State

Department of Computer Science
University of Pretoria

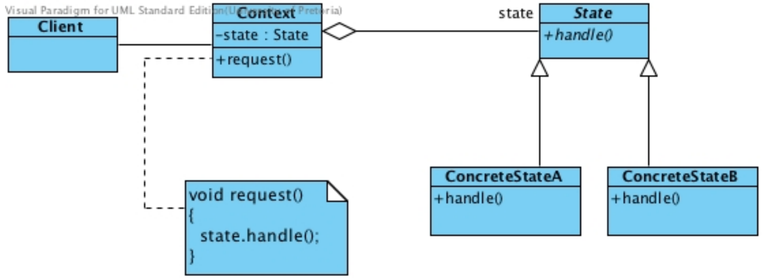
14 August 2023

Name and Classification:

State (Behavioural)

Intent:

"Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class." GoF(305)


Visual Paradigm for UML Standard Edition(University of Pretoria)
Client
Context
-state : State
+request()
state
State
+handle()
void request()
{
state.handle();
}
ConcreteStateA
+handle()
ConcreteStateB
+handle()

State

- Defines an interface for encapsulating the behaviour associated with a particular state of the Context.

ConcreteState

- Implements a behaviour associated with a state of the Context.

Context

- Maintains an instance of a ConcreteState subclass that defines the current state.
- Defines the interface of interest to clients.

Related Patterns

- **Strategy**(315): State and Strategy have the same structure and apply the same techniques to achieve their goals, but differ in intent. Strategy is about implementations which accomplish the same result. One implementation can replace the other as the strategy requires. State is about doing different things based on the state, this relieves the caller from the burden of accommodating every possible state.

Related Patterns cont.

- **Prototype** (117): When implementing the State pattern, the programmer has to decide on how the state objects will be created. Often the application of the Prototype pattern will be ideal.
- **Singleton** (127): State objects are often Singletons.
- **Flyweight**(195): State objects can be shared by applying Flyweight.

The State design pattern is effective when:

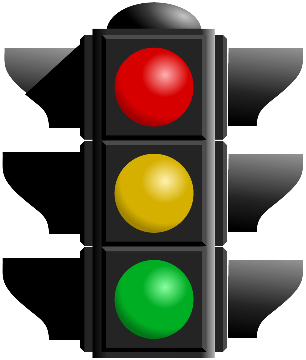
- **an object becomes large** - state of the object is managed externally to the object itself
- **an object can experience an extensive number of state changes** - specifically when flow-control is characterised by multiple *if* or *switch* statements. State is managed externally, by modelling each state as an individual object.

Changing the state of an object will influence the behaviour of the object at run-time.

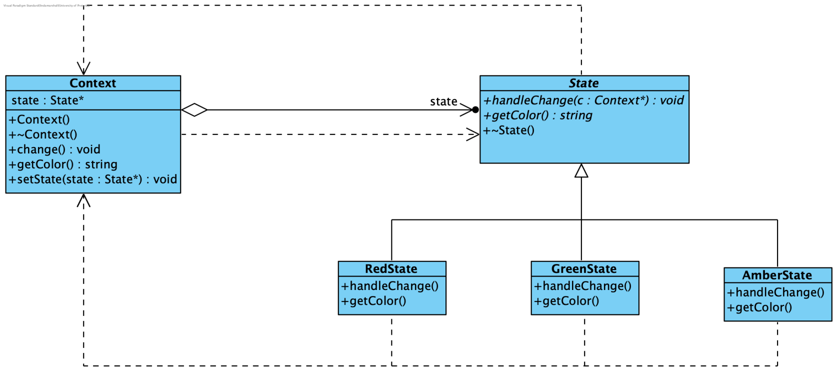
The `Context` is the object and the `ConcreteStates` the external state of the object.

```
int main(){

Player* redPlayer = new Player("Red", 15, 6);
Player* blackPlayer = ...

PlayerContext* redContext =
        new PlayerContext(redPlayer, new Advance());
PlayerContext* blackContext = ...

srand(time(0));

bool redAlive = true;
bool blackAlive = true;
while (redAlive && blackAlive) {
        redAlive = redContext->action(blackPlayer);
        if (redAlive && blackAlive)
                blackAlive = blackContext->action(redPlayer);
}

...
return 0;
}
```

```
class PlayerContext {
public:
        PlayerContext(Player*, PlayerState*);
        bool action(Player*);
        Player* getPlayer();
        void setPlayerState(PlayerState*);
        virtual ~PlayerContext();
protected:
        PlayerContext();
private:
        Player* player;
        PlayerState* pState;
};
```



Class diagram from existing code



```
int main(){
    Context* context = new Context();
    for (;;) {
        cout << "Traffic light is currently: "
             << context->getColor() << endl;
        context->change();
    }
    delete context;
    return 0;
}
```

```
class Context {
  public:
    Context();
    ~Context();
    void change();
    string getColor();
    void setState(State* state);

  protected:
    Context(State* state_);
    State* getState();

  private:
    State* state;
};
```

```
class State {
  public:
    virtual void handleChange(Context* c) = 0;
    virtual string getColor() = 0;
    virtual ~State();
};

class RedState : public State {
  public:
    virtual void handleChange(Context* c);
    virtual string getColor();
};
class GreenState : public State {
  public:
    virtual void handleChange(Context* c);
    virtual string getColor();
};
class AmberState : public State {
  public:
    virtual void handleChange(Context* c);
    virtual string getColor();
};
```

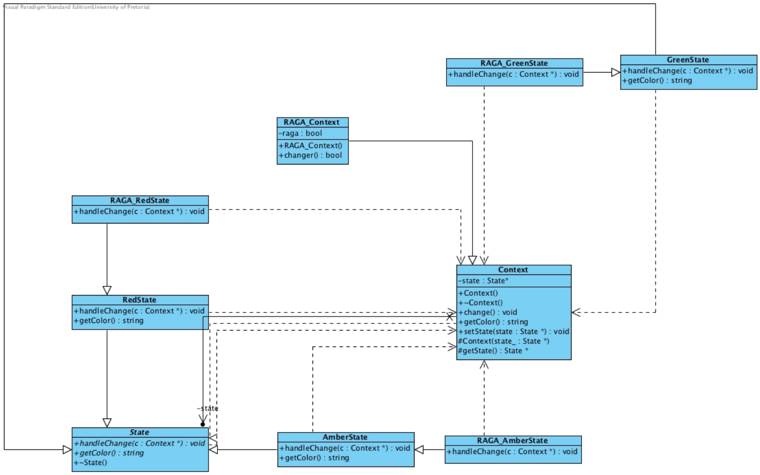
Note the dependency on `Context` in `State` and the subclasses. `Context` holds a pointer to a `State` object.
This results in a circular dependency. `State` must be defined before `Context`, which in turn must be defined before `State`.
How do we resolve this?

```
class State; // Forward declaration

class Context {
  // as above
};

class State {
  // as above
};

class RedState : public State {
  // as above
};

// same for GreenState and AmberState
```

```
Context::Context() {
    this->state = new RedState();
}
Context::~Context() {
    delete this->state;
}
Context::Context(State* state_) {
    this->state = state_;
}
State* Context::getState() {
    return state;
}
void Context::setState(State* state) {
    delete this->state;
    this->state = state;
}
void Context::change() {
    state->handleChange(this);
}
string Context::getColor() {
    return state->getColor();
}
```

```
void RedState::handleChange(Context* c) {
    c->setState(new GreenState());
}

string RedState::getColor() {
    return "Red";
}

// Similar for GreenState and AmberState
```

What if the sequence needs to change to:
Red $\rightarrow$ Amber $\rightarrow$ Green $\rightarrow$ Amber ?


Visual Paradigm Standard Edition(University of Pretoria)
RAGA_GreenState
+handleChange(c : Context *) : void
GreenState
+handleChange(c : Context *) : void
+getColor() : string
RAGA_Context
-raga : bool
+RAGA_Context()
+changer() : bool
RAGA_RedState
+handleChange(c : Context *) : void
RedState
+handleChange(c : Context *) : void
+getColor() : string
Context
-state : State*
+Context()
+~Context()
+change() : void
+getColor() : string
+setState(state : State *) : void
#Context(state_ : State *)
#getState() : State *
-state
State
+handleChange(c : Context *) : void
+getColor() : string
+~State()
AmberState
+handleChange(c : Context *) : void
+getColor() : string
RAGA_AmberState
+handleChange(c : Context *) : void

```
class RAGA_Context : public Context {
public:
  RAGA_Context();
  bool changer();
private:
  bool raga;
};

RAGA_Context::RAGA_Context() : Context() {
  setState(new RAGA_RedState());
  raga = true;
}

bool RAGA_Context::changer() {
  raga = !raga;
  return raga;
}
```

```
class RAGA_AmberState : public AmberState {
public:
  virtual void handleChange(Context* c);
};

void RAGA_AmberState::handleChange(Context* c) {
   // Note the cast
  RAGA_Context* RAGA_c = static_cast<RAGA_Context*>(c);
  if (RAGA_c->changer())
    c->setState(new RAGA_RedState());
  else
    c->setState(new RAGA_GreenState());
}
```

An alternative solution would be to make `RAGA_AmberState` a state pattern in our state pattern.

In the UK the traffic lights have the following sequence, how do we model them?