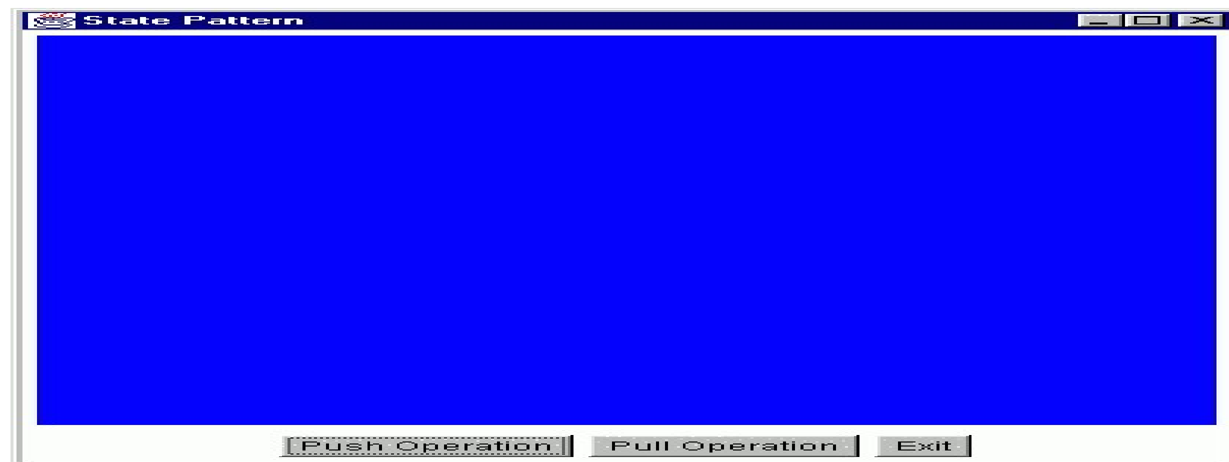
Design Patterns: State

State Pattern Ex.1

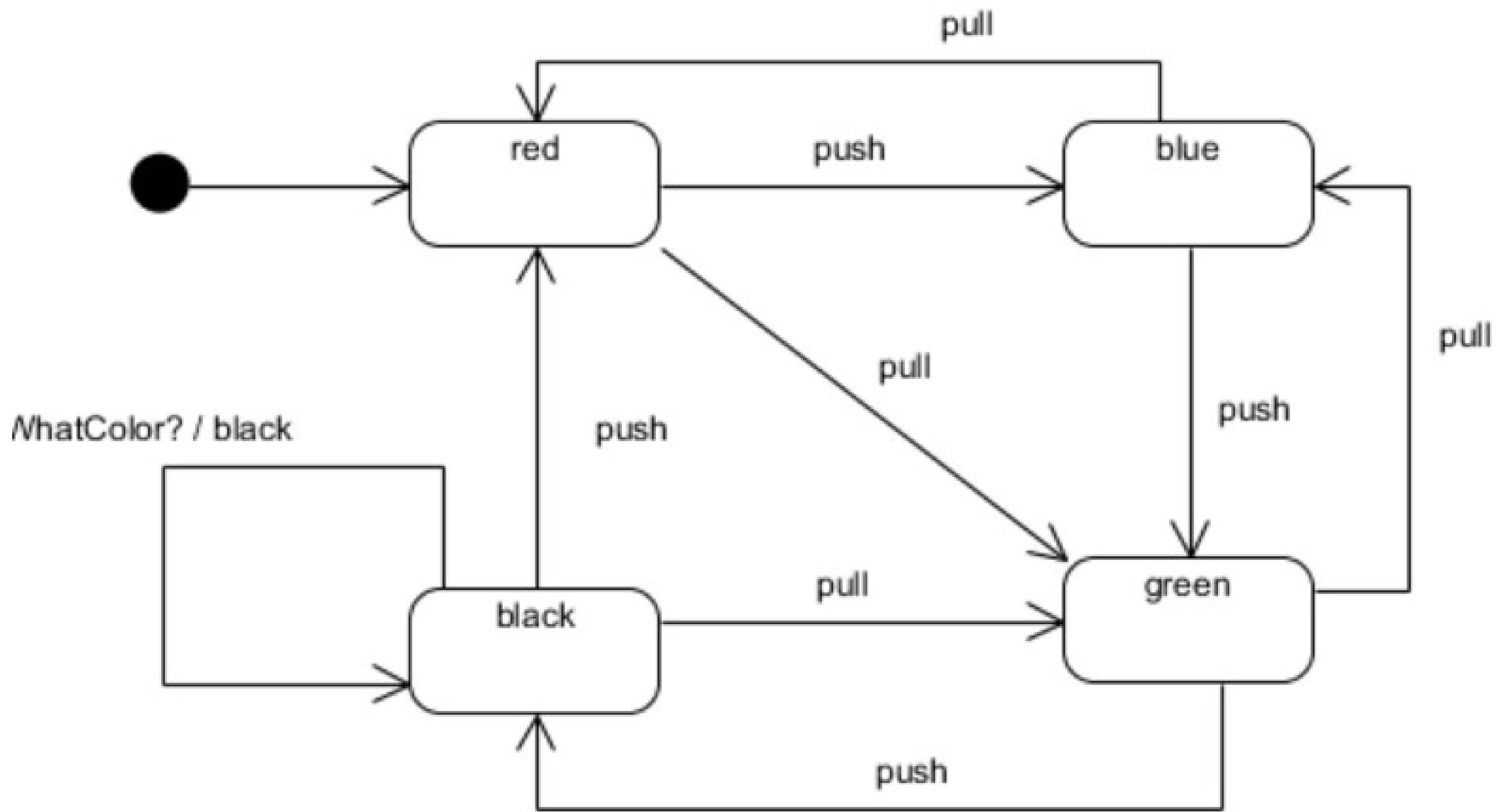
Consider a class with two methods, push() and pull(), whose behavior changes depending on the object state

To send the push and pull requests to the object, we'll use the following GUI with "Push" and "Pull" buttons:



The state of the object will be indicated by the color of the canvas in the top part of the GUI

The states are: black, red, blue and green


pull
red
push
blue
pull
pull
push
WhatColor? / black
push
pull
black
green
push

Example 1 (without the State pattern)

First, let's do this without the State pattern:

```
/**
* Class ContextNoSP has behavior dependent on its state.  The push() and pull()
*  methods do different things depending on the state of the object.
* This class does NOT use the State pattern.
*/
public class ContextNoSP {
    // The state!
private Color state = null;
    // Creates a new ContextNoSP with the specified state (color).
public ContextNoSP(Color color) {state = color;}
    // Creates a new ContextNoSP with the default state
public ContextNoSP() {this(Color.red);}
```

Example 1 (without the State pattern)

```
    // Returns the state.
public Color getState() {return state;}
    // Sets the state.
public void setState(Color state) {this.state = state;}
    /**
    * The push() method performs different actions depending on the state
    of the object. Actually, right now the only action is to make a state
    transition.
*/
public void push() {
    if (state == Color.red) state = Color.blue;
    else if (state == Color.green) state = Color.black;
      else if (state == Color.black) state = Color.red;
         else if (state == Color.blue) state = Color.green;
}
```

Example 1 (without the State pattern)

```
/**
* The pull() method performs different actions depending
* on the state of the object. Actually, right now
* the only action is to make a state transition.
*/
public void pull() {
    if (state == Color.red) state = Color.green;
    else if (state == Color.green) state = Color.blue;
      else if (state == Color.black) state = Color.green;
         else if (state == Color.blue) state = Color.red;
}
}
```

Example 1 (without the State pattern)

Here’s part of the GUI test program:

```
/**
* Test program for the ContextNoSP class which does NOT use
   the State pattern.
*/
public class TestNoSP extends Frame implements
   ActionListener {
   // GUI attributes.
   private Button pushButton = new Button("Push Operation");
   private Button pullButton = new Button("Pull Operation");
   private Button exitButton = new Button("Exit");
   private Canvas canvas = new Canvas();
   // The Context.
   private ContextNoSP context = null;
```

Example 1 (without the State pattern)

```
public TestNoSP() {
   super("No State Pattern");
   context = new ContextNoSP();
   setupWindow();
}
private void setupWindow() { // Setup GUI }

// Handle GUI actions.
public void actionPerformed(ActionEvent event) {
   Object src = event.getSource();
   if (src == pushButton) {
   context.push();
   canvas.setBackground(context.getState());
     }
```

Example 1 (without the State pattern)

```
else if (src == pullButton) {
    context.pull();
    canvas.setBackground(context.getState());
    }
else if (src == exitButton) {
    System.exit(0);
    }
  }
// Main method.
public static void main(String[] argv) {
    TestNoSP gui = new TestNoSP();
    gui.setVisible(true);
    }
}
```

State Pattern

Behavioral Pattern

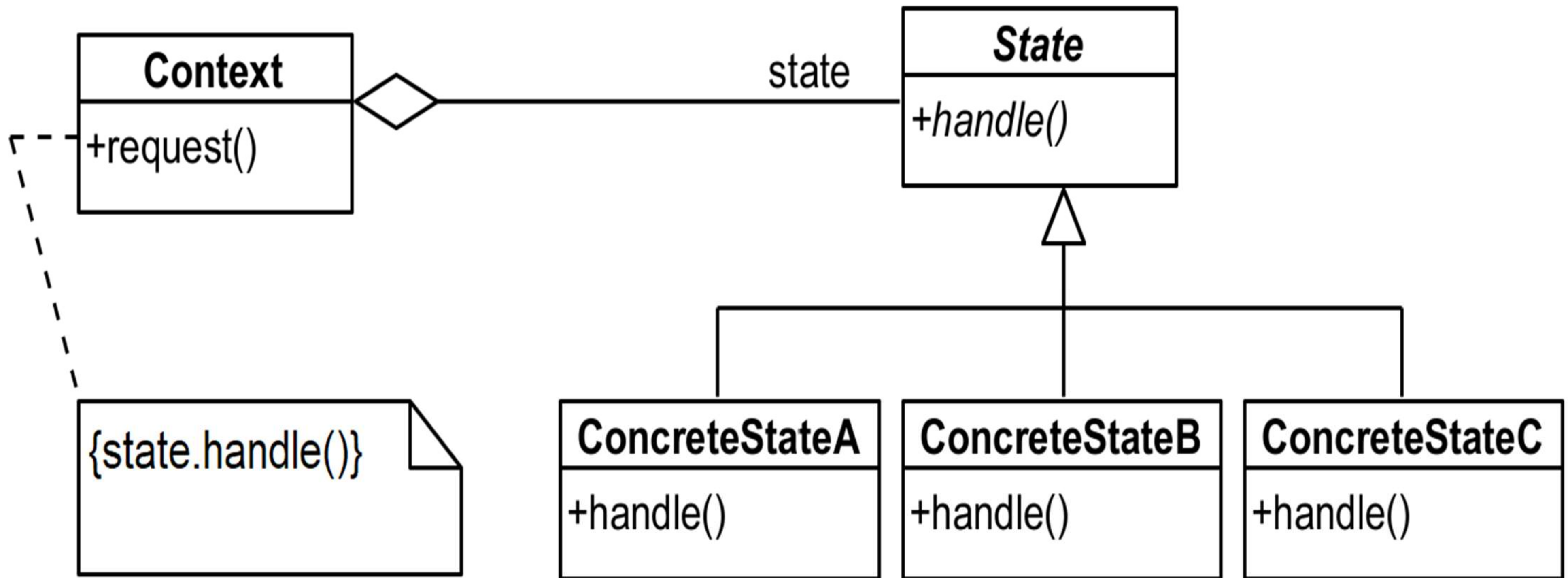
Intent

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

Example

- TCP Connection responds differently to clients based on state
  - Established
  - Listening
  - Closed

Structure



Participanti

Context

- Definisce l'interfaccia di interesse per i client
- Mantiene un'istanza di ConcreteState che definisce lo stato corrente.

State

- Definisce un'interfaccia per incapsulare il comportamento associato a un particolare stato del contesto.
- Può anche essere una classe concreta con un'implementazione predefinita

ConcreteState

- Ogni sottoclasse implementa un comportamento associato a uno stato del contesto

State Pattern Example 1 (Continued)

First, we'll define the abstract State class:

```
/**
 * Abstract class which defines the interface for the
 * behavior of a particular state of the Context.
 */
public abstract class State {
    public abstract void handlePush(Context c);
    public abstract void handlePull(Context c);
    public abstract Color getColor();
}
```

Next, we'll write concrete State classes for all the different states:

RedState, BlackState, BlueState and GreenState

State Pattern Example 1 (Continued)

For example, here's the BlackState class:

```
public class BlackState extends State {
    // Next state for the Black state:
    // On a push(), go to "red"
    // On a pull(), go to "green"
public void handlePush(Context c) {
    c.setState(new RedState());
}
public void handlePull(Context c) {
    c.setState(new GreenState());
}
public Color getColor() {return (Color.black);}
}
```

State Pattern Example 1 (Continued)

And, here's the new Context class that uses the State pattern and the State classes:

```
/**
* Class Context has behavior dependent on its state.
* This class uses the State pattern.
* Now when we get a pull() or push() request, we
* delegate the behavior to our contained state object!
*/
public class Context {
    // The contained state.
  private State state = null;          // State attribute
    // Creates a new Context with the specified state.
  public Context(State state) {this.state = state;}
```

State Pattern Example 1 (Continued)

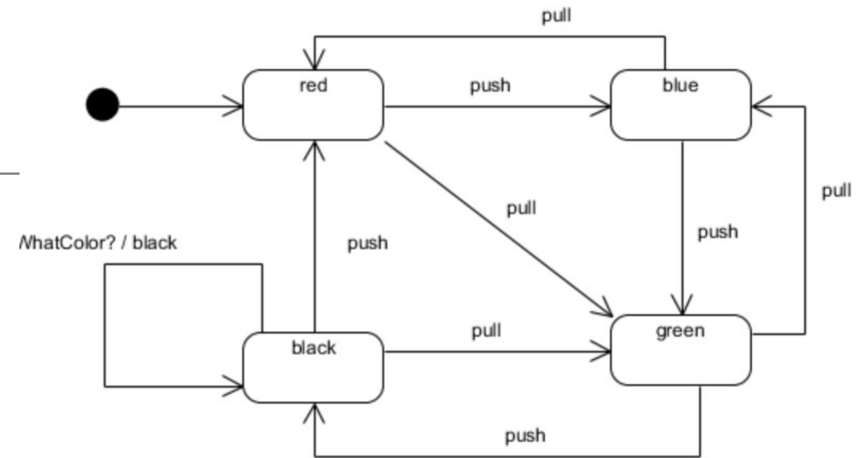
```
    // Creates a new Context with the default state.
public Context() {this(new RedState());}
    // Returns the state.
public State getState() {return state;}
    // Sets the state.
public void setState(State state) {this.state = state;}
```

State Pattern Example 1 (Continued)

```
/**
* The push() method performs different actions depending
* on the state of the object. Using the State pattern,
* we delegate this behavior to our contained state object.
*/
public void push() {state.handlePush(this);}

/**
* The pull() method performs different actions depending
* on the state of the object. Using the State pattern,
* we delegate this behavior to our contained state object.
*/
public void pull() {state.handlePull(this);}
}
```

Ex1 with State pattern

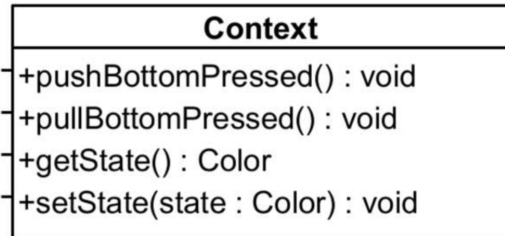


{state.handlePush();}

{state.handlePull();}

{state.getState();}

{this.state = state;}



context

state

Color

+handlePush()
+handlePull()

GreenState

+handlePush()
+handlePull()
+getState() : Color

BlueState

+handlePush()
+handlePull()
+getState() : Color

RedState

+handlePush()
+handlePull()
+getState() : Color

BlackState

+handlePush()
+handlePull()
+getState() : Color

{context.setState(new BlackState);}

{return this;}

Osservazioni

1. L'invocazione dei metodi push() e pull() porta solo a una transizione di stato e nessun'altra azione viene eseguita.
   - Questo dipende dall'esempio scelto e potrebbe non valere in altri casi

2. Gli stati concreti definiscono la transizione di stato
   - Conoscono il loro "next state"
   - Soluzione generalmente più flessibile, anche se causa dipendenze tra le classi ConcreteState

Potrebbe essere il context a cambiare stato
   - Nelle situazioni semplici

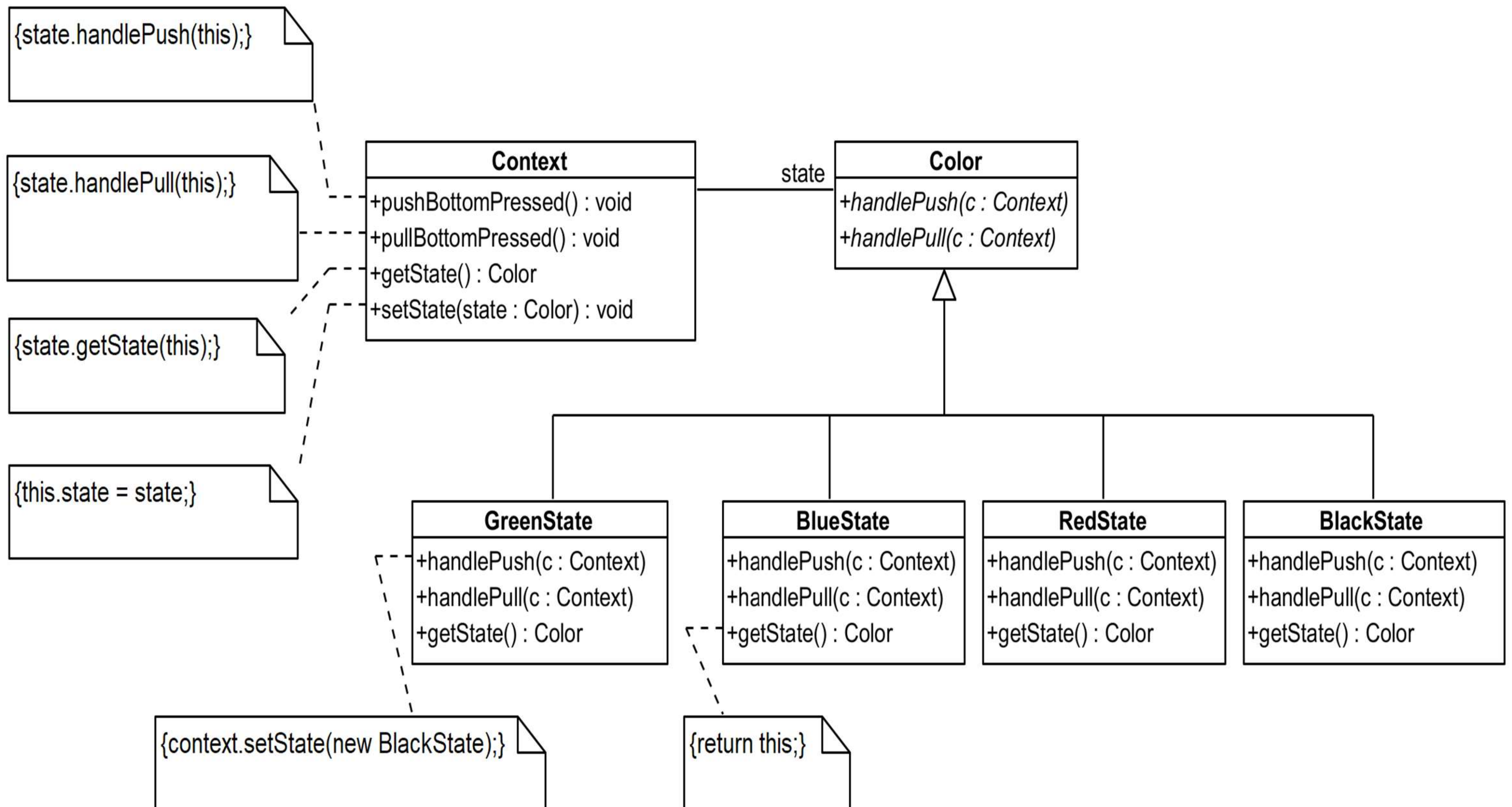
Osservazioni

3. Ogni volta che si cambia stato si crea un nuovo oggetto di tipo Stato
   - Questa è una scelta: creare oggetti State solo quando sono necessari e distruggerli in seguito.
   - Ok se i cambiamenti di stato sono poco frequenti

   Il pattern prevede anche il caso in cui gli oggetti Stato sono creati in anticipo e non distrutti mai
   - Ok se i cambiamenti di stato sono frequenti

4. Gli oggetti di tipo Color hanno una variabile (final) context
   - Anche questa è una scelta
   - Context può essere passato per riferimento

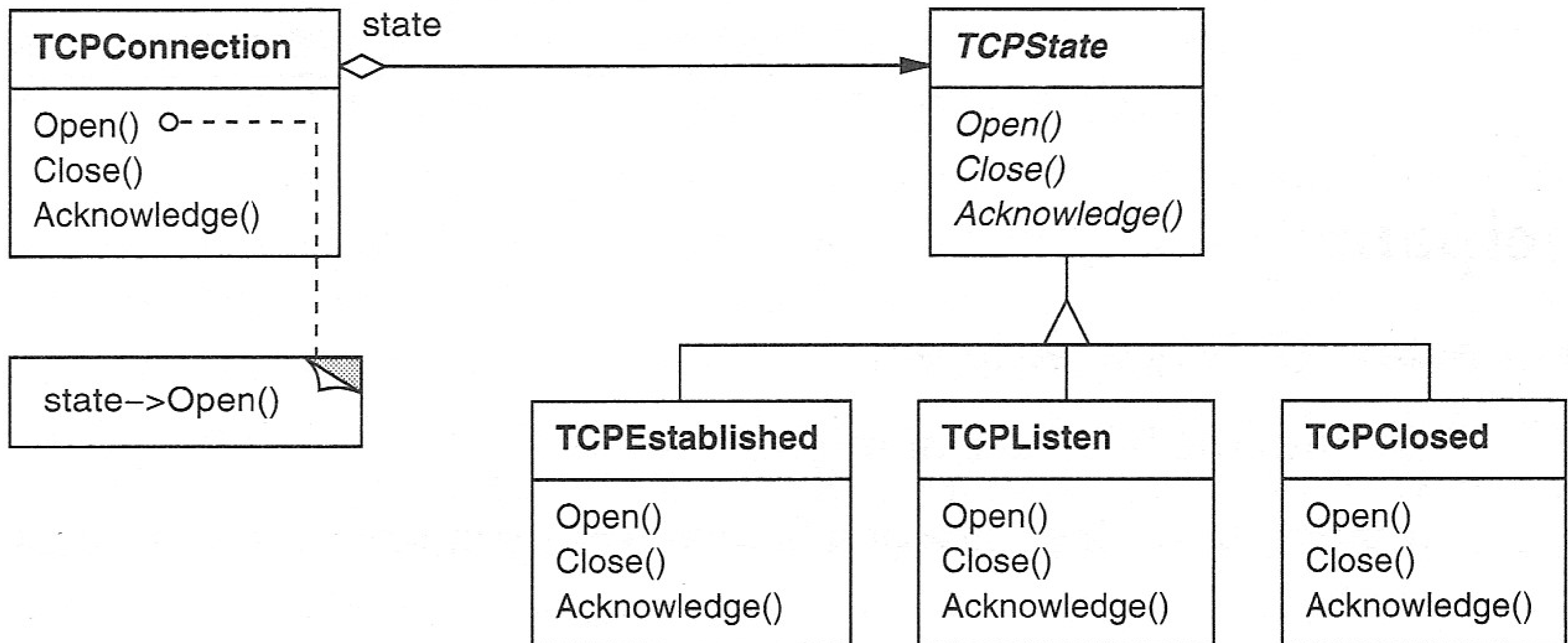
Variante: context passato come argomento



Example: TCP

TCP State is abstract and defines interface

Each subclass of TCPState represents a state



State: interface or class?

State (TCPState)

- Can be a concrete class giving a default implementation:

```
void open( TCPConnection t) { }
void close ( TCPConnection t) { }
void acknowledge( TCPConnection t, TCPOctetStream os) { }
```

ConcreteState (TCPEstablished, TCPListen, TCPClosed)

- Each subclass implements state-specific behavior for valid requests in the state

```
class TCPClosed extends TCPState{
    open( TCPConnection t ) {
        // send SYN, receive SYN, ACK, etc.
        ChangeState( t, new TCPEstablished: );
        }
}
```

Applicabilità

Utilizzare lo State pattern in uno dei seguenti casi:

- Il comportamento di un oggetto dipende dal suo stato e deve cambiare il suo comportamento a run-time in base a tale stato.
- Le operazioni hanno dichiarazioni condizionali complesse che dipendono dallo stato dell'oggetto.

Applicazione del pattern i 6 passi

1. Identificare una classe esistente o crearne una nuova che funga da "macchina a stati" dal punto di vista del cliente. Questa classe è la classe "Context".
2. Creare una classe base State che replichi i metodi dell'interfaccia della macchina a stati. Ogni metodo richiede un parametro aggiuntivo: un'istanza della classe Context. La classe State specifica qualsiasi comportamento utile "predefinito".
3. Creare una classe derivata State per ogni stato del dominio. Queste classi derivate sovrascrivono solo i metodi che devono sovrascrivere.
4. La classe Context mantiene un oggetto State "corrente".
5. Tutte le richieste del client alla classe Context sono semplicemente delegate all'oggetto State corrente e viene passato il puntatore this dell'oggetto Context.
6. I metodi State modificano lo stato "corrente" dell'oggetto Context, come specificato

Discussione

- State localizza il comportamento che dipende dallo stato e lo suddivide tra i diversi stati.
- Tutti i comportamenti specifici di uno stato sono memorizzati in una classe
- L'alternativa è costituita da case statement giganti
- Può produrre un gran numero di classi, ma è meglio dell'alternativa
- Rende esplicite le transizioni di stato
- Lo stato corrente è memorizzato in un'unica posizione
- Gli oggetti State possono essere condivisi
  - Quando non hanno variabili di istanza

Implementazione

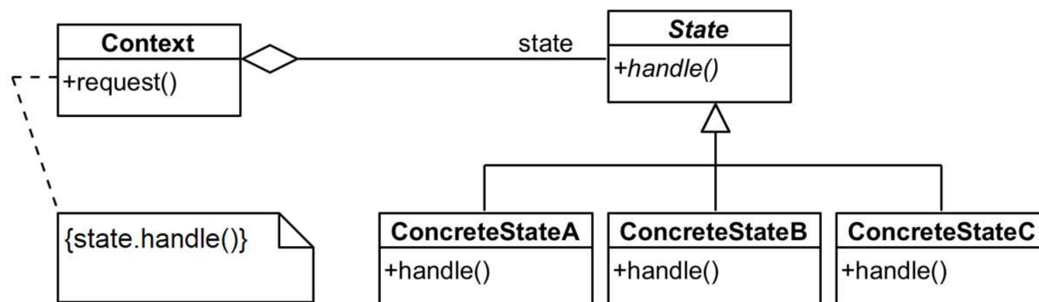
- Non possiamo usare una tabella di transizione degli Stati per tutto questo?
  - Più difficile da capire
  - Difficile aggiungere altre azioni e comportamenti

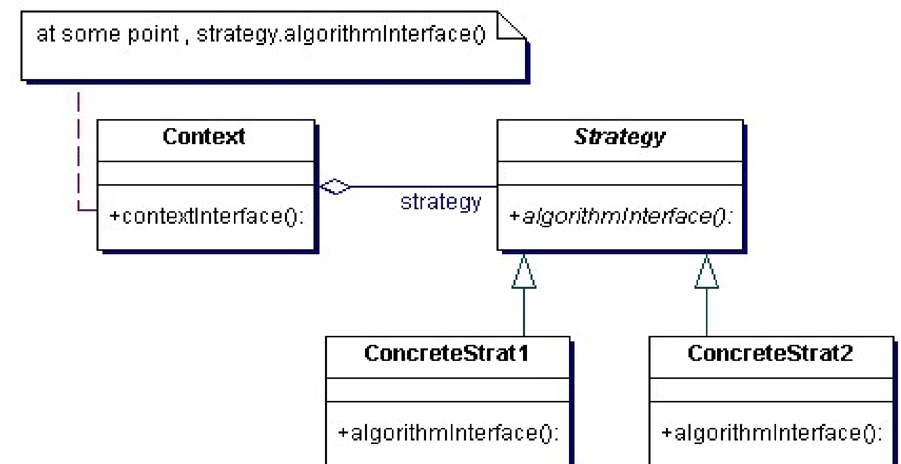
- State può essere una classe astratta, se c’è del comportamento comune o di default

State vs Strategy

Similitudini tra i patterns State e Strategy!

- Ad esempio, sono entrambi esempi di composizione con delega.
- La differenza sta nell'intento.
  - Un oggetto State incapsula un comportamento dipendente dallo stato (ed eventualmente transizioni di stato).
  - Un oggetto Strategy incapsula un algoritmo.
  - In State un oggetto Context cambia stato in base a ben precise transizioni di stato.





Homework: Complete the colours example

Implement different variants, it accordingly to the following implementation startegies:

1. Create State objects ahead of time and never destroy them
2. Letting Context to decide the flow of state transitions.

Notice that there are two different kinds of functionalities to deal with

```
// handle state change: you can have different strategies here
handlePush(Context c) {…} and handlePull(Context c) {…}

// handle a request that depends on the state: no variability in
   implementation, states deal with it
public Color getColor() {…}
```