The State Pattern

Mel Ó Cinnéide
School of Computer Science
University College Dublin

The State Pattern

Intent

Allow an object to alter its behaviour when its internal state changes. The object will *appear* to change class.

In most cases, objects of a class respond to the same message in a similar way, e.g. `calculateNetPrice` will do a similar calculation for any product it is applied to.

Sometimes the same message will elicit a fundamentally different response, depending on the **state** of the receiver.

e.g. how a car responds to `pressAccelerator` depends on whether the engine is running or not.

State — Motivating Example

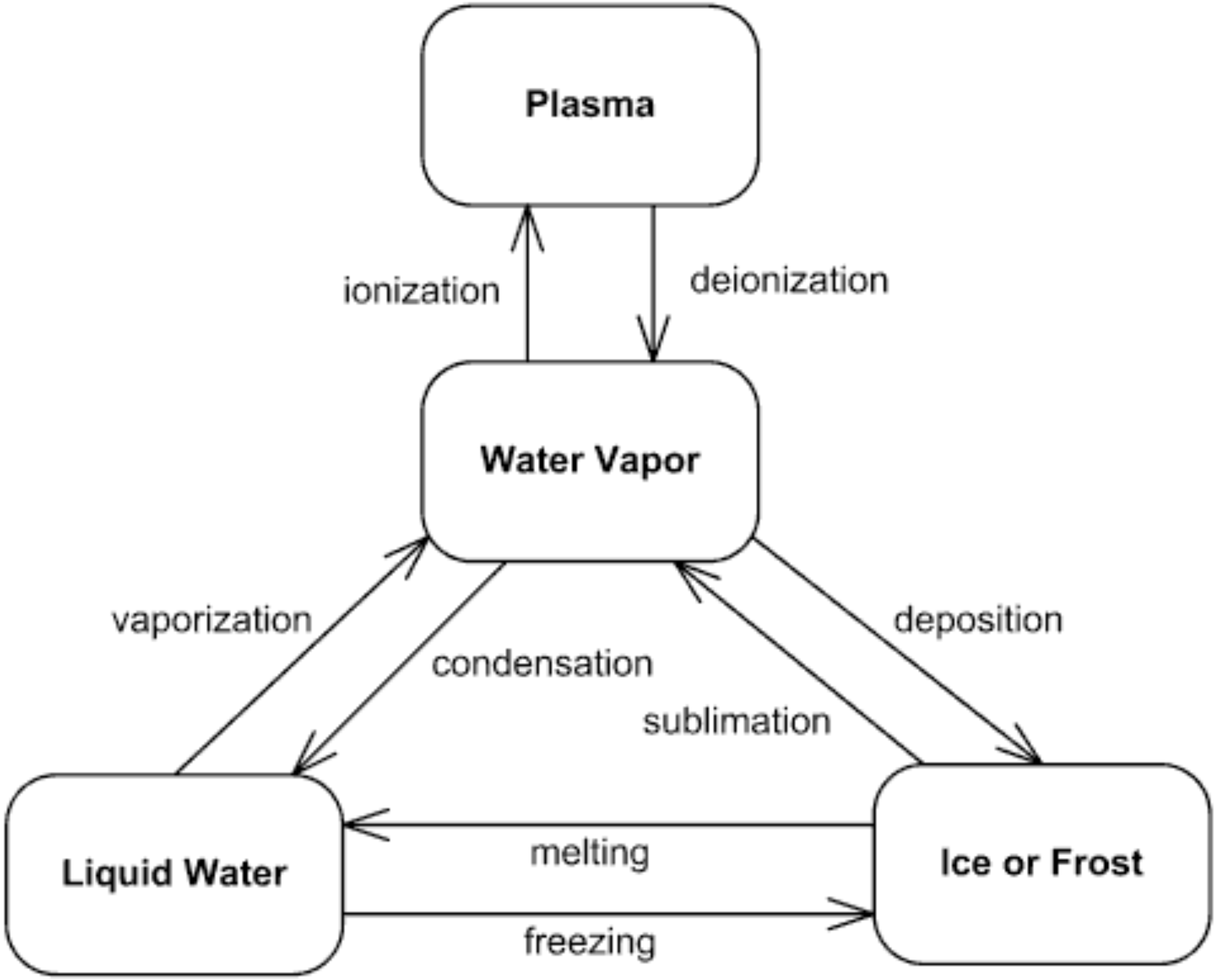
Consider modelling H_20 as a class, and how these methods might be implemented:

```
pour
mix(other: Chemical)
freeze
melt
```

We’ll consider just two states: water and ice.

Water can be poured, but not ice. Ice can be melted, but water can’t, etc.

States of Water



Coding the H2O class

```
public void pour(){
  if (state == WATER)
    // pour it
  else
    // do nothing
}

public void freeze(){
  if (state == ICE)
    // do nothing
  else
    state = ICE;
}

private H2OState state;
```

Each method checks and may change the state

Hard to understand each state individually

Hard to understand state transitions

Adding/removing states will be hard work!

Applying the State Pattern

Each state is represented by a class, so we have classes for `Water` and `Ice`.

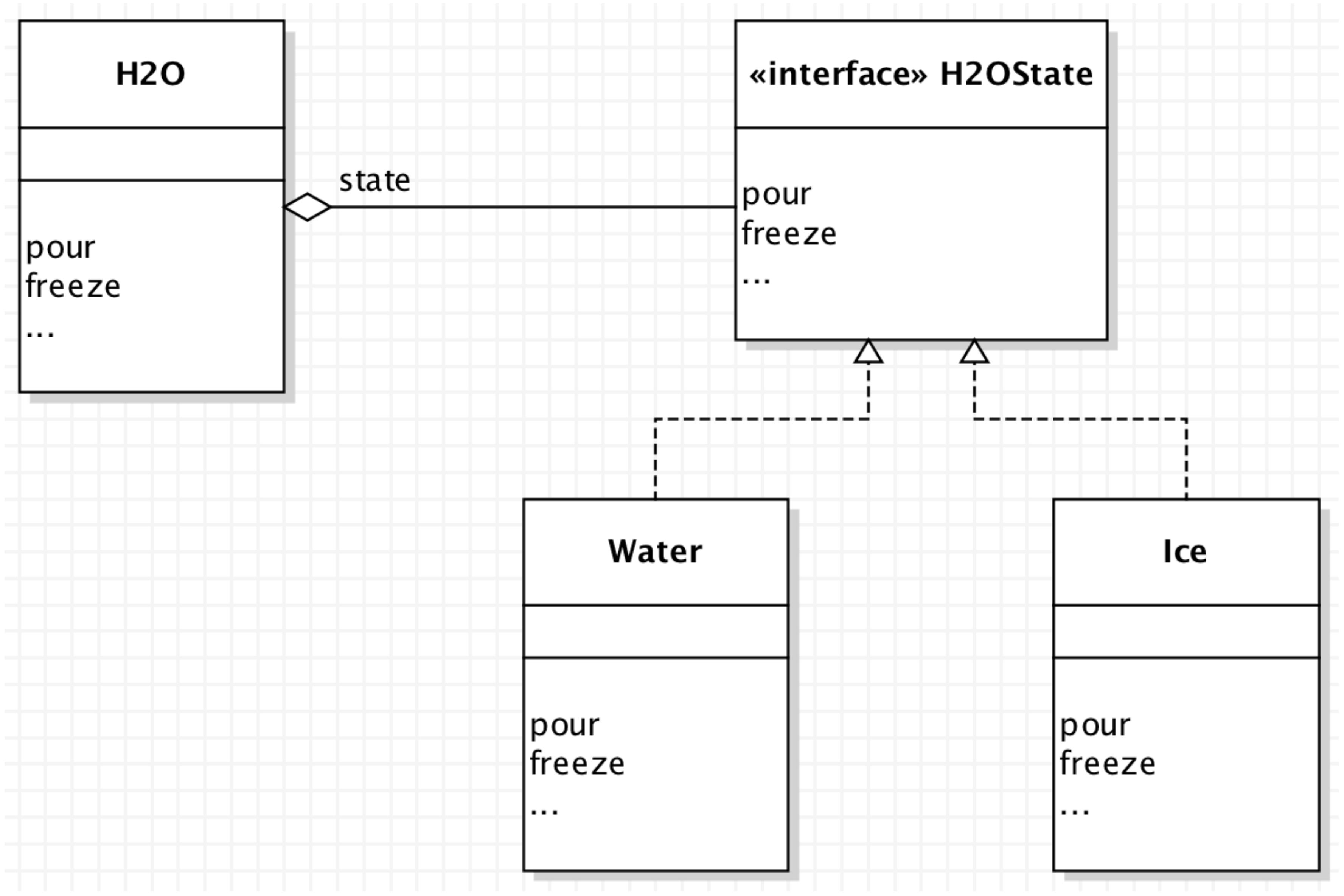
In these classes, the methods (pour, freeze etc.) are implemented but only for the state the class represents.

`Water` and `Ice` have the same interface, which is placed in the `H2OState` interface.

The `H2O` class is given a reference to a `H2OState` object, and delegates any messages to this object.

State transitions are achieved by assigning the `H2OState` field a different state object.

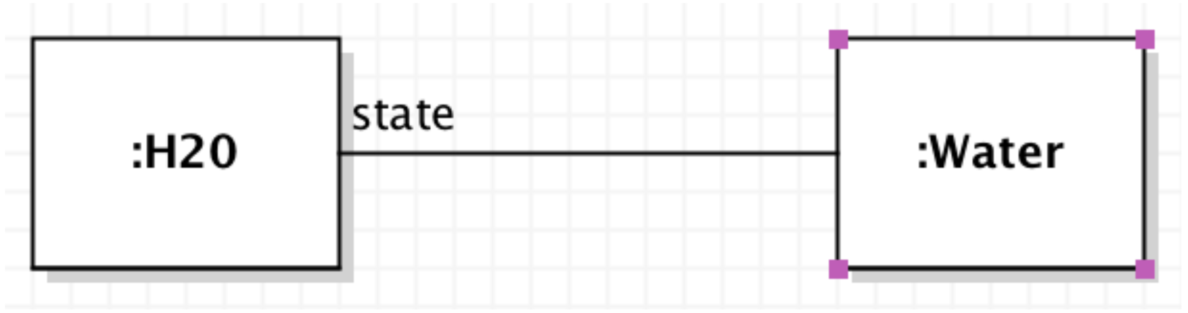
Structure of the State Pattern Solution



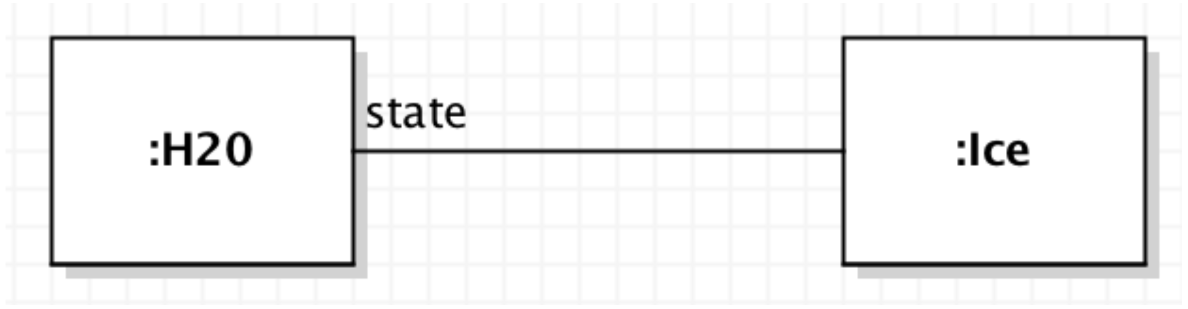
State Object Structure

Each **H2O** object now has a reference to its state object.

Water



Ice



Sample Java Code from this solution

The **`H2O`** class delegates everything to its **`H2OState`** object.

```
class H2O {
  public void H2O(){
    state = new Water();
  }

  public void pour(){
    state.pour();
  }

  H2OState state;
}
```

Methods whose implementations depend on the object state are delegated to the `H2OState` object.

Sample Code from this solution

Each state class implements the methods appropriately for its state, e.g.

```
class Water {

  public void pour(){
    // perform pouring
  }

  public void freeze(){
    // change state to ice
  }

  public void thaw(){
  }

}
```

See next slide on **state transitions**

Water is already thawed.

Where are state transitions handled?

There are three possibilities:

1. In the Context class

2. In the State class.

3. In a separate state transition object

We look at these in the following slides.

(1) State transitions in the Context class

Here the Context explicitly encodes the state transitions.

For example, when water freezes, the `H2O` object changes its state reference to refer to an `Ice` object.

Pros:

+ easy to code

Cons:

- context is cluttered with state transitions, and knows about all the states
- state transition code is decentralised

This approach works best if the states and state transitions are fixed.

(2) State transitions in the State classes

Here each state specifies its successor state, and when to make the transition.

For example, if a `Water` object is asked to freeze, it informs the `H2O` object to set its state object to `Ice`.

Pros:

+ context is decoupled from the states
+ adding a new state is easy (no need to change the context)

Cons:

- each state knows about at least one other state (unless a terminal state)
- Context interface must allow state object to be changed
- state transition code is still decentralised

(3) State transitions in a separate object

The state transitions are stored in a separate object, which typically contains a table that maps each (state, event) pair to the next state, e.g.

```
(water, freeze) => ice
```

Suitable for complex state transitions, or when changing state transition rules at runtime is necessary.

Pros:

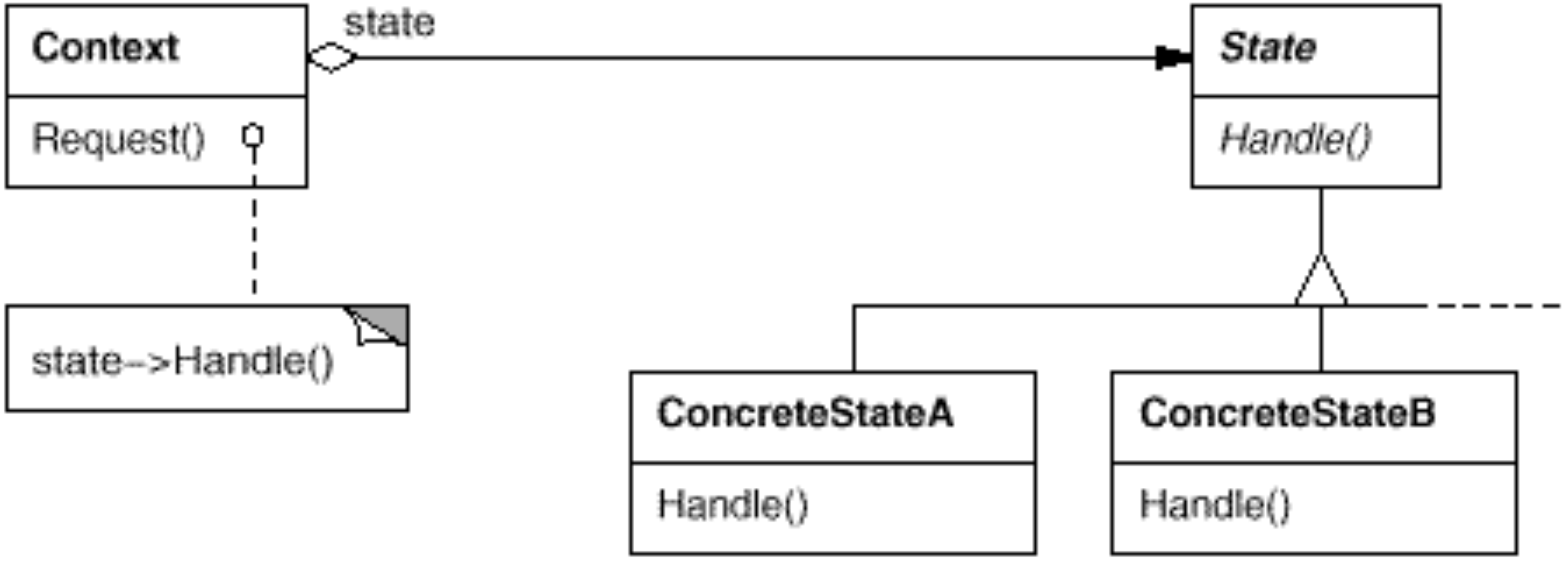
+ state transitions are centralised
+ state transitions can be changed dynamically

Cons:

- complex to code
- transition criteria are encoded in the table => less explicit

State -- Structure

Typical class diagram for State:



State -- Comments

State-transition code can be placed either in Context or in State. Most flexible approach is to make this table-driven in a separate object.

States can be created upfront, or created as needed.

A state object is usually stateless (may sound odd, but think about it).

- Many contexts can share the one state object, i.e. the state object is also a **Flyweight.**
- **Singleton** may be useful here too.

State -- Comments

State and Strategy may have a similar structure, but their **intents** are different.

As with Strategy, the states will likely need to access their Context object. Same solutions are possible.

Helps with SRP and OCP. Tends to create additional coupling between State classes and Context.

Using the state pattern for a simple case like a regular light switch would be over-engineering.