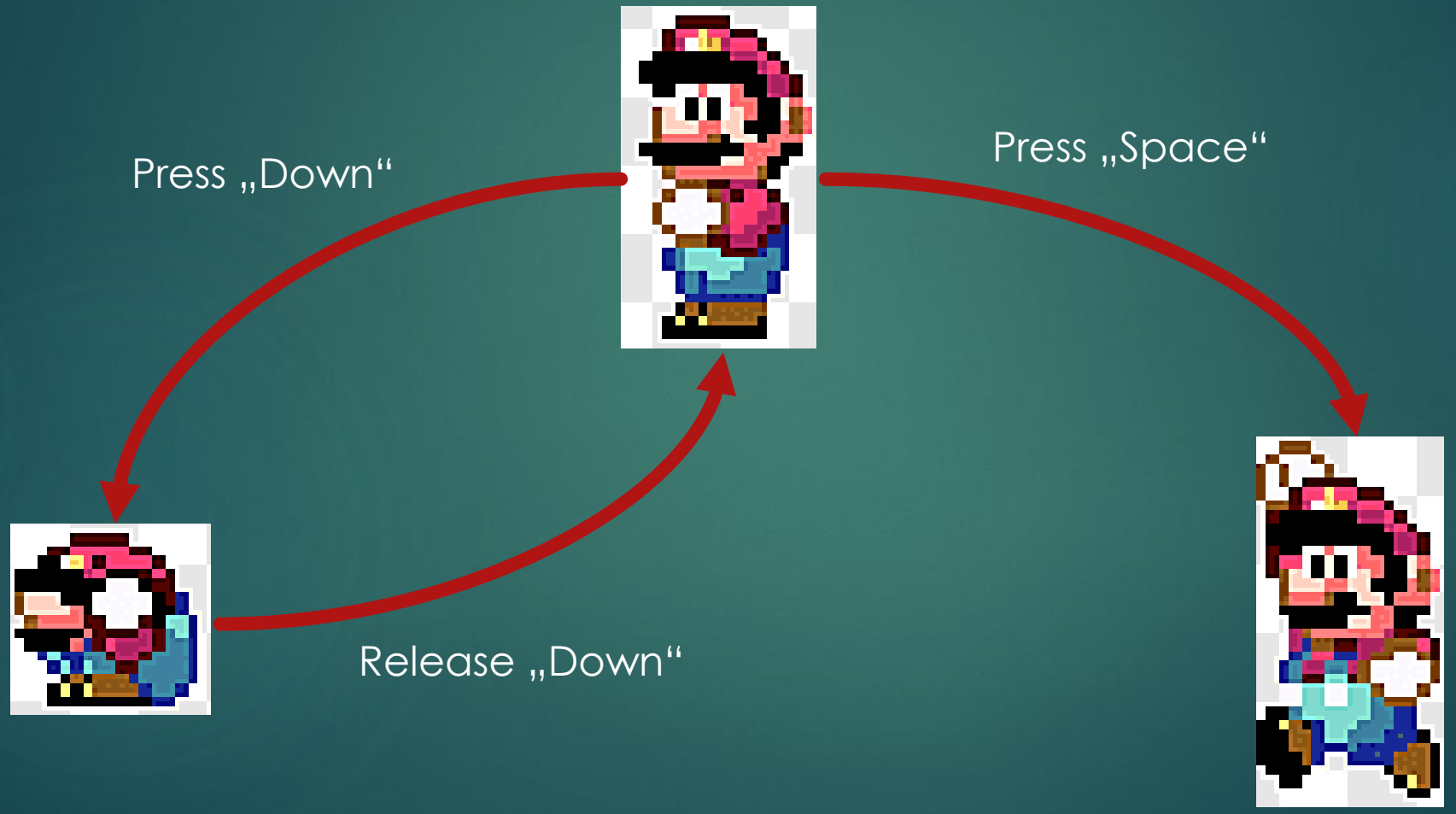
MODUL: PROGRAMMIERUNG II
PRÜFER: CHRISTOPH HAHN
ANDREAS JÄGER

MAXIMILIAN RÖCK
MATRIKELNR. 11016849
13.09.2022

Die Grundidee des State-Patterns

- Ein Objekt soll sein Verhalten ändern können, wenn sich sein interner Zustand ändert
- Das Hinzufügen von neuen Zuständen soll dabei das Verhalten existierender Zustände **nicht** beeinflussen.
  - Zustandsspezifisches Verhalten wird vom Quellcode anderer Zustände unabhängig definiert.

Die Grundidee des State-Patterns



Negativbeispiel:


```
class BadCharacter
{
    // Attributes
public:
    BadState state; // Enum

    int duckTimer;
    std::string debugMessage; // For visualisation
```

```
void BadCharacter::handleInput()
{
    switch (this->state)
    {
        case STATE_STANDING:
            if (IsKeyPressed( key: KEY_SPACE))
            {
                this->state = STATE_JUMPING;
                this->debugMessage = "This character is jumping!";
                this->duckTimer = 0;

                // Adjust player velocity
                // Change texture
                // etc.
            }
            else if (IsKeyPressed( key: KEY_DOWN))
            {
                this->state = STATE_DUCKING;
                this->debugMessage = "This character is ducking!";

                // Change texture
                // etc.
            }
            break;
        case STATE_DUCKING:
            if (IsKeyReleased( key: KEY_DOWN))
            {
                state = STATE_STANDING;
                this->debugMessage = "This character is standing!";

                // Change texture
                // etc.
            }
            break;
            // ...
```

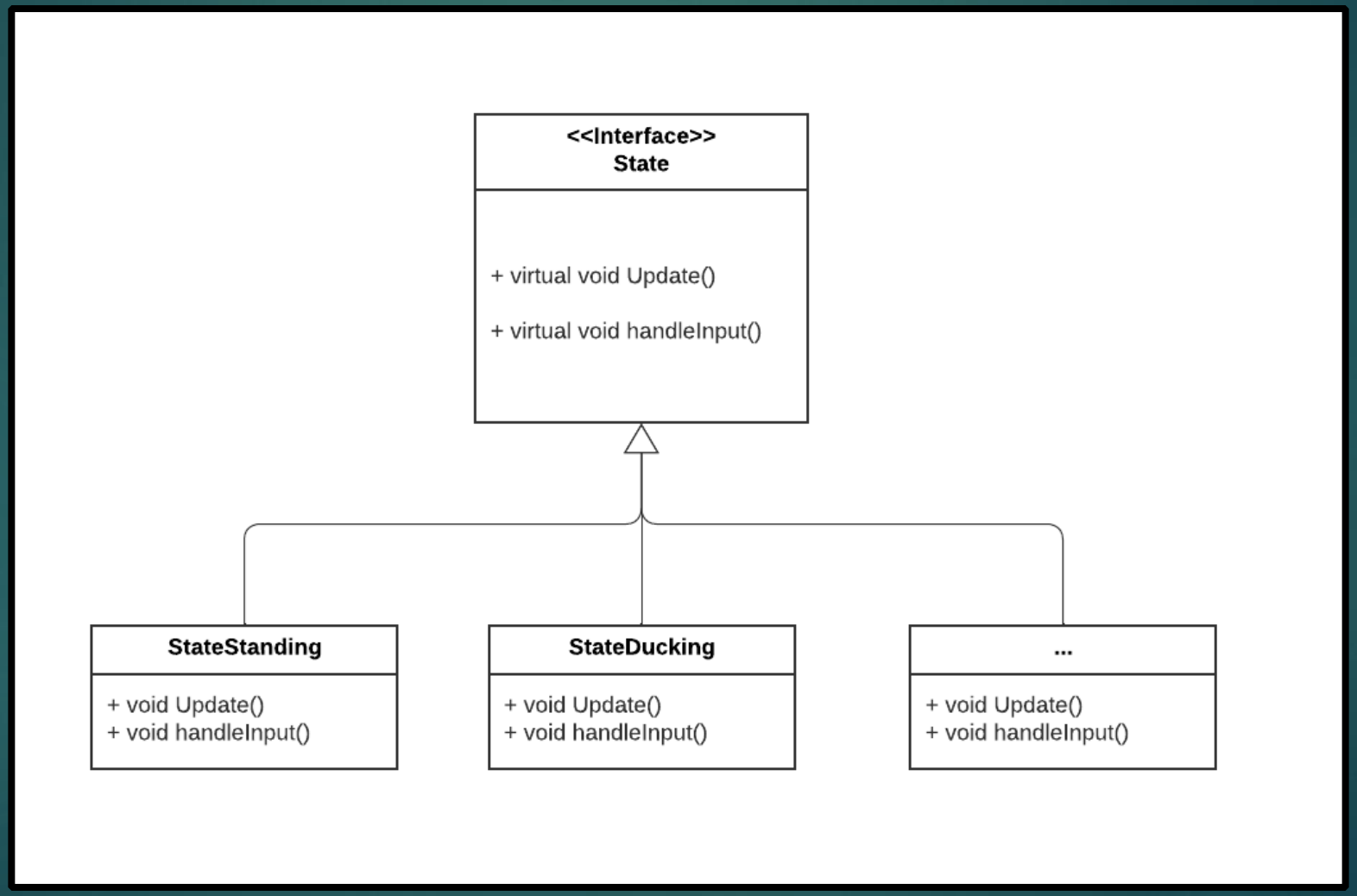
Negativbeispiel:


```
void BadCharacter::Update()
{
    switch (this->state)
    {
        case STATE_DUCKING:
            // Counts how many frames the player has been ducking for
            // (stupid but it's for demonstration purposes)
            this->duckTimer++;

            TraceLog( logLevel: LOG_INFO,  text: std::to_string( val: duckTimer).c_str());

            break;

            // Here can go more cases that define state behaviour
    }
}
```

Nachteile der Switch-Enum-Methode

- Riesige Switch-Anweisungen
- Unübersichtlichkeit
- Aufblähen der Character-Klasse
- Haufenweise Codedopplung
- Schlecht erweiterbar

Die Lösung:


<<Interface>>
State
+ virtual void Update()
+ virtual void handleInput()
StateStanding
+ void Update()
+ void handleInput()
StateDucking
+ void Update()
+ void handleInput()
...
+ void Update()
+ void handleInput()

Die Lösung:


```
class Character
{
    // Attributes
public:
    std::string debugMessage; // For visualisation

    State* currentState;

    // Methods
public:
    Character();

    void Update();

    void Draw();

    void handleInput();
};
```

```
void Character::Update()
{
    this->currentState->Update( &: *this);
}
```

Wo werden State-Objekte instanziiert?

Statische Zustände

- Ein einziges Zustandsobjekt wird erzeugt
- Sinnvoll für simplere Zustände, die z.B. nur Werte verändern
- Können nach Belieben an vielen Stellen im Code erzeugt werden

```
class State
{
public:
    static StateStanding standing;
    static StateDucking ducking;
    static StateJumping jumping;
    // ...
```

Instanziierte Zustände

- Objekte werden nach Bedarf erzeugt und wieder gelöscht
- Sinnvoll für komplexere Zustände, die z.B. von mehreren Charakteren gleichzeitig benutzt werden sollen

Instanziierte Zustände


```
State* StateStanding::handleInput(Character& character)
{
    if (IsKeyPressed( key: KEY_SPACE))
    {
        character.debugMessage = "This character is jumping!";

        // Adjust player velocity
        // Change texture (jumping)
        // etc.
        return new StateJumping;
    }
    else if (IsKeyPressed( key: KEY_DOWN))
    {
        character.debugMessage = "This character is ducking!";

        // Adjust player velocity
        // Change texture (ducking)
        // etc.
        return new StateDucking;
    }
    else
    {
        return NULL;
    }
```

```
void Character::handleInput()
{
    State* state = this->currentState->handleInput( &: *this);

    // For instantiated states
    if (state != NULL)
    {
        delete this->currentState;
        this->currentState = state;
```

Enter-Aktionen


```
void StateDucking::enter(Character &character)
{
    character.debugMessage = "This character is ducking!";
    this->duckTimer = 0;

    // Change texture (Ducking)
    // Adjust hitbox
    // etc.
}
```

```
void Character::handleInput()
{
    State* state = this->currentState->handleInput( & *this);

    // For instantiated states
    if (state != NULL)
    {
        delete this->currentState;
        this->currentState = state;


        this->currentState->enter( & *this);
    }
}
```

Vorteile des State-Patterns

- Riesige Switch-Anweisungen werden vermieden
- Bessere Übersichtlichkeit des Programms
  - Die Character-Klasse bleibt schlank
  - Alle relevanten Daten und Verhalten eines Zustands befinden sich in seiner Klasse
- Neue Zustände können unkompliziert hinzugefügt werden, indem neue Klassen definiert werden
- Codedopplung wird reduziert