AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Architecture and Design of Embedded Real-Time Systems (TI-AREM)

GoF State Pattern
a Behavioral Pattern

Version: 5-2-2015

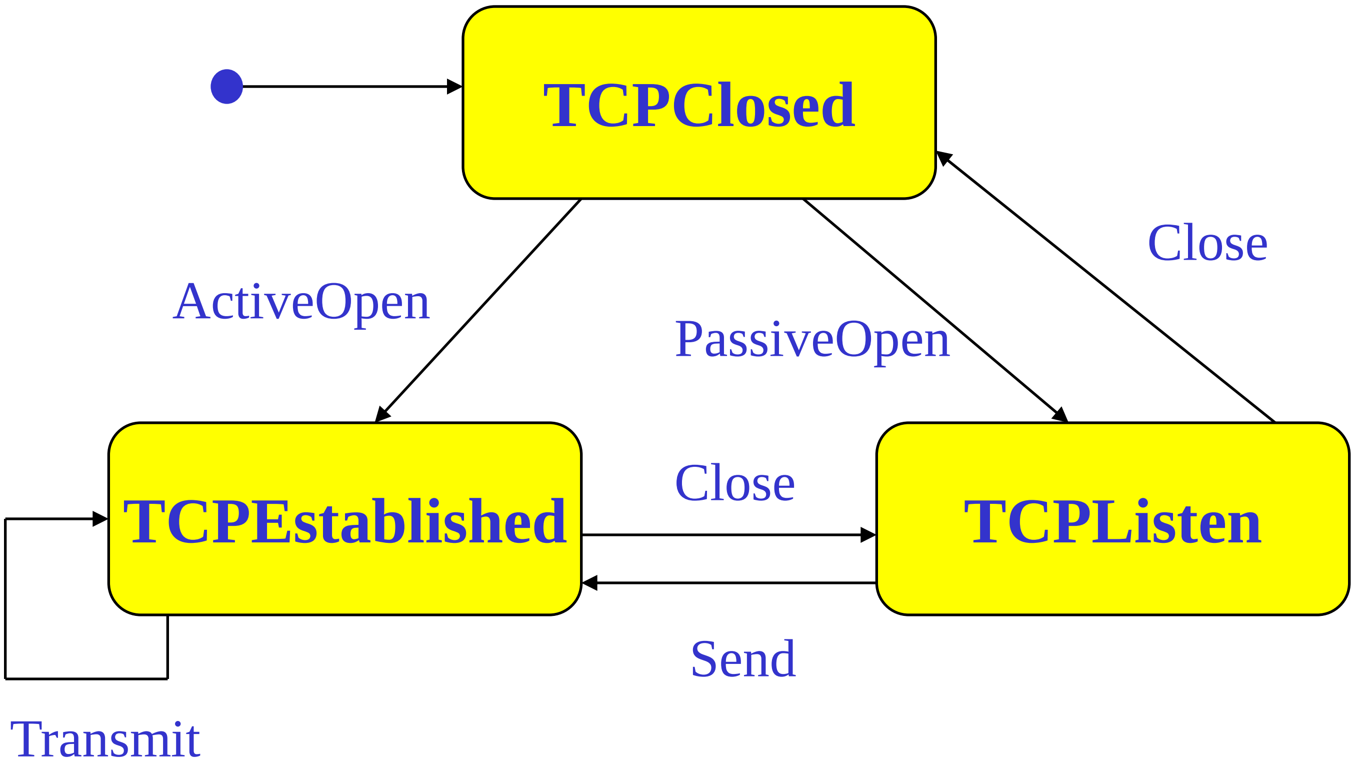
State Pattern – Behavioral Pattern

Intent:

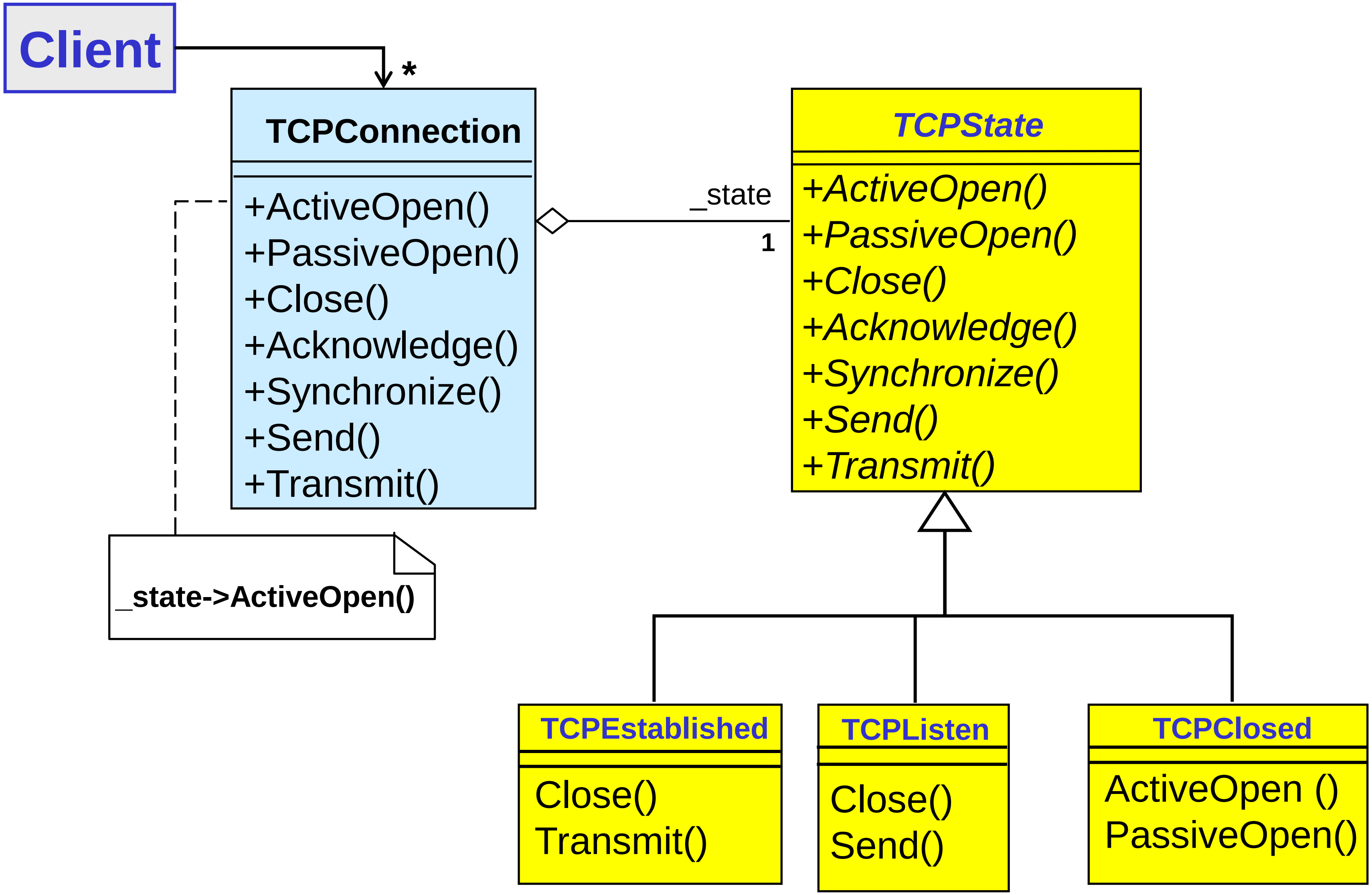
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class (its state).

An implementation technique for realizing a state machine described by a UML State Diagram. Each state is implemented as a class with the events as operations.

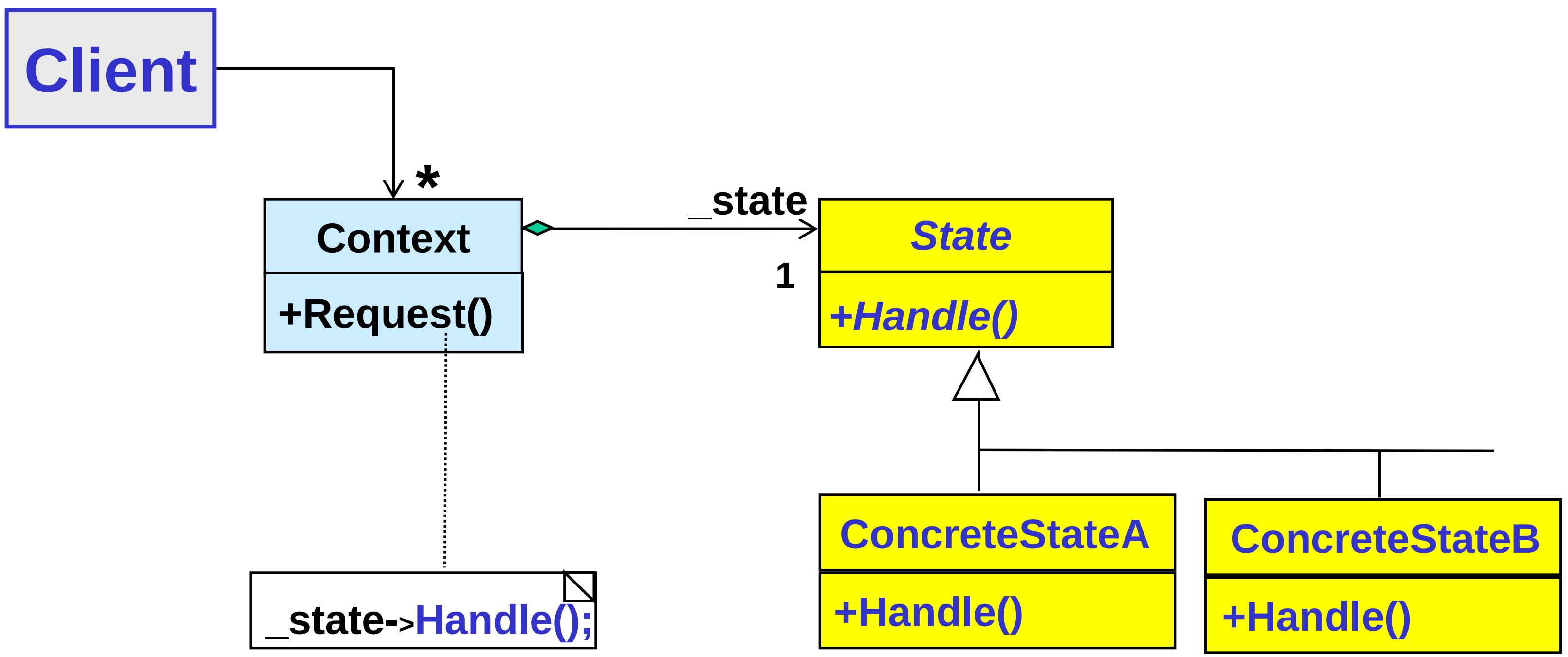
TCP State Diagram Example



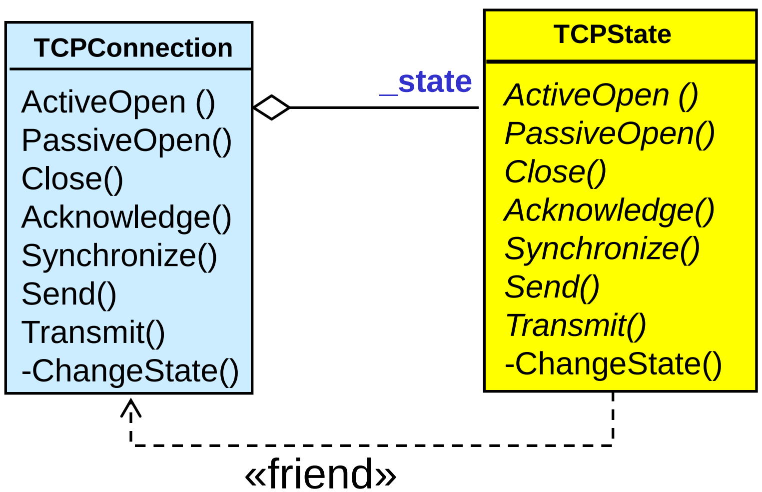
State Pattern Class Diagram



State Pattern - GoF Structure



State Pattern – C++ Example (1)




```
class TCPConnection
{
public:
    TCPConnection(); // constructor
    void ActiveOpen();
    void PassiveOpen();
    void Close();
    void Acknowledge();
    void Synchronize();
    void Send();
    void Transmit();
    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
    TCPState* _state;
};
```

State Pattern – C++ Example (2)

```
TCPConnection::TCPConnection ()            { _state = TCPClosed::Instance(); }

void TCPConnection::ActiveOpen ()          { _state->ActiveOpen(this); }

void TCPConnection::PassiveOpen ()         { _state->PassiveOpen(this); }

void TCPConnection::Close ()               { _state->Close(this); }

void TCPConnection::Acknowledge ()         { _state->Acknowledge(this); }

void TCPConnection::Synchronize ()         { _state->Synchronize(this); }

void TCPConnection::ChangeState (TCPState* s) {  _state = s; }
```

State Pattern – C++ Example (3)

Notice!

```
class TCPState
{
public:
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Send(TCPConnection*);
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
protected:
    TCPState() { }
    void ChangeState(TCPConnection*, TCPState*);
};
```

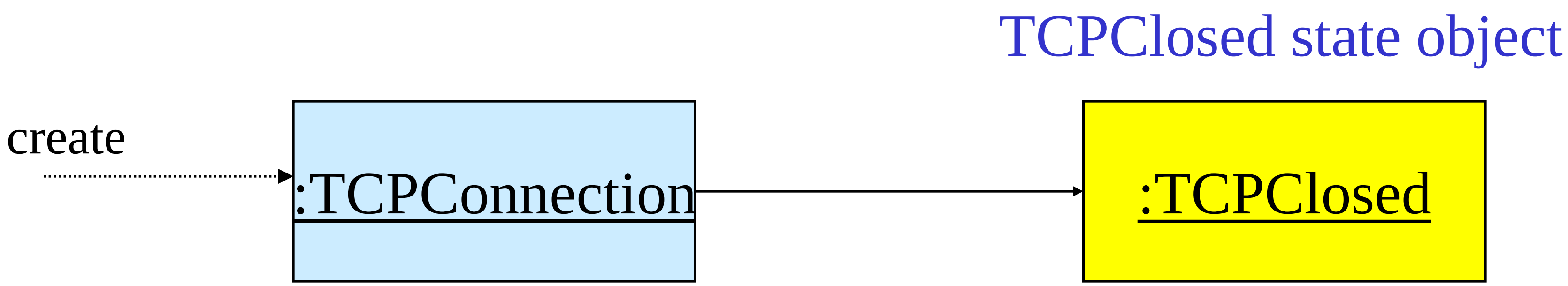
State Pattern – C++ Example (4)

Implementation of **TCPState** with **default code**:

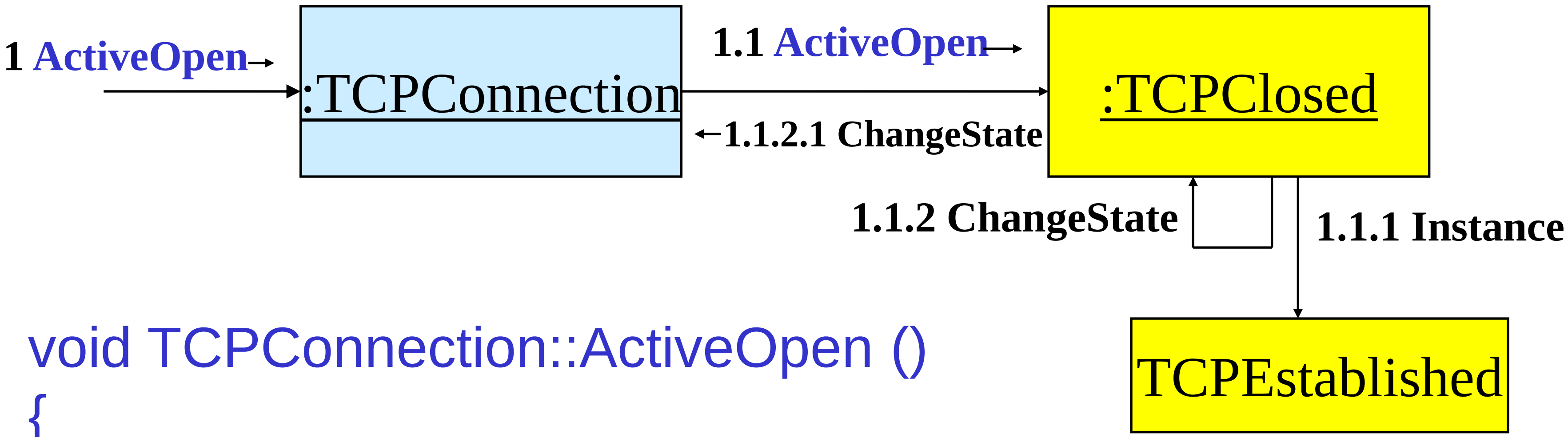
```
void TCPState::ActiveOpen (TCPConnection*)      { default(); }
void TCPState::PassiveOpen (TCPConnection*)     { default(); }
void TCPState::Close (TCPConnection*)           { default(); }
void TCPState::Acknowledge (TCPConnection*)     { default(); }
void TCPState::Synchronize (TCPConnection*)     { default(); }
void TCPState::Send (TCPConnection*)            { default(); }
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { default();}

void TCPState::ChangeState (TCPConnection* t, TCPState* s)
{
    t->ChangeState(s);
}
```

State Pattern – C++ Example (5)

```
TCPConnection::TCPConnection ()
{
    _state = TCPClosed::Instance();     // start state
}
```



State Pattern – C++ Example (6)




```
void TCPConnection::ActiveOpen ()
{
   _state->ActiveOpen(this);
}

void TCPClosed::ActiveOpen (TCPConnection* t)
{   // send SYN, receive SYN, ACK, etc.
   ChangeState(t, TCPEstablished::Instance());   // state shift
}
```

State Pattern – C++ Example (7)

```
void TCPClosed::PassiveOpen (TCPConnection* t)
{
    ChangeState(t, TCPListen::Instance());
}


void TCPEstablished::Transmit (TCPConnection* t, TCPOctetStream* o)
{
   t->ProcessOctet(o);              // no state change
}


void TCPEstablished::Close (TCPConnection* t)
{
   // send FIN, receive ACK of FIN
   ChangeState(t, TCPListen::Instance());
}
```

State Pattern – C++ Example (8)

Singleton pattern

```
class Singleton
{
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance ()
{
    if (_instance == 0)
   {
        _instance = new Singleton;
    }
    return _instance;
}
```

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

State Pattern – C++ Example (9)

Singleton pattern used on TCPClosed class

```
class TCPClosed : public TCPState
{
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
 protected:
    TCPClosed();
 private:
    static TCPState* _instance;
}
```

```
TCPState*
       TCPClosed::_instance = 0;

TCPState* TCPClosed::Instance()
{
     if (_instance = = 0)
    {
       _instance = new TCPClosed;
     }
    return _instance;
}
```

State Pattern Implementation Details (1)

TCPState
ActiveOpen () *PassiveOpen()* *Close()* *Acknowledge()* *Synchronize()* *Send()* *Transmit()*

1. **Design choice 1.**
   - **All event operations specified as pure virtual (C++)**
   - **Requires that all operations are defined in the subclasses (forced by the compiler)**
2. **Design choice 2.**
   - **All event operations have default implementation in superclass**
   - **Only necessary to implement event operations for the actual state**

State Pattern Implementation Details (2)

Event parameters, guard, entry and exit actions

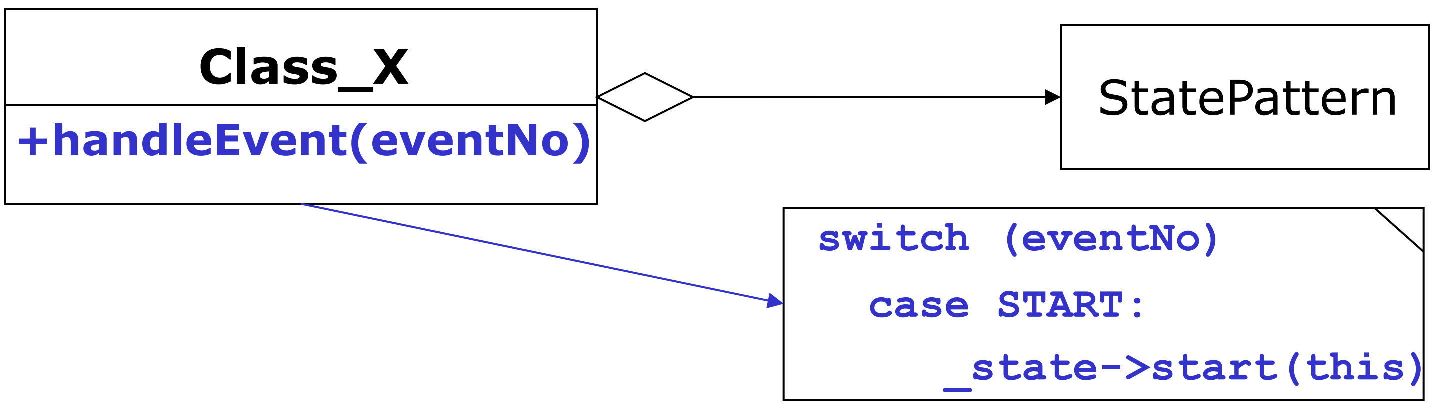
```
void TCPClosed::event1(TCPConnection* t,type parameter1)
{
    if ( t->guard1() )
    {
        exit();  // explicit call of exit action
        t->action_1();
        ChangeState(t,TCPListen::Instance());
    }
}
```

```
void TCPConnection::ChangeState(State* pS)
{
    _state= pS;
    _state->entry(this); // call “entry” in new state
}
```

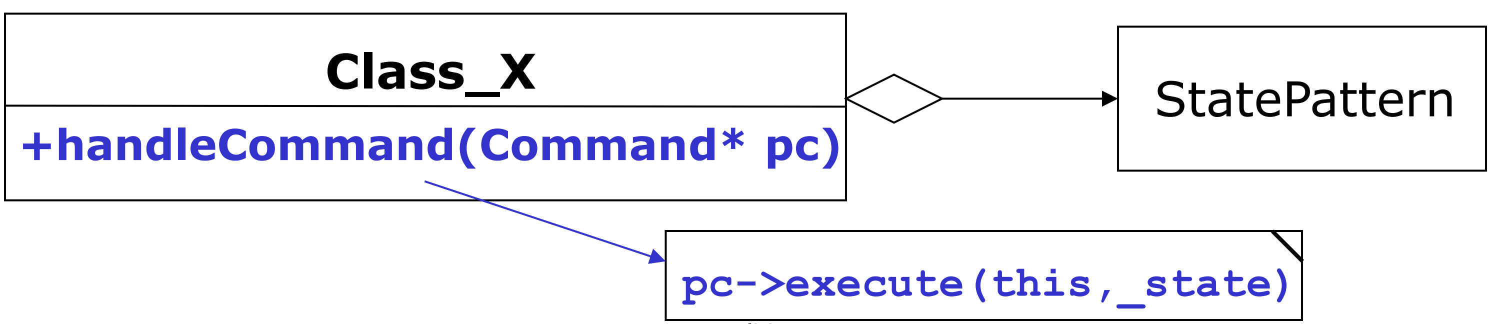
Three Solutions for Event Handling

1.

2.

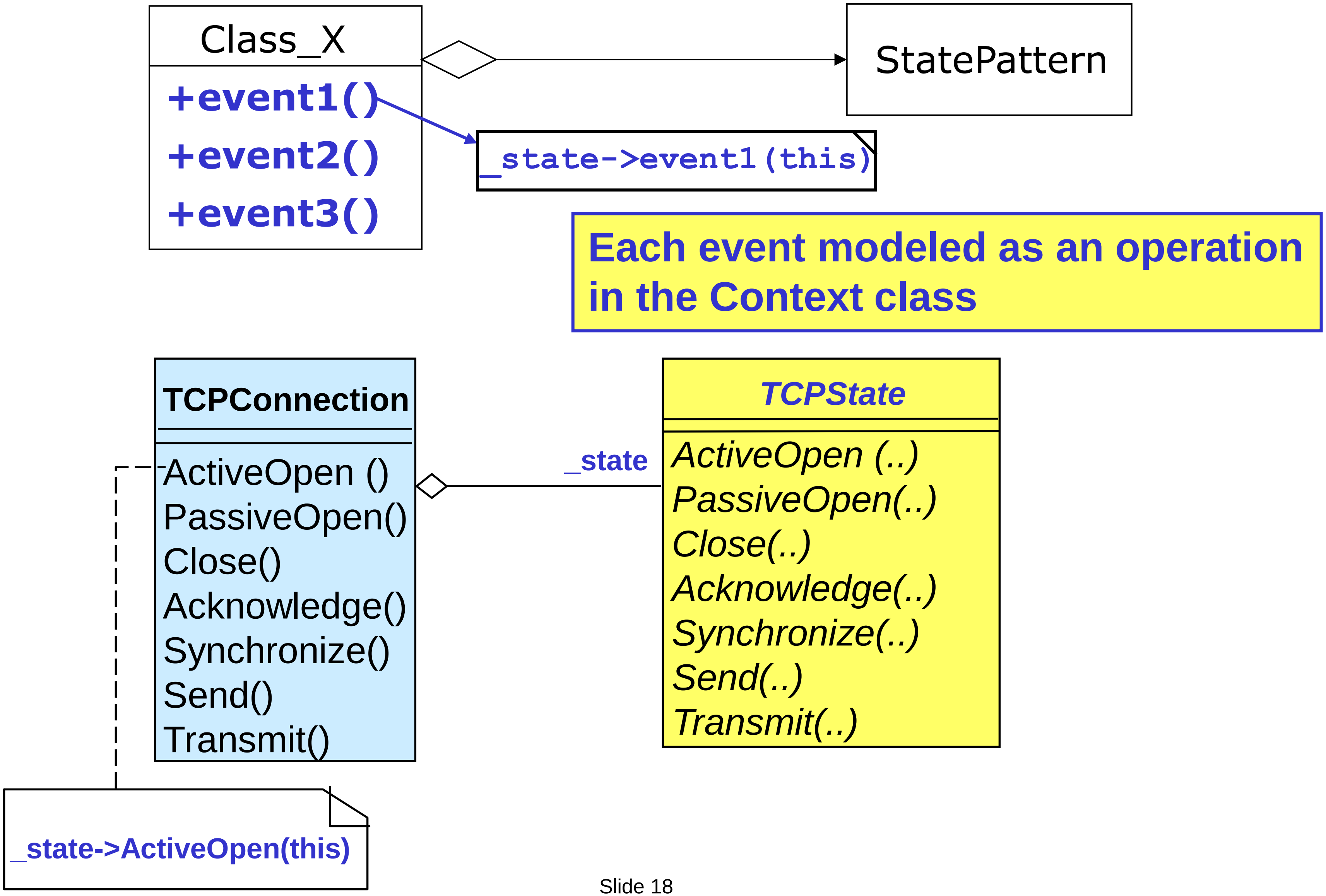


3.



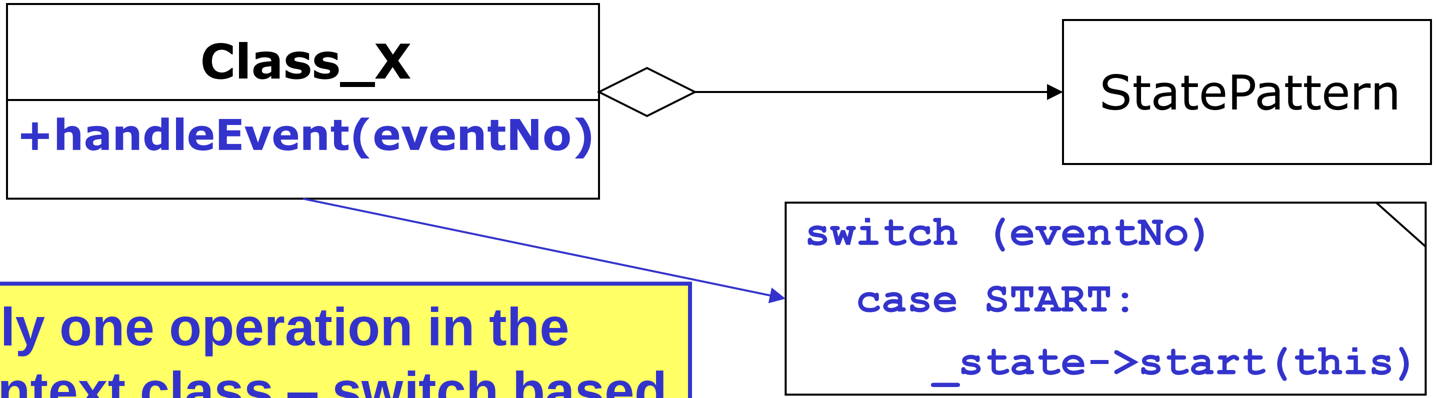
AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Event Handling - Solution 1.

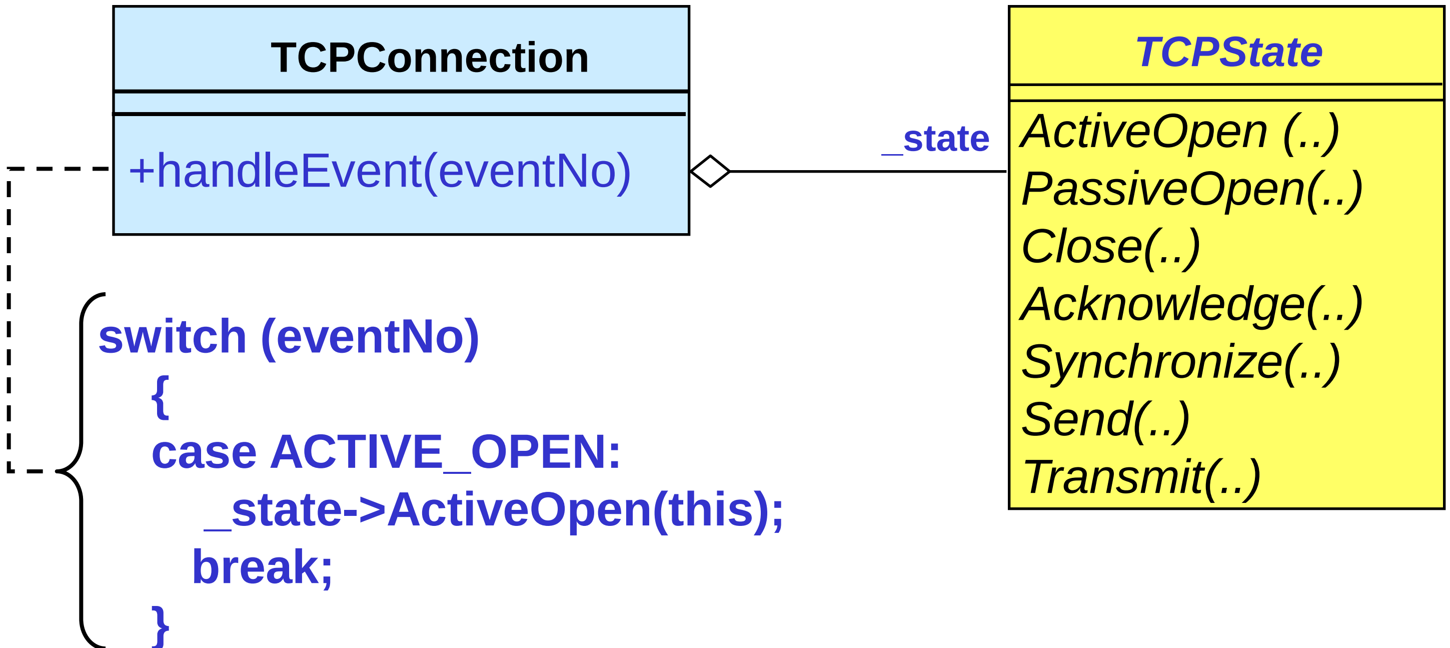


AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

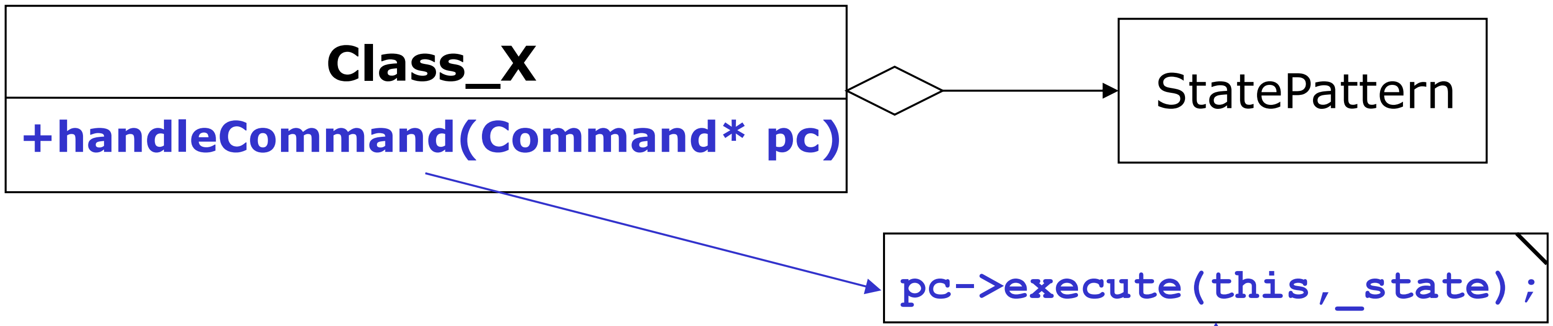
Event Handling - Solution 2.



Only one operation in the Context class – switch based



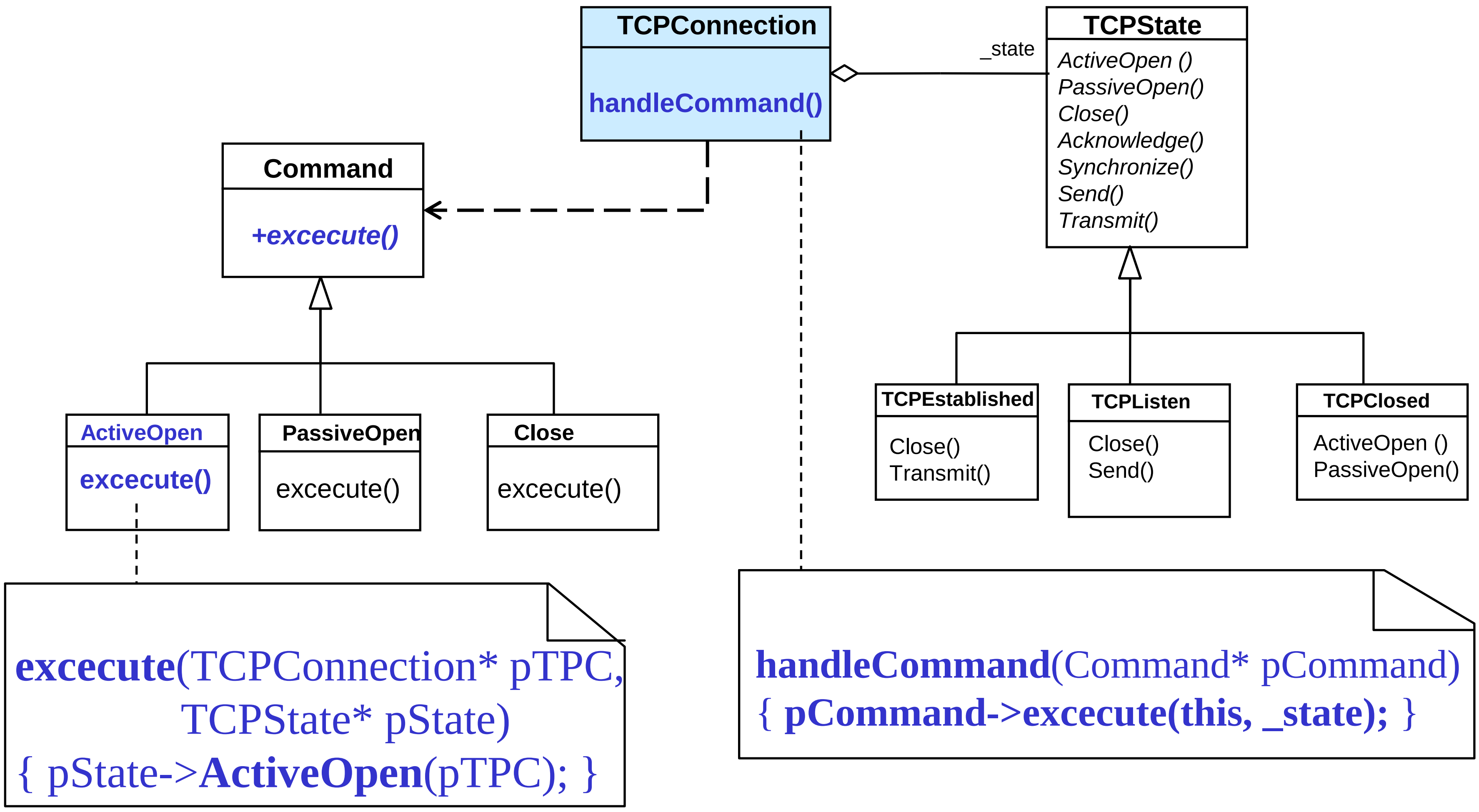
Event Handling - Solution 3.



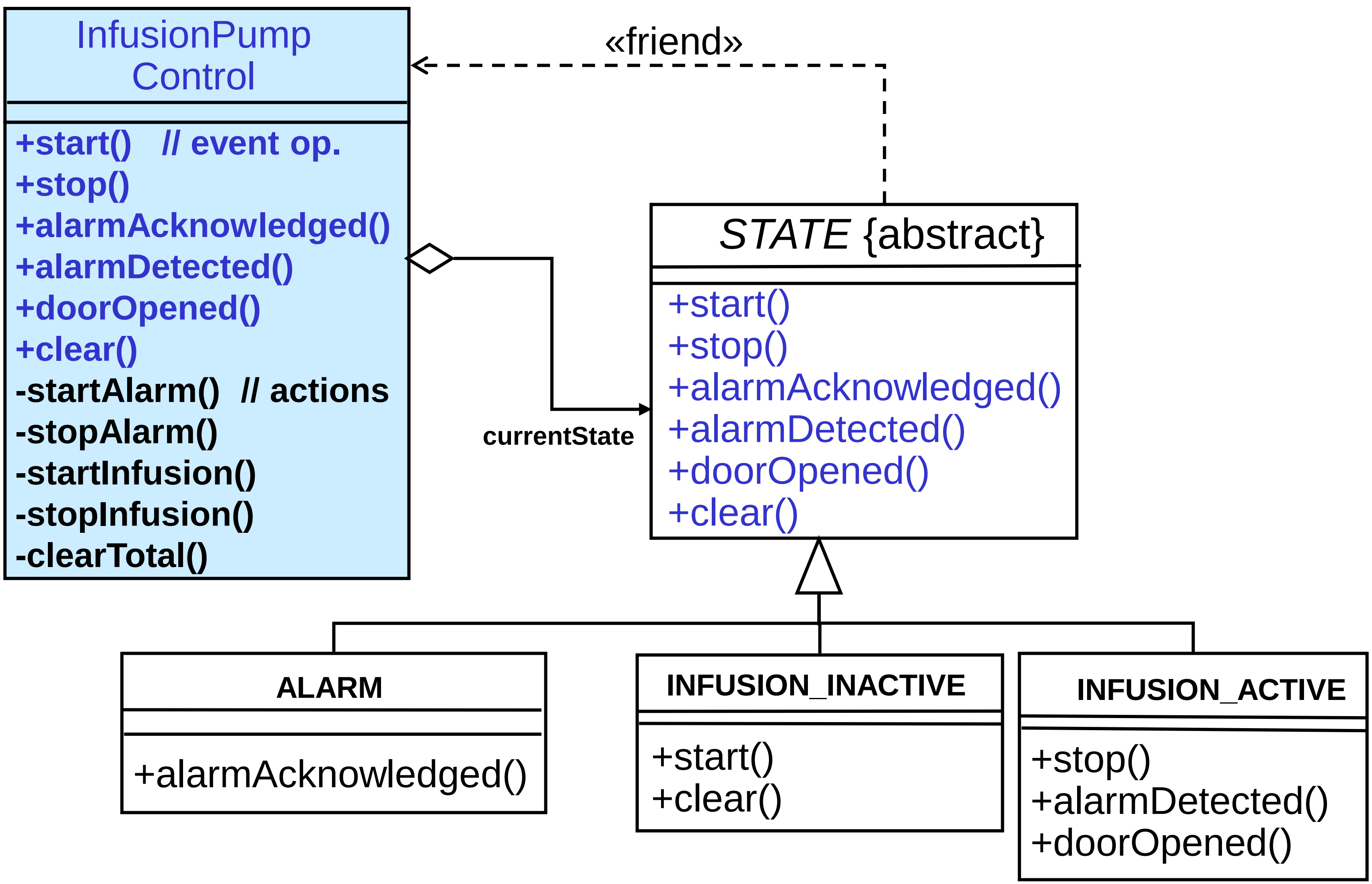
Only one operation in the Context class – polymorphic based

An example of the Command Pattern used in concert with the State Pattern

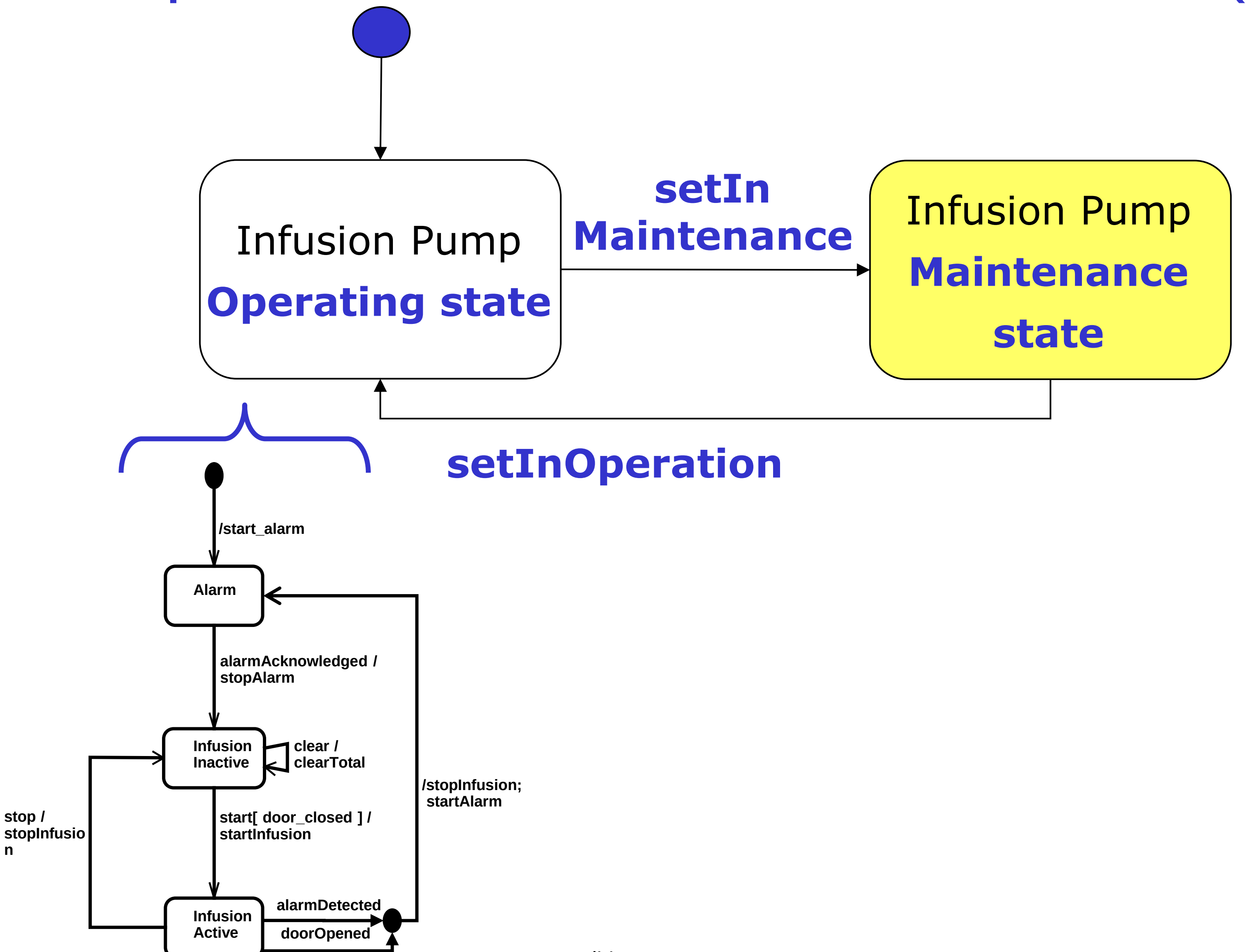
Command (GoF) & State Pattern (C++)



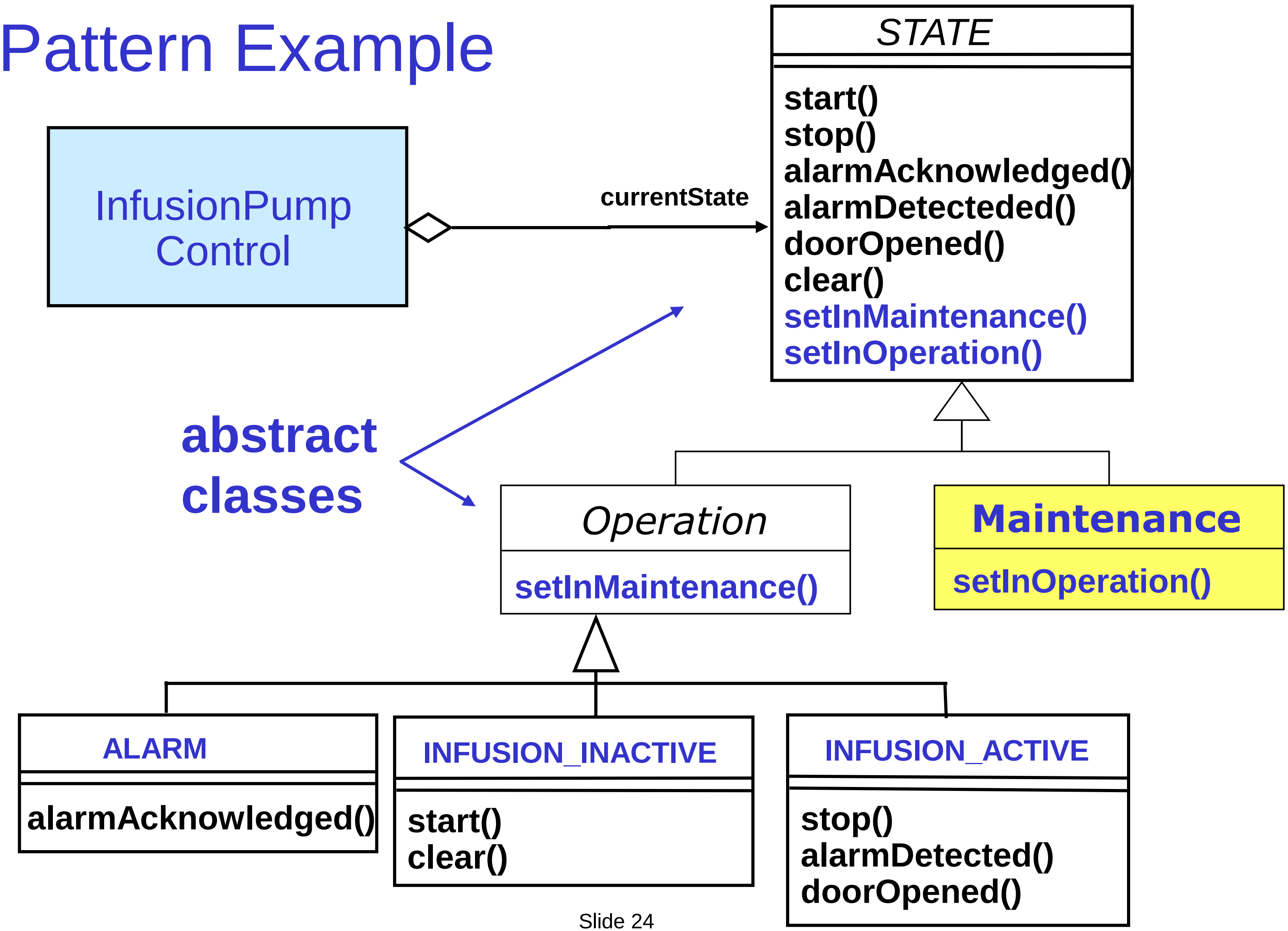
State Pattern – Infusion Pump Example



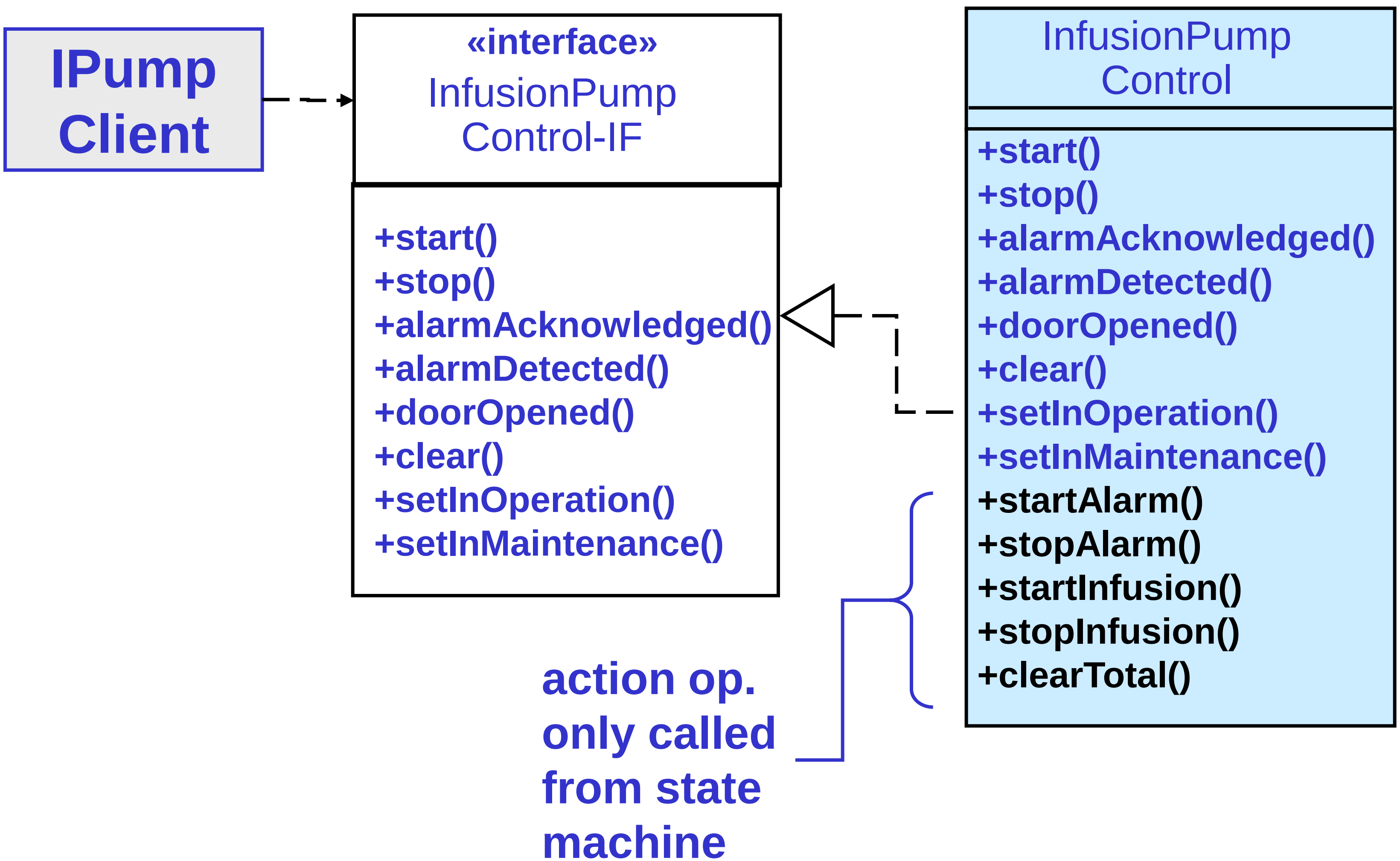
Example with Hierarchic State Machine (1)



Hierarchic State Pattern Example



Definition of Interface

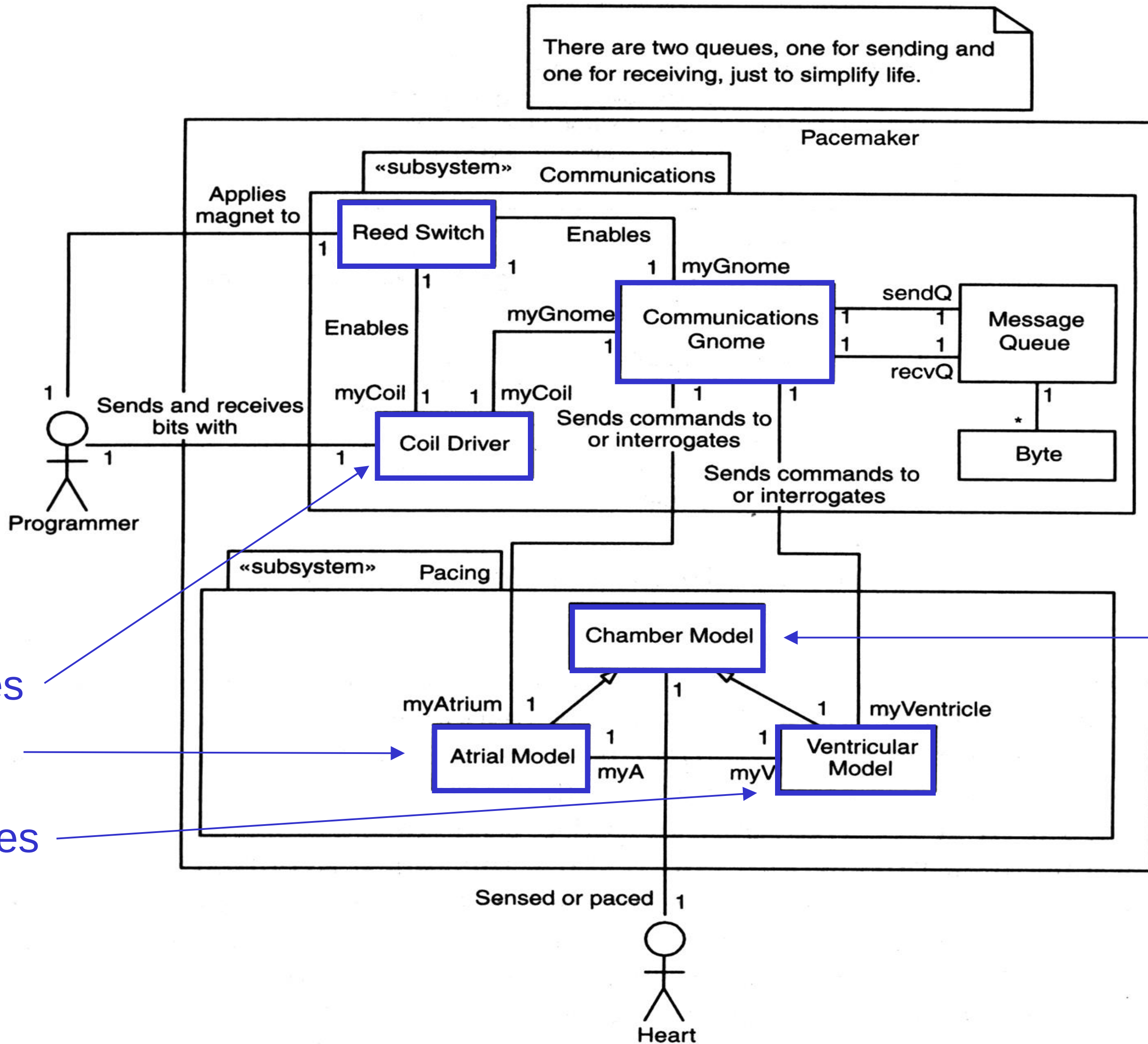


Discussion of State Pattern Implementation

- Disadvantages
  - Results in many small classes
  - Breaks encapsulation (of the state machine class)
- Advantages
  - Easy to implement most UML state diagram notations (guards, event parameters, entry, exit)
  - Easy to implement hierarchic state machines
  - State Pattern can be reverse engineered from code
  - All state handling logic in separate state classes
  - Easy to extend with new states
- **Recommendation:**
  - Very useful for nontrivial flat and hierarchical state machines

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Pacemaker – Class Diagram

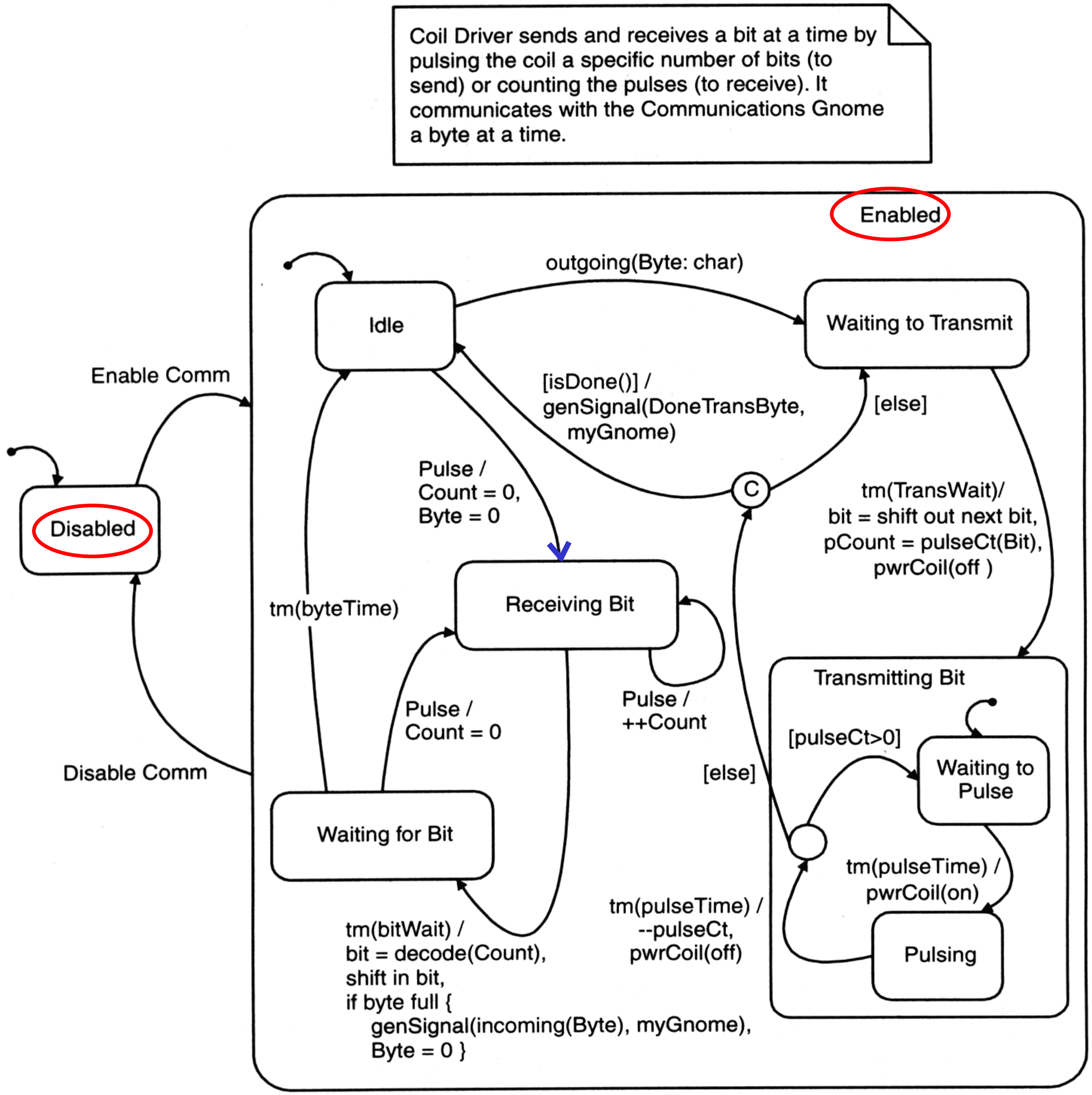


Classes with state machines

abstract class

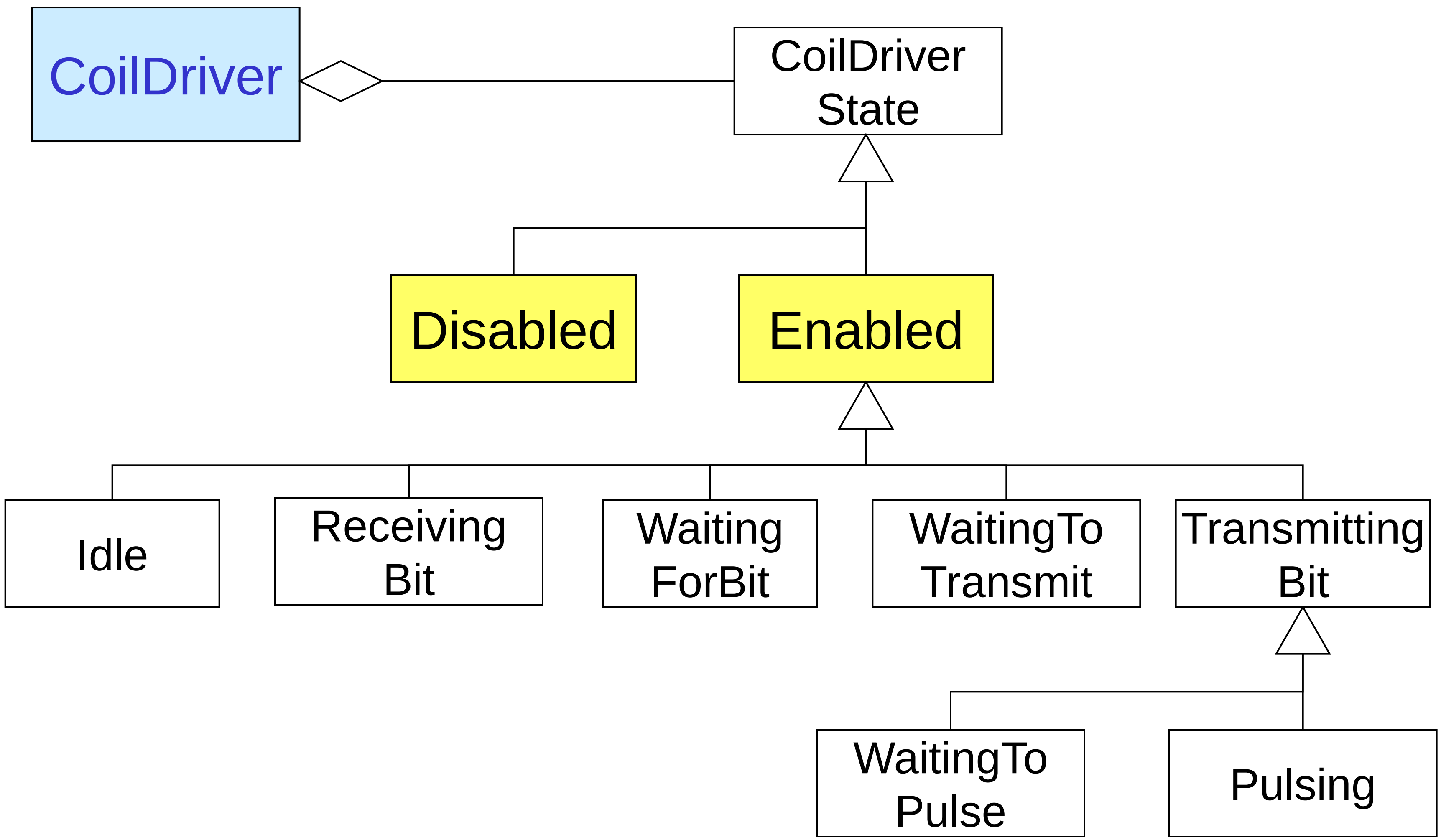
AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Pacemaker – State Diagram for Class Coil Driver



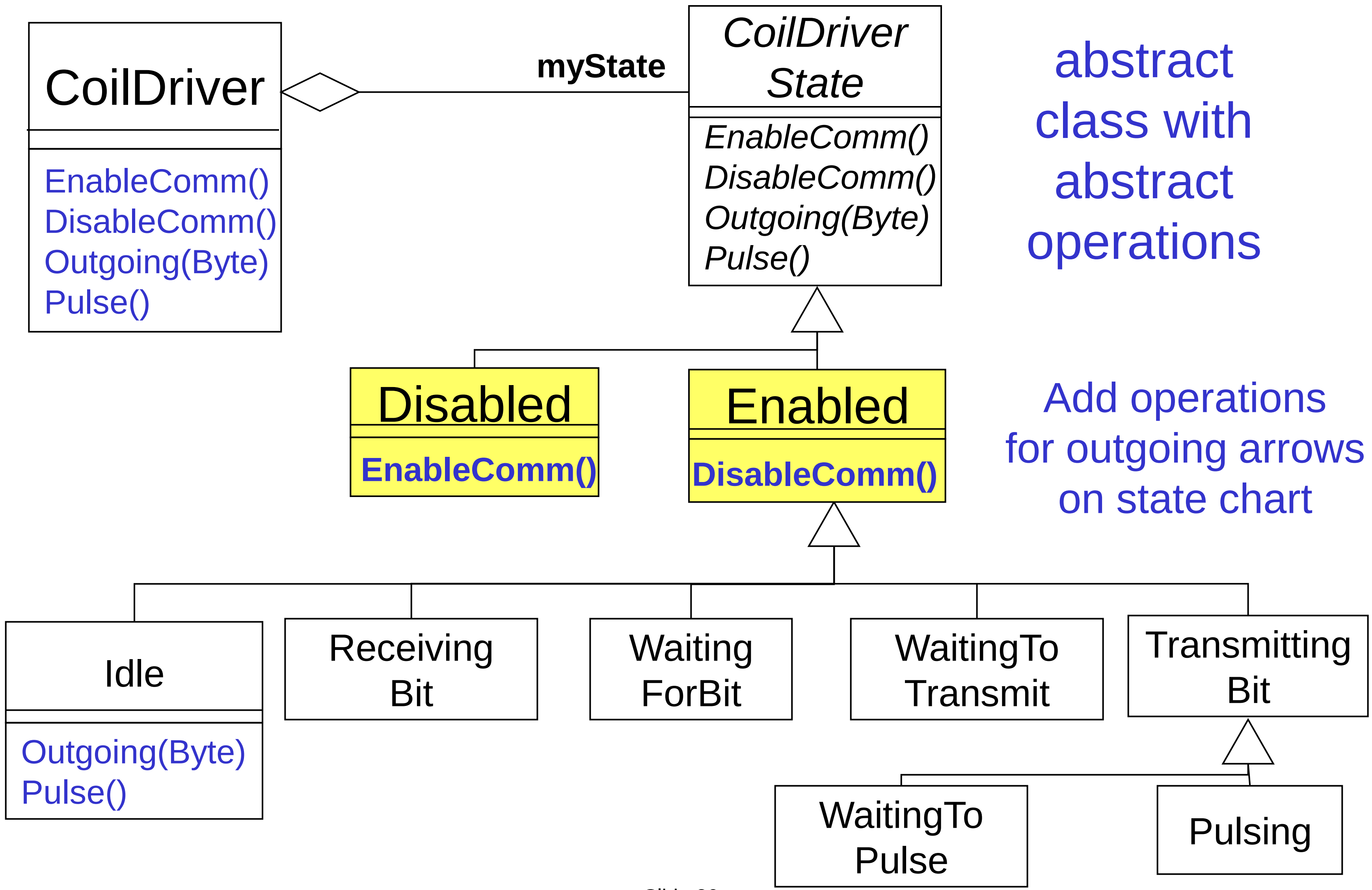
AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Step 1: Construct Class Hierarchy



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Step 2: Add Event Operations



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Summary

- State Pattern
  - extremely useful