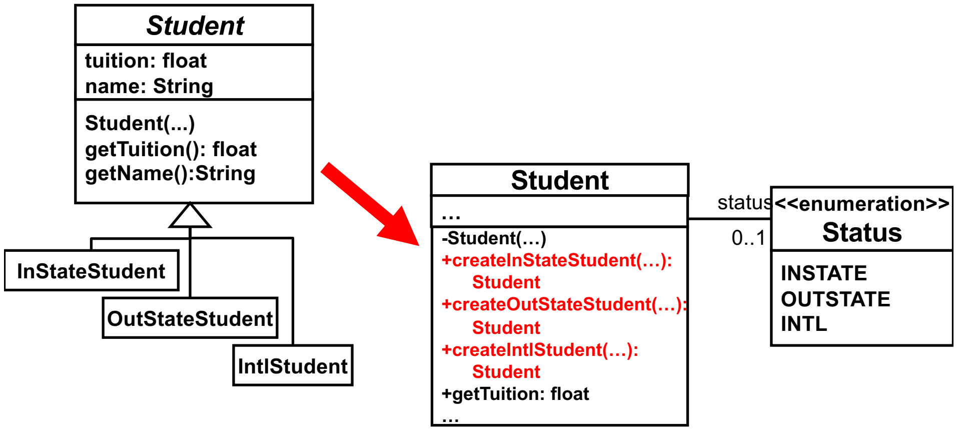
State **Design Pattern**

State **Design Pattern**

- Intent
  - Allows an object to change its behavior according to its state.
    - Allows an object to perform *state-dependent behaviors*.

An Example



Student

tuition: float
name: String

Student(...)
getTuition(): float
getName():String

InStateStudent

OutStateStudent

IntlStudent

Student

...

-Student(…)
+createInStateStudent(…): Student
+createOutStateStudent(…): Student
+createIntlStudent(…): Student
+getTuition: float
...

status

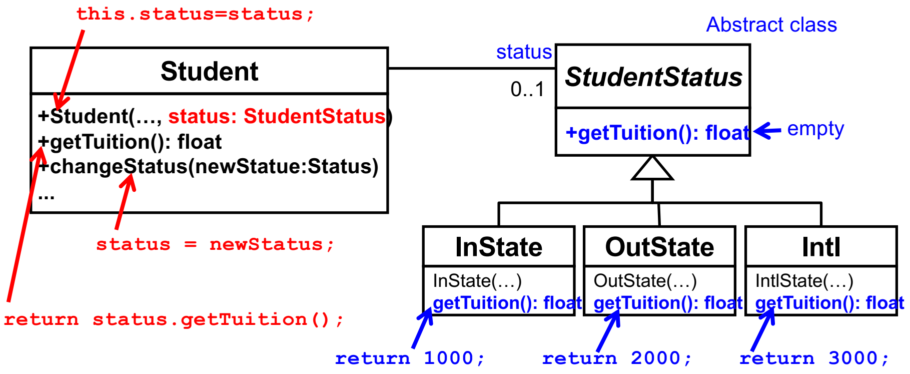
0..1

<<enumeration>>
Status

INSTATE
OUTSTATE
INTL

- c.f. previous lecture notes
- Allows each student to change his/her status dynamically
- Needs a conditional in getTuition()
  - Can eliminate the conditional with *State*.

Design Improvement with ***State***



this.status=status;

Student

+Student(…, status: StudentStatus)
+getTuition(): float
+changeStatus(newStatue:Status)
...

status = newStatus;

return status.getTuition();

status

0..1

Abstract class

StudentStatus

+getTuition(): float

empty

InState

InState(…)
getTuition(): float

return 1000;

OutState

OutState(…)
getTuition(): float

return 2000;

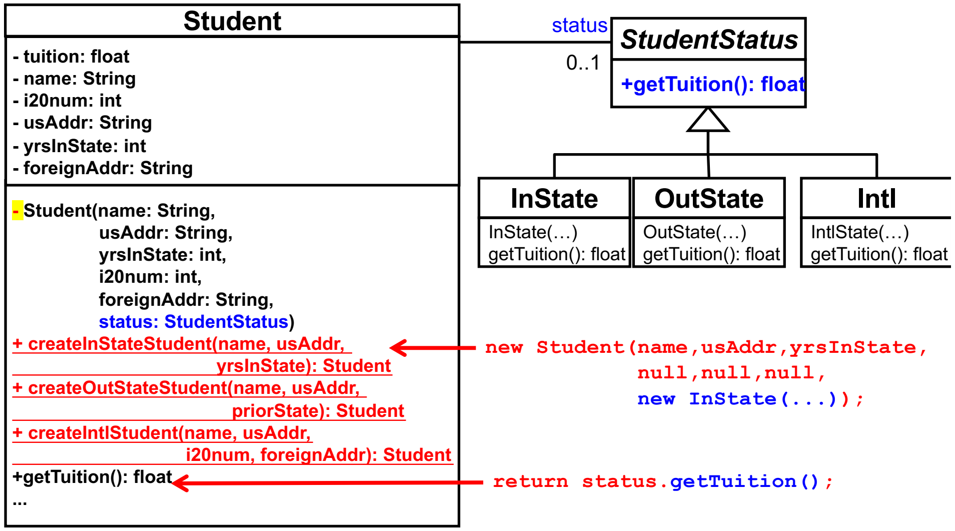
Intl

IntlState(…)
getTuition(): float

return 3000;

```
Student s1 = new Student( ..., new OutState(...) );
s1.getTuition();
s1.changeStatus(new InState);
s1.getTuition();
```

Adding *Static Factory Methods*



Student

- tuition: float
- name: String
- i20num: int
- usAddr: String
- yrsInState: int
- foreignAddr: String

- Student(name: String,
usAddr: String,
yrsInState: int,
i20num: int,
foreignAddr: String,
status: StudentStatus)
+ createInStateStudent(name, usAddr, yrsInState): Student
+ createOutStateStudent(name, usAddr, priorState): Student
+ createIntlStudent(name, usAddr, i20num, foreignAddr): Student
+getTuition(): float
...

status 0..1

StudentStatus

+getTuition(): float

InState
InState(…)
getTuition(): float

OutState
OutState(…)
getTuition(): float

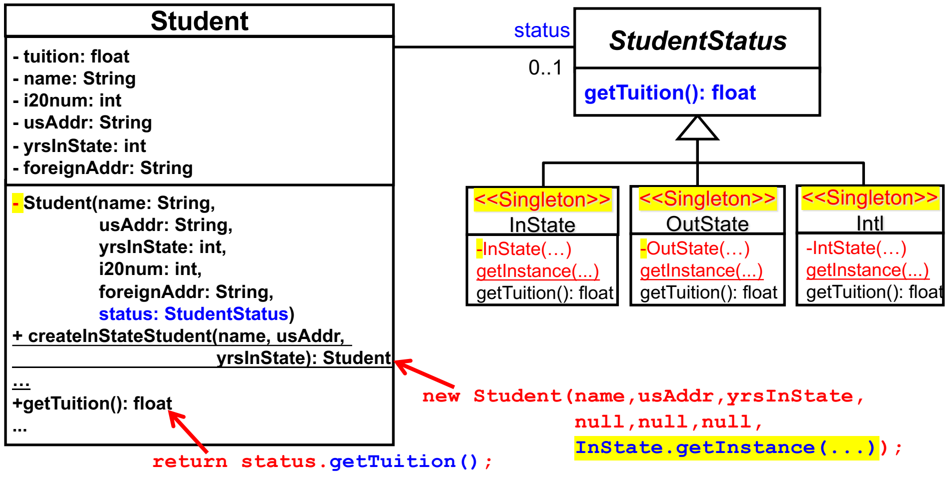
Intl
IntlState(…)
getTuition(): float

```
new Student(name,usAddr,yrsInState,
            null,null,null,
            new InState(...));
```

```
return status.getTuition();
```

```
Student s1 = Student.createInStateStudent("John Smith",...);
s1.getTuition();
```

State Classes as *Singleton*



Student

- tuition: float
- name: String
- i20num: int
- usAddr: String
- yrsInState: int
- foreignAddr: String

- Student(name: String,
usAddr: String,
yrsInState: int,
i20num: int,
foreignAddr: String,
status: StudentStatus)
+ createInStateStudent(name, usAddr, yrsInState): Student
...
+getTuition(): float
...

status 0..1

StudentStatus

getTuition(): float

<<Singleton>>
InState
-InState(…)
getInstance(...)
getTuition(): float

<<Singleton>>
OutState
-OutState(…)
getInstance(...)
getTuition(): float

<<Singleton>>
Intl
-IntState(…)
getInstance(...)
getTuition(): float

```
new Student(name,usAddr,yrsInState,
            null,null,null,
            InState.getInstance(...));
```

```
return status.getTuition();
```

```
Student s1 = Student.createInStateStudent("John Smith",...);
s1.getTuition();
```

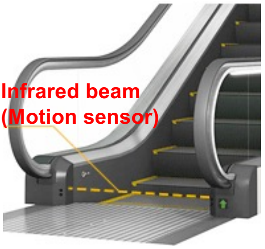
State-dependent Behaviors

- *State* design pattern
  - – Allows each student (`Student` class instance) to change his/her behavior (i.e. returning different tuition $) according to his/her status.
    - State-dependent behavior: tuition calculation
- Benefits
  - – Allows each student to change his/her status dynamically.
  - – Can eliminate conditionals in `Student`'s methods.

Another Example: Firmware to Control Escalators

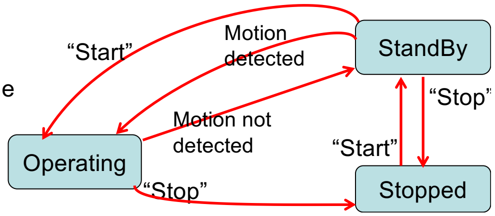
- An escalator performs different behaviors upon an event depending on its current state.
- Focus on an escalator's *behaviors* upon *events*.

- 4 Events
  - – The "Start" button is pushed
  - – The "Stop" button is pushed
  - – Motion detected (with a motion sensor)
  - – Motion not detected for a while (with a motion sensor)
- 3 states
  - – Operating: Keeps moving escalator steps
  - – Standby (idle)
    - Does not move steps because motion has not been detected for a while
    - Keeps running its motion sensor to possibly start moving steps
  - – Stopped: Does not move steps. Does not run its motion sensor.

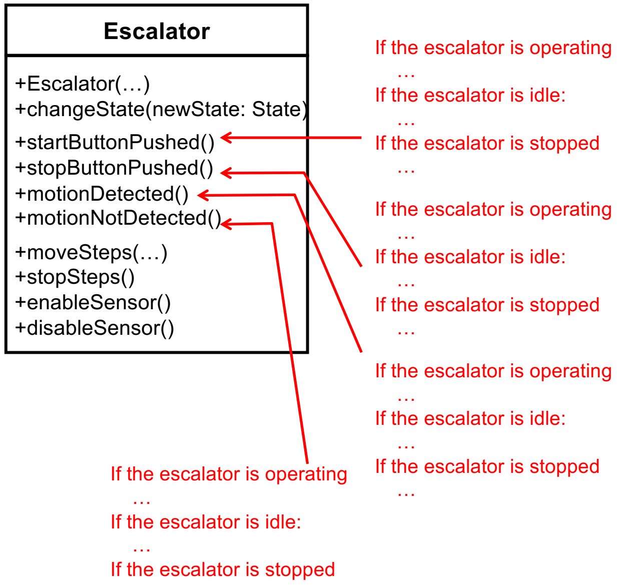


State-Dependent Behaviors

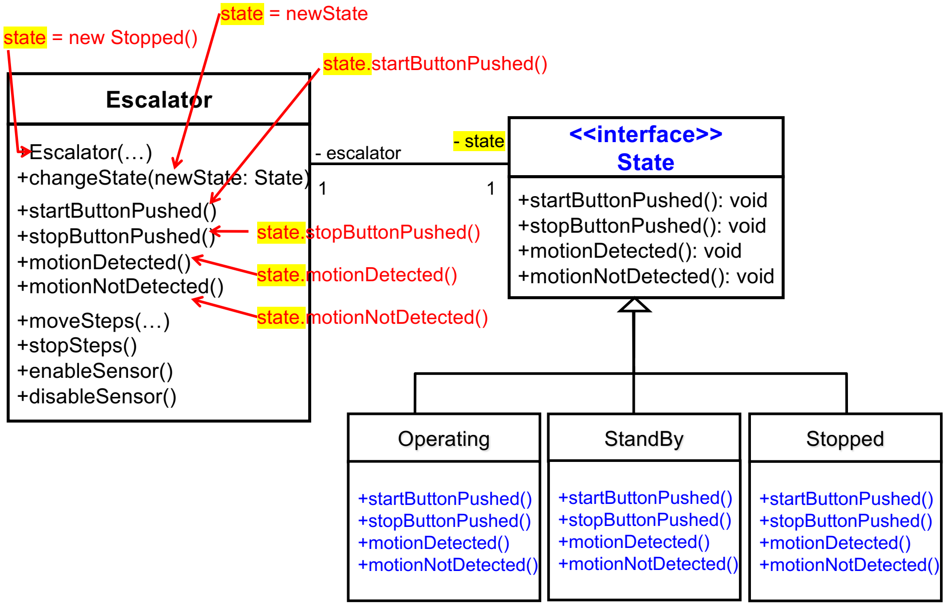
- When the “Start” button is pushed,
  - Does nothing (keeps moving steps)
    - If currently in “Operating”
  - Starts moving steps
    - If currently in “StandBy”
  - Enables the motion sensor and stands by
    - If currently in “Stopped”
- When the “Stop” button is pushed,
  - Does nothing (keeps the escalator stopped)
    - If currently in “Stopped”
  - Disables the motion sensor and stops the escalator
    - If currently in “StandBy”
  - Stops moving steps and disables the motion sensor
    - If currently in “Operating”
- When motion is detected,
  - Does nothing (keeps moving steps)
    - If currently in “Operating”
  - Starts moving steps
    - If currently in “StandBy”
- If motion is not detected,
  - Does nothing (keeps standing by)
    - If currently in “StandBy”
  - Stops moving steps and stands by
    - If currently in “Operating”



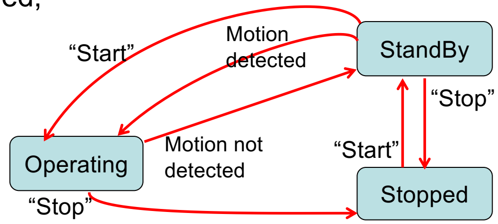
How to Implement State-dependent Behaviors


Escalator
+Escalator(…)
+changeState(newState: State)
+startButtonPushed()
+stopButtonPushed()
+motionDetected()
+motionNotDetected()
+moveSteps(…)
+stopSteps()
+enableSensor()
+disableSensor()
If the escalator is operating
…
If the escalator is idle:
…
If the escalator is stopped
…
If the escalator is operating
…
If the escalator is idle:
…
If the escalator is stopped
…
If the escalator is operating
…
If the escalator is idle:
…
If the escalator is stopped
…
If the escalator is operating
…
If the escalator is idle:
…
If the escalator is stopped

Using *State* Design Pattern

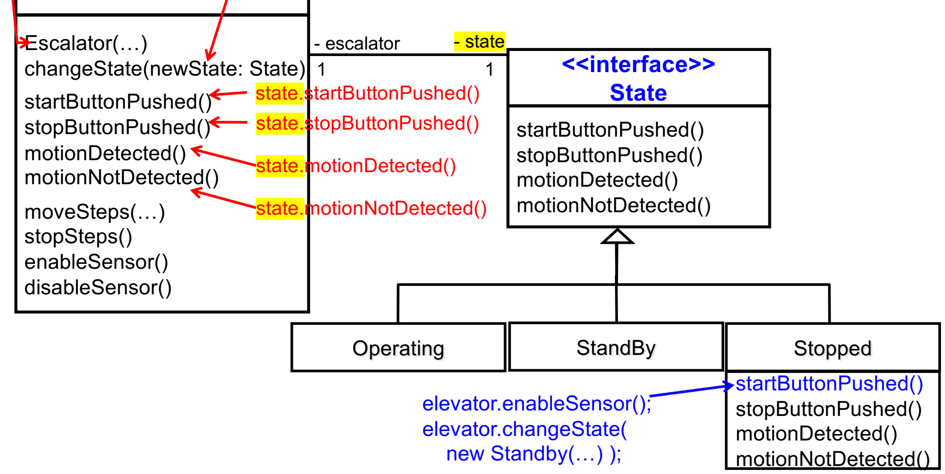


- When the “Start” button is pushed,
  - Enables the motion sensor (and stands by)
    - If currently in “Stopped”

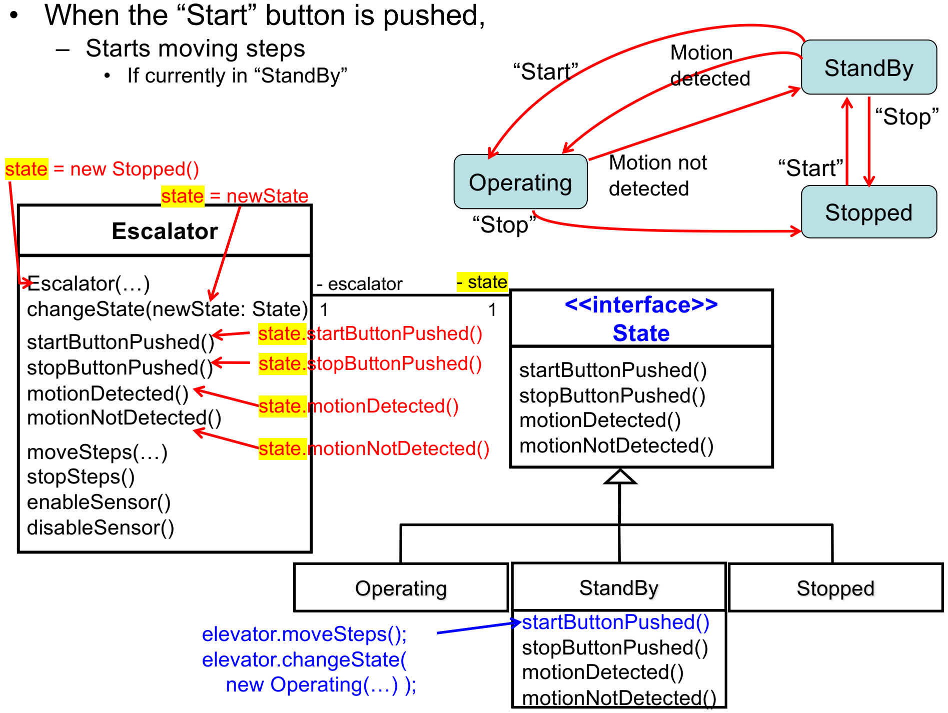


state = new Stopped()

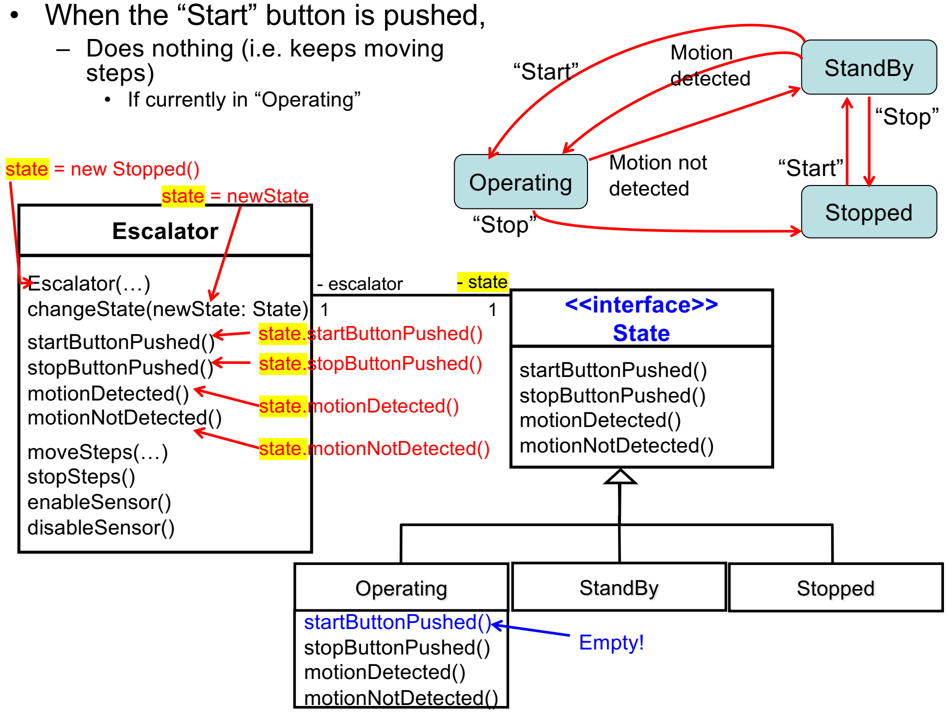
state = newState



- When the “Start” button is pushed,
  - Starts moving steps
    - If currently in “StandBy”



- When the “Start” button is pushed,
  - Does nothing (i.e. keeps moving steps)
    - If currently in “Operating”



Conditional-based or *State*-based Design

- Conditional-based
  - Maybe intuitive/straightforward to implement at first
  - Hard to maintain a long sequence of conditional branches

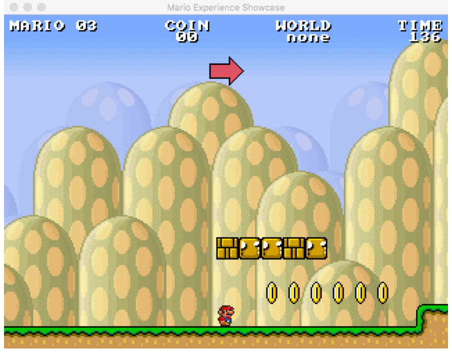
- *State*-based
  - May not be that intuitive/straightforward to implement at first
  - Easier (more principled/disciplined) to maintain
    - If a new button/event is added on the DVD player, just add an extra method to DVDPlayer and add an extra state class.
    - No need to modify many existing methods.
  - Initial cost may be higher, but maintenance cost (or total cost) should be lower over time
    - as changes are made in the future.

Note:

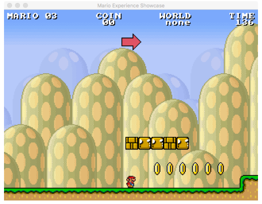
- Each **`State`** subclass and **`Escalator`** class can be *Singleton*.

One More Example: Game Characters

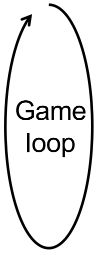
- Game characters often have state-dependent behaviors.
- Think of a simple 2D game like Super Mario



Handling User Inputs



- 5 types of inputs
  - The user can push the right arrow, left arrow, down arrow and “s” keys.
    - R arrow to move right
    - L arrow to move left
    - D arrow to duck
    - “s” to jump
  - The user releases the D arrow to stand up.
- **InputHandler**
  - **handleInput()**
    - identifies a keyboard input since the last game loop iteration (i.e. since the last frame).
    - 60 frames/s (FPS): One input per frame (i.e. during 1.6 msec)

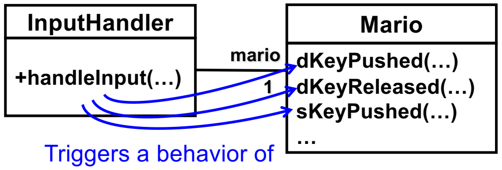


Detect an input from the user
→ Call InputHandler.handleInput()

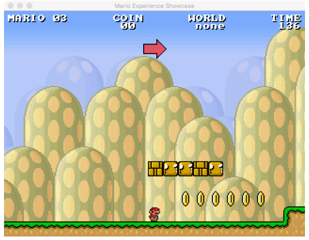
Update the scene (game world)

Render the game world

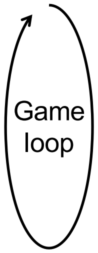
```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput(...);
  ...
}
```



Triggers a behavior of Mario



For simplicity, let’s focus on 3 inputs only here:
D arrow pushed, D arrow released, and “s” key pushed

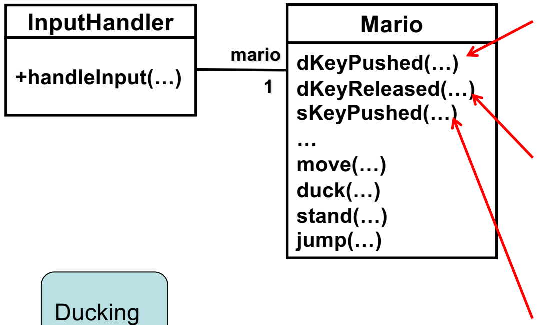


Detect an input from the user
→ Call InputHandler.handleInput()

Update the scene (game world)

Render the game world

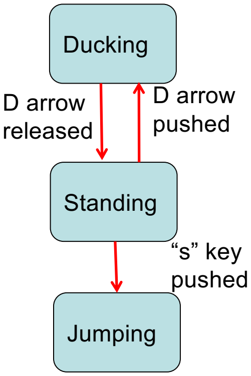
```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput(...);
    // If D arrow is pushed,
    //     call dKeyPushed() on Mario
    // If D arrow is released,
    //     ...
  ...
}
```



```
if(!isJumping){
  // Cannot duck when jumping
  duck(...);
}
```

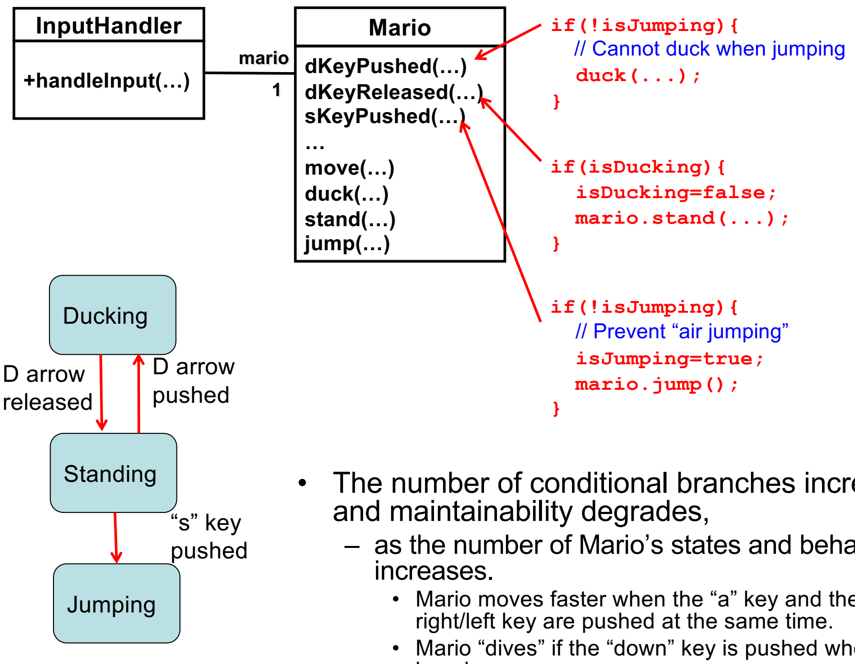
```
if(isDucking){
  isDucking=false;
  mario.stand(...);
}
```

```
if(!isJumping){
  // Prevent “air jumping”
  isJumping=true;
  mario.jump();
}
```



Mario differently behaves upon an event (or responds to an event) depending on his current state.

Mario has and performs state-dependent behaviors

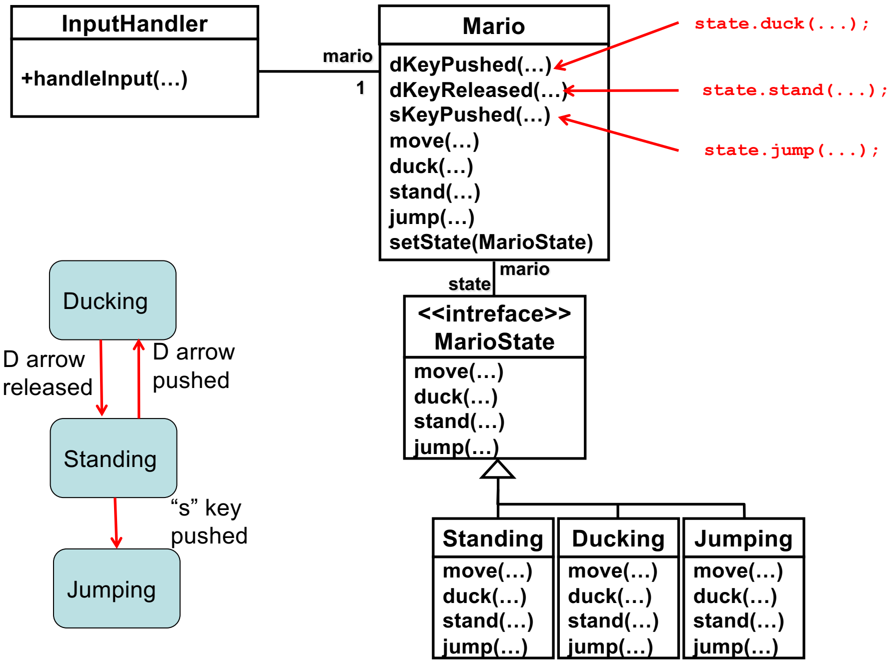


```
if(!isJumping){
  // Cannot duck when jumping
  duck(...);
}
```

```
if(isDucking){
  isDucking=false;
  mario.stand(...);
}
```

```
if(!isJumping){
  // Prevent “air jumping”
  isJumping=true;
  mario.jump();
}
```

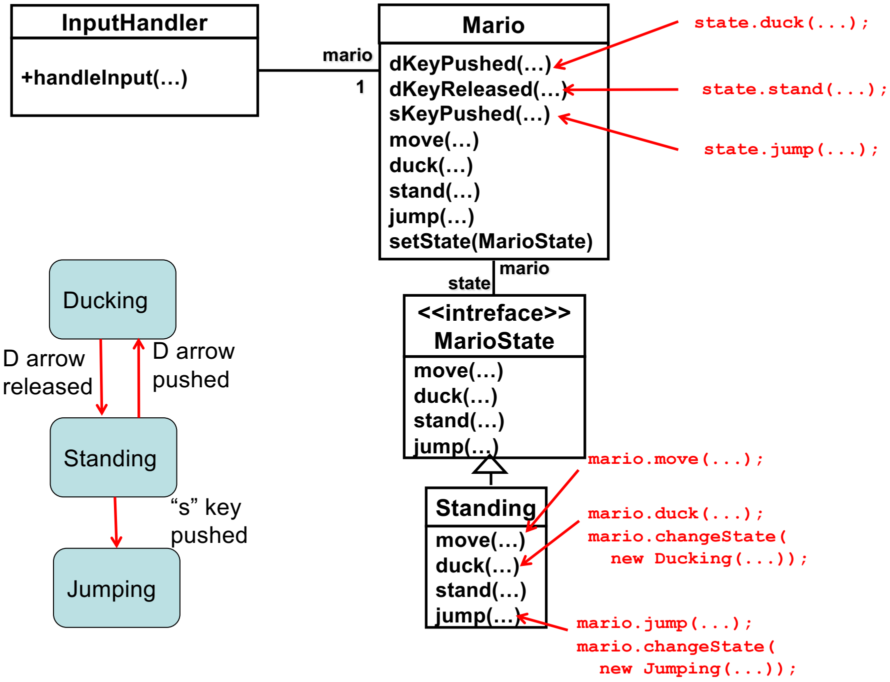
- The number of conditional branches increases and maintainability degrades,
  - as the number of Mario’s states and behaviors increases.
    - Mario moves faster when the “a” key and the right/left key are pushed at the same time.
    - Mario “dives” if the “down” key is pushed when jumping.



```
state.duck(...);
```

```
state.stand(...);
```

```
state.jump(...);
```



```
state.duck(...);
```

```
state.stand(...);
```

```
state.jump(...);
```

```
mario.move(...);
```

```
mario.duck(...);
mario.changeState(
  new Ducking(...));
```

```
mario.jump(...);
mario.changeState(
  new Jumping(...));
```