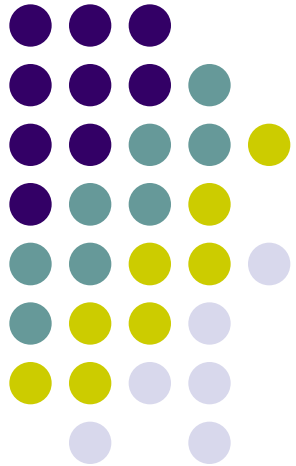
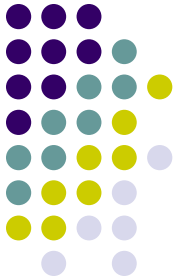
STATE Pattern

By :
Atri Chatterjee
Souvik Ghosh

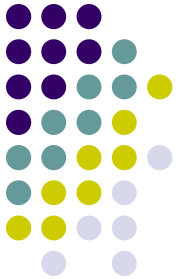


General Description



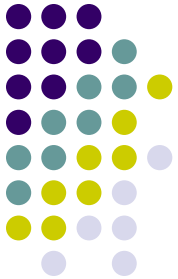
- A type of Behavioral pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Uses Polymorphism to define different behaviors for different states of an object.

State Design Pattern



- Encapsulate states as concrete objects.
- Abstract them into an abstract class.
- The system contains a reference to an abstract state class, which can be any of the concrete state classes dynamically.

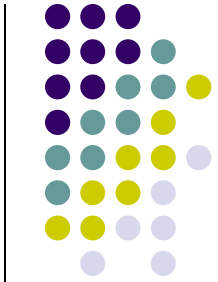
When to use STATE pattern ?



- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state.
- To simplify operations that have large conditional statements that depend on the object's state.

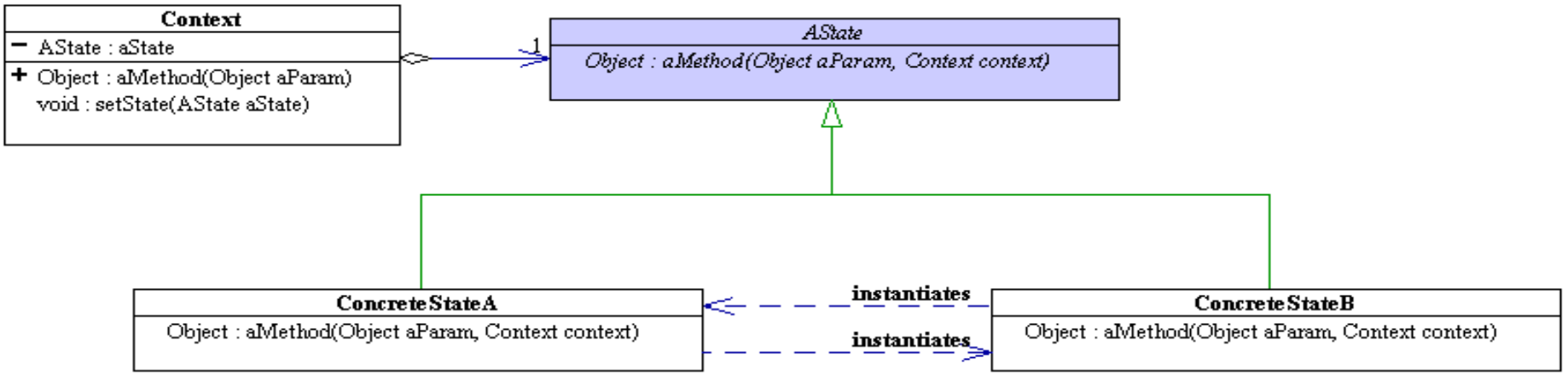
```
if (myself = happy) then
{
    eatIceCream();
    ….
    }
else if (myself = sad) then
{
    goToFriend();
    ….
  }
else if (myself = ecstatic) then
{
    ….
```

Applicability

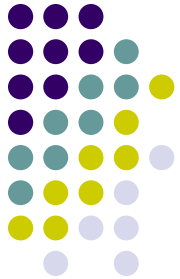


- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. This state usually represented by one or more enumerated constants. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

Structure

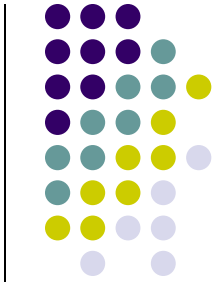

Context
− AState : aState
+ Object : aMethod(Object aParam)
void : setState(AState aState)
1
AState
Object : aMethod(Object aParam, Context context)
ConcreteStateA
Object : aMethod(Object aParam, Context context)
instantiates
instantiates
ConcreteStateB
Object : aMethod(Object aParam, Context context)

Participants



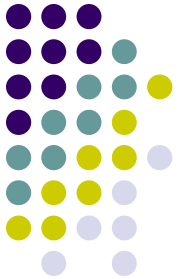
- Context
  - Defines the interface of interest to clients.
  - Maintains an instance of a ConcreteState subclass that defines the current state.
- State
  - Defines an interface for encapsulating the behavior associated with a particular state of the Context.
- ConcreteState subclasses
  - Each subclass implements a behavior associated with a state of the Context.

Collaborations



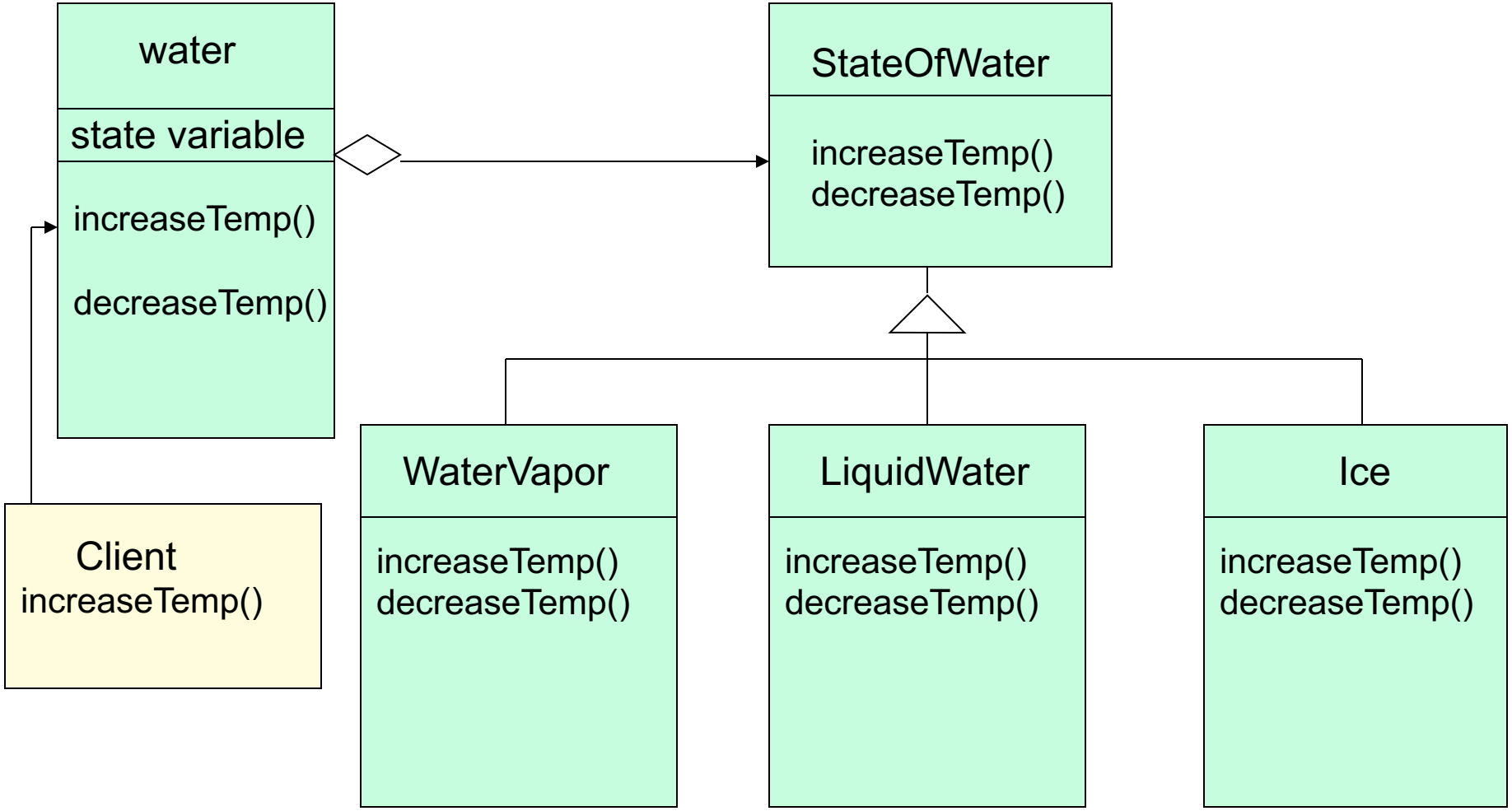
- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

Consequences

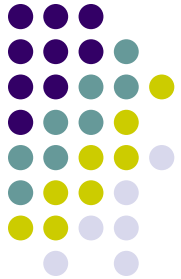


- It localizes state-specific behavior and partitions behavior for different states.
- It makes state transitions explicit.
- State objects can be shared.

Example I


water
state variable
increaseTemp()
decreaseTemp()
StateOfWater
increaseTemp()
decreaseTemp()
Client
increaseTemp()
WaterVapor
increaseTemp()
decreaseTemp()
LiquidWater
increaseTemp()
decreaseTemp()
Ice
increaseTemp()
decreaseTemp()

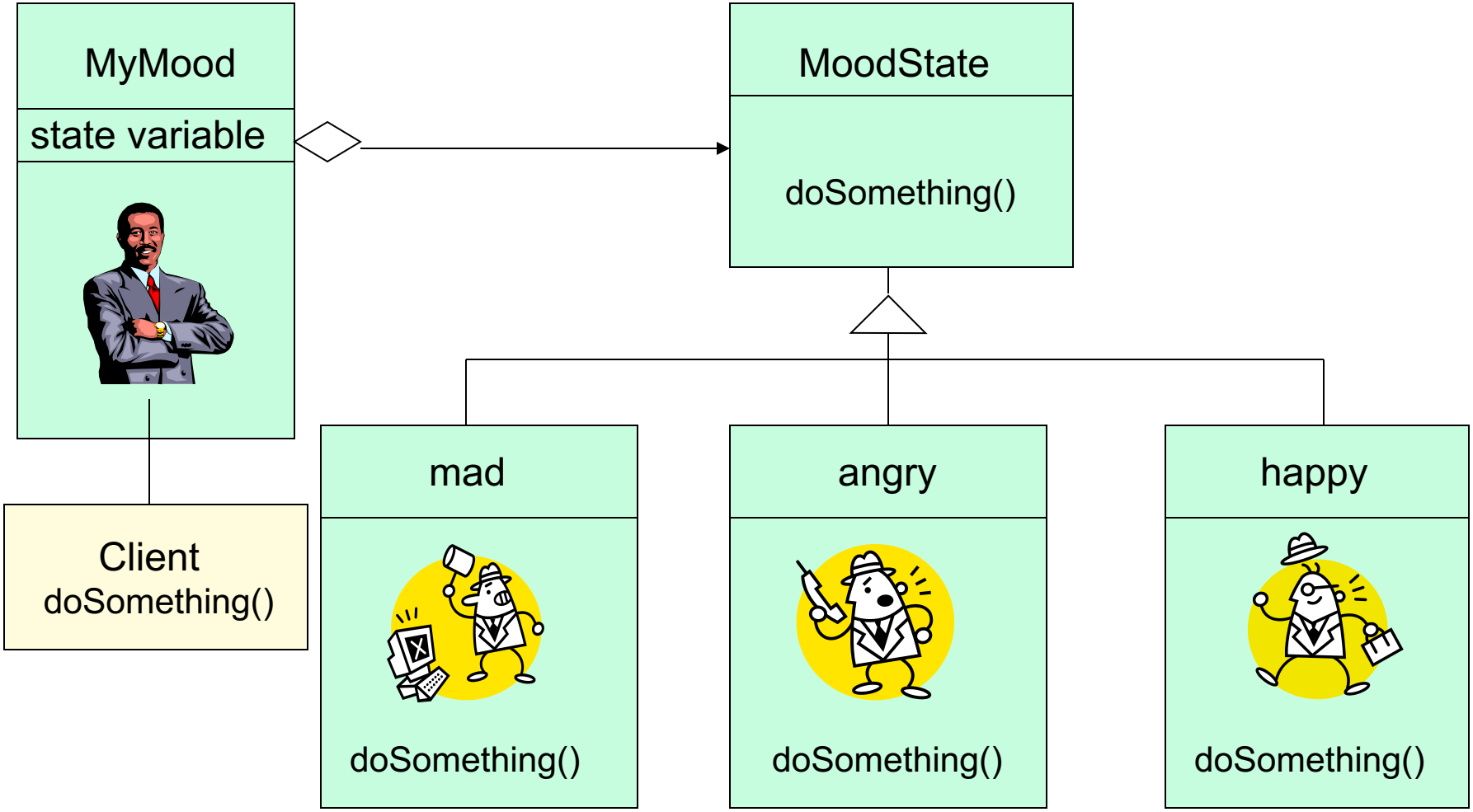
How is STATE pattern implemented ?



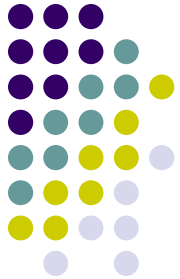
- "Context" class:
  Represents the interface to the outside world.
- "State" abstract class:
  Base class which defines the different states of the "state machine".
- "Derived" classes from the State class:
  Defines the true nature of the state that the state machine can be in.

Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed.

Example II


MyMood
state variable
MoodState
doSomething()
Client
doSomething()
mad
doSomething()
angry
doSomething()
happy
doSomething()

Benefits of using STATE pattern

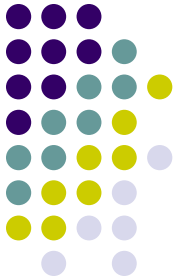


- **Localizes all behavior associated with a particular state into one object.**
  - New state and transitions can be added easily by defining new subclasses.
  - Simplifies maintenance.

- **It makes state transitions explicit.**
  - Separate objects for separate states makes transition explicit rather than using internal data values to define transitions in one combined object.

- **State objects can be shared.**
  - Context can share State objects if there are no instance variables.

Food for thought…



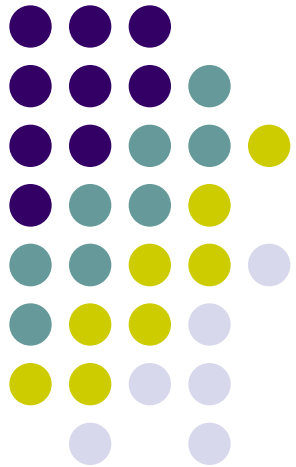
- **To have a monolithic single class or many subclasses ?**
  - Increases the number of classes and is less compact.
  - Avoids large conditional statements.

- **Where to define the state transitions ?**
  - If criteria is fixed, transition can be defined in the context.
  - More flexible if transition is specified in the State subclass.
  - Introduces dependencies between subclasses.

- **Whether to create State objects as and when required or to create-them-once-and-use-many-times ?**
  - First is desirable if the context changes state infrequently.
  - Later is desirable if the context changes state frequently.

The State Design Pattern Example

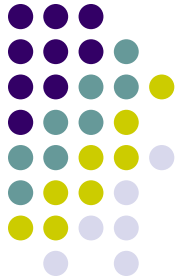
A behavioral design pattern.



Shivraj Persaud

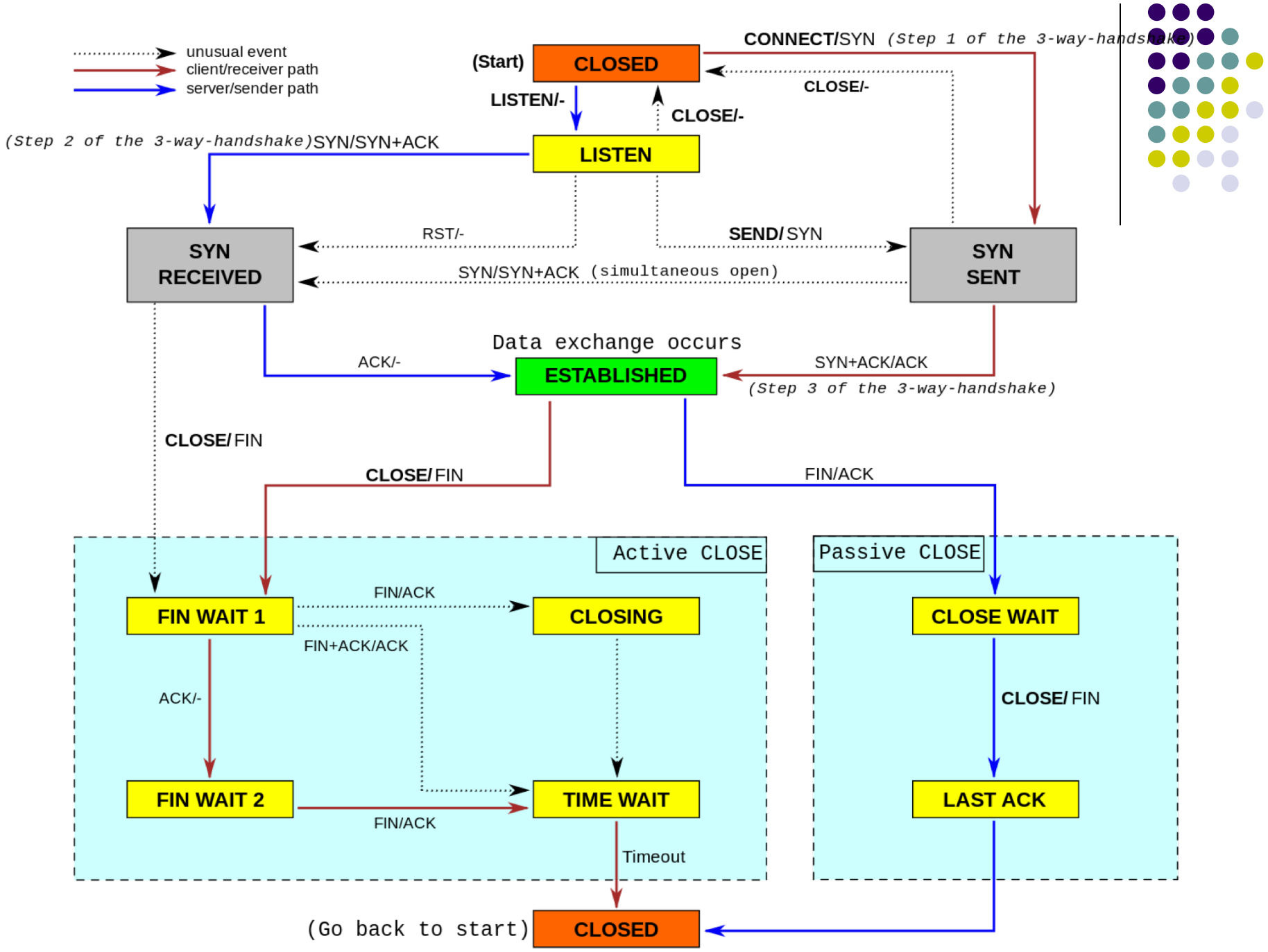
E-mail: persas@rpi.edu

Example – TCPConnection class

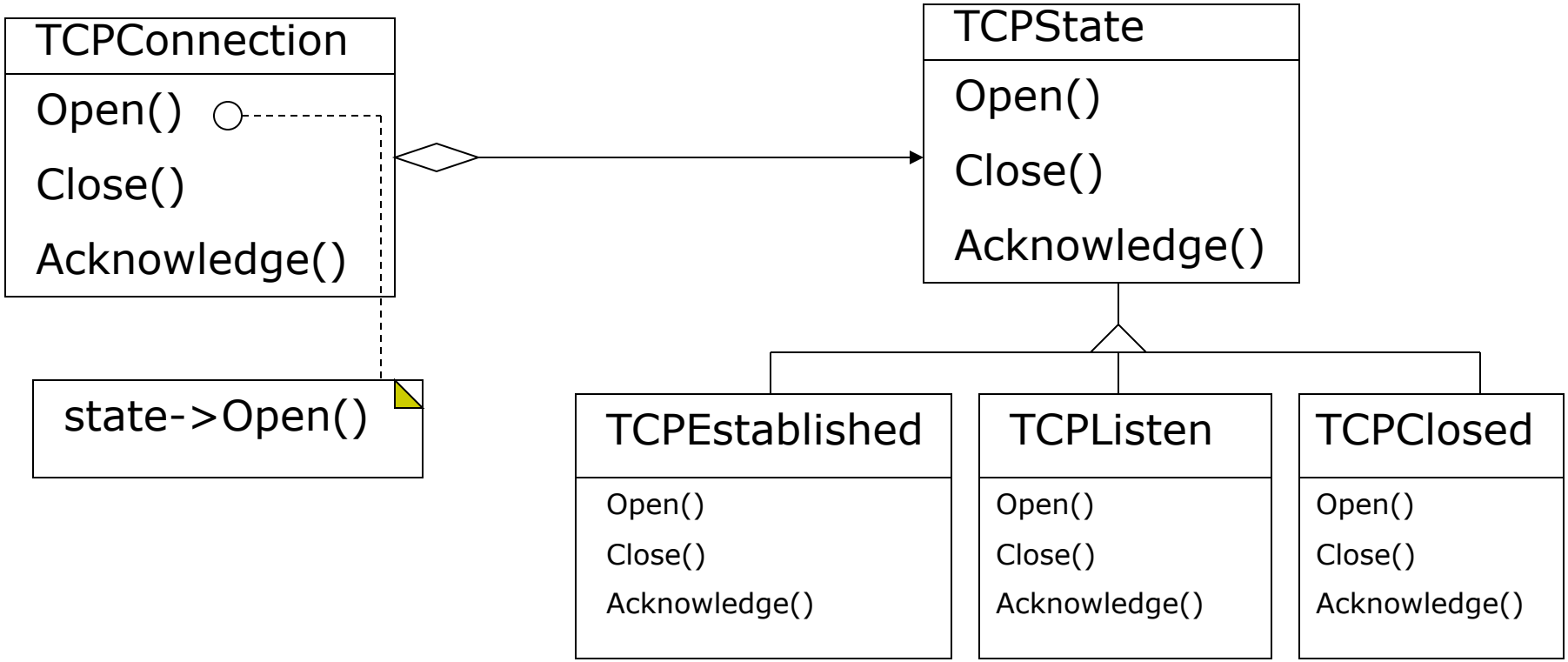


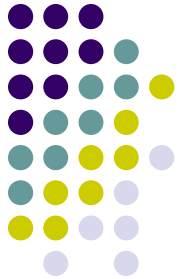
- A TCPConnection object can be in one of several different states, established, listening or closed.

- The behavior of TCPConnection is different in each of these three states.


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake)SYN/SYN+ACK
LISTEN
RST/-
SEND/ SYN
SYN RECEIVED
SYN/SYN+ACK (simultaneous open)
SYN SENT
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/ FIN
CLOSE/ FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
FIN+ACK/ACK
CLOSE WAIT
ACK/-
CLOSE/ FIN
FIN WAIT 2
TIME WAIT
FIN/ACK
LAST ACK
Timeout
(Go back to start)
CLOSED

Abstract state class

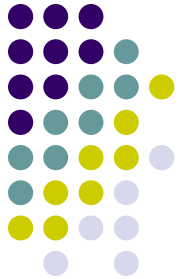




- TCPConnection maintains a state object which is a subclass of TCPState.

- When the connection changes state the TCPConnection object changes the state object it uses.

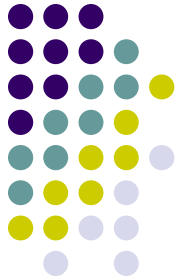
Participants



Context (TCPConnection)

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

Participants



State (TCPState)

- defines an interface for encapsulating the behavior associated with a particular state of the Context.

ConreteState Subclasses (TCPEstablished, TCPListen, TCPClosed)

- each subclass implements a behavior associated with a state of the Context.