28 Behavioral: State Pattern — Traffic-Light Controller Assignment

Model a traffic light that rotates through **Green → Yellow → Red** using the **State pattern**.  
 Each color is its own class; the TrafficLightContext delegates behavior to the current state and swaps it after each request. Add JavaDoc to every class and finish with a short **Reflection**.

.

├── analysis

│ └── state\_need.md ← why states > switch-blocks

├── src/main/java

│ └── traffic

│ ├── state

│ │ ├── TrafficLightState.java

│ │ ├── GreenState.java

│ │ ├── YellowState.java

│ │ └── RedState.java

│ ├── context

│ │ └── TrafficLightContext.java

│ └── StateDemo.java

├── src/test/java/traffic

│ ├── CycleTest.java

│ └── RepeatCycleTest.java

├── reflection.md

└── README.md

#### **1 TrafficLightState.java**

package traffic.state;

/\*\*

\* Common contract for all traffic-light colours.

\* {@link #handle()} prints a message and asks the context to switch state.

\*/

public interface TrafficLightState {

void handle(); // perform colour-specific action

}

#### **2 Concrete states**

package traffic.state;

import traffic.context.TrafficLightContext;

/\*\* Cars may go; next colour is yellow. \*/

public class GreenState implements TrafficLightState {

private final TrafficLightContext ctx;

public GreenState(TrafficLightContext ctx){ this.ctx = ctx; }

@Override public void handle(){

System.out.println("Green: cars GO");

ctx.setState(new YellowState(ctx));

}

}

package traffic.state;

import traffic.context.TrafficLightContext;

/\*\* Vehicles prepare to stop; next colour is red. \*/

public class YellowState implements TrafficLightState {

private final TrafficLightContext ctx;

public YellowState(TrafficLightContext ctx){ this.ctx = ctx; }

@Override public void handle(){

System.out.println("Yellow: prepare to STOP");

ctx.setState(new RedState(ctx));

}

}

package traffic.state;

import traffic.context.TrafficLightContext;

/\*\* Vehicles must halt; next colour is green. \*/

public class RedState implements TrafficLightState {

private final TrafficLightContext ctx;

public RedState(TrafficLightContext ctx){ this.ctx = ctx; }

@Override public void handle(){

System.out.println("Red: cars STOP");

ctx.setState(new GreenState(ctx));

}

}

#### **3 TrafficLightContext.java**

package traffic.context;

import traffic.state.\*;

/\*\*

\* Context owning the current {@link TrafficLightState}.

\* Clients call {@link #request()} instead of inspecting colour flags.

\*/

public class TrafficLightContext {

private TrafficLightState state = new GreenState(this); // initial colour

/\*\* Package-private mutator used only by state classes. \*/

public void setState(TrafficLightState s){ state = s; }

/\*\* Delegates to current state; state may change afterwards. \*/

public void request(){ state.handle(); }

}

#### **4 Demo client**

package traffic;

import traffic.context.TrafficLightContext;

/\*\* Runs four state transitions to demonstrate the cycle. \*/

public class StateDemo {

public static void main(String[] args){

TrafficLightContext light = new TrafficLightContext();

light.request(); // Green → Yellow

light.request(); // Yellow → Red

light.request(); // Red → Green

light.request(); // Green → Yellow again

}

}

Console

Green: cars GO

Yellow: prepare to STOP

Red: cars STOP

Green: cars GO

#### **5 JUnit-style tests (outline)**

/\* CycleTest.java \*/

TrafficLightContext tl = new TrafficLightContext();

tl.request(); tl.request(); tl.request();

StringWriter sw = new StringWriter();

System.setOut(new PrintStream(new WriterOutputStream(sw), true));

tl.request(); // fourth call should print “Green”

assertTrue(sw.toString().contains("Green"));

/\* RepeatCycleTest.java \*/

for(int i=0;i<300;i++) tl.request(); // should never throw

## **reflection.md**

Encapsulating colour logic inside independent **state classes** removes bulky switch/if ladders from the context, letting each class handle its own transition decision.

| **Benefit** | **Note** |
| --- | --- |
| Single-responsibility | Each state owns only its colour’s behaviour. |
| Easy extension | Add “Flashing Yellow” by introducing one new class; context code unchanged. |
| Readable runtime flow | context.request() mirrors a real-world “tick”. |

*Drawbacks*

* More classes (3 + interface vs one enum with switch).
* Tiny runtime overhead from one extra delegation per call.

Use State when behaviour varies dramatically by mode and you anticipate new modes or complex rules; stick with simple conditionals for two-state toys.