State Diagram

C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

Contents

- Introduction to object states
- Interface vs. States
- Object with States
- Intuitive Implementation
- Explicit States
- State Diagram
- Systematic Implementation of the State Diagram
- Modification of the Design
- Design with the “State” Pattern

Introduction

- **State Diagram** is used to described the dynamic behavior of an object.

Introduction

- **State Diagram** is used to described the dynamic behavior of an object.
- What is the state of an object?

Introduction

- **State Diagram** is used to described the dynamic behavior of an object.
- What is the state of an object?
  - All objects have internal states.

Introduction

- **State Diagram** is used to described the dynamic behavior of an object.
- What is the state of an object?
  - All objects have internal states.
  - The response of an object to a message depends on its state

Introduction

- **State Diagram** is used to described the dynamic behavior of an object.
- What is the state of an object?
  - All objects have internal states.
  - The response of an object to a message depends on its state

  Ex.

Introduction

- **State Diagram** is used to described the dynamic behavior of an object.
- What is the state of an object?
  - All objects have internal states.
  - The response of an object to a message depends on its state

  Ex.
  - I can answer the phone, but whether I answer or not depends on I am **busy or not** when the phone rings.

Introduction

- **State Diagram** is used to described the dynamic behavior of an object.
- What is the state of an object?
  - All objects have internal states.
  - The response of an object to a message depends on its state

  Ex.
  - I can answer the phone, but whether I answer or not depends on I am **busy or not** when the phone rings.
  - A television set usually has a couple of control buttons, e.g. volume up/down, channel up/down, setup, power etc.

Introduction

- **State Diagram** is used to described the dynamic behavior of an object.
- What is the state of an object?
  - All objects have internal states.
  - The response of an object to a message depends on its state

  Ex.
  - I can answer the phone, but whether I answer or not depends on I am **busy or not** when the phone rings.
  - A television set usually has a couple of control buttons, e.g. volume up/down, channel up/down, setup, power etc. However, not every button is responding at any moment, e.g. volume up/down do not function when **power is off**.

Introduction

- **State Diagram** is used to described the dynamic behavior of an object.
- What is the state of an object?
  - All objects have internal states.
  - The response of an object to a message depends on its state

  Ex.
  - I can answer the phone, but whether I answer or not depends on I am **busy or not** when the phone rings.
  - A television set usually has a couple of control buttons, e.g. volume up/down, channel up/down, setup, power etc. However, not every button is responding at any moment, e.g. volume up/down do not function when **power is off**. Most of the buttons have a different set of functions when entering **setup mode**.

Introduction (cont'd)

- A turnstile has two states: **locked** or **unlocked** (coin deposited)

Introduction (cont'd)

- A turnstile has two states: **locked** or **unlocked** (coin deposited)
- When using an ifstream object for file input, a read operation of an integer might fail if the **current file pointer points to a non-numeric character** or if the file pointer points to the **end of file**.

Introduction (cont'd)

- A turnstile has two states: **locked** or **unlocked** (coin deposited)
- When using an ifstream object for file input, a read operation of an integer might fail if the **current file pointer points to a non-numeric character** or if the file pointer points to the **end of file**.
- We can push a value into a Stack object, only when the stack is **not full**. We can pop a value out only when the stack is **not empty**.

Introduction (cont'd)

- A turnstile has two states: **locked** or **unlocked** (coin deposited)
- When using an ifstream object for file input, a read operation of an integer might fail if the **current file pointer points to a non-numeric character** or if the file pointer points to the **end of file**.
- We can push a value into a Stack object, only when the stack is **not full**. We can pop a value out only when the stack is **not empty**.
- An editor has two input modes: **insert** or **overwrite**. In the insert mode, the input characters from the keyboard are inserted right before the cursor. In the overwrite move, the input characters overwrite the characters at the cursor.

Introduction (cont'd)

- A turnstile has two states: **locked** or **unlocked** (coin deposited)
- When using an ifstream object for file input, a read operation of an integer might fail if the **current file pointer points to a non-numeric character** or if the file pointer points to the **end of file**.
- We can push a value into a Stack object, only when the stack is **not full**. We can pop a value out only when the stack is **not empty**.
- An editor has two input modes: **insert** or **overwrite**. In the insert mode, the input characters from the keyboard are inserted right before the cursor. In the overwrite move, the input characters overwrite the characters at the cursor.
- An editor has two document modes: documents **modified** or **not modified**.

Introduction (cont'd)

- A turnstile has two states: **locked** or **unlocked** (coin deposited)
- When using an ifstream object for file input, a read operation of an integer might fail if the **current file pointer points to a non-numeric character** or if the file pointer points to the **end of file**.
- We can push a value into a Stack object, only when the stack is **not full**. We can pop a value out only when the stack is **not empty**.
- An editor has two input modes: **insert** or **overwrite**. In the insert mode, the input characters from the keyboard are inserted right before the cursor. In the overwrite move, the input characters overwrite the characters at the cursor.
- An editor has two document modes: documents **modified** or **not modified**.
- An editor has two UI modes: document **specified** or **not specified**.

Introduction (cont'd)

- A turnstile has two states: **locked** or **unlocked** (coin deposited)
- When using an ifstream object for file input, a read operation of an integer might fail if the **current file pointer points to a non-numeric character** or if the file pointer points to the **end of file**.
- We can push a value into a Stack object, only when the stack is **not full**. We can pop a value out only when the stack is **not empty**.
- An editor has two input modes: **insert** or **overwrite**. In the insert mode, the input characters from the keyboard are inserted right before the cursor. In the overwrite move, the input characters overwrite the characters at the cursor.
- An editor has two document modes: documents **modified** or **not modified**.
- An editor has two UI modes: document **specified** or **not specified**.
- …

Introduction (cont'd)

Note: 1. A very simple object might have a fixed state such that its behavior is all the way consistent.

Introduction (cont'd)

Note: 1. A very simple object might have a fixed state such that its behavior is all the way consistent.

2. The timing of messages to an object with various internal states is important and determines how an object responds.

Introduction (cont'd)

Note: 1. A very simple object might have a fixed state such that its behavior is all the way consistent.

2. The timing of messages to an object with various internal states is important and determines how an object responds.

3. Usually the states of an object cannot be observed directly from outside. The messages an object received up to now affect its current state and therefore its future behaviors.

Object Interface vs. Object State

- The object interface depends also on its current state.

Object Interface vs. Object State

- The object interface depends also on its current state.
- Object interface (the usage of an object)
  - Public operations (member functions)
  - The sequence (order) of the operations being executed

Object Interface vs. Object State

- The object interface depends also on its current state.
- Object interface (the usage of an object)
  - Public operations (member functions)
  - The sequence (order) of the operations being executed
- “Some operations are required to follow other operations” indicates the existence of object’s internal state.

Object Interface vs. Object State

- The object interface depends also on its current state.
- Object interface (the usage of an object)
  - Public operations (member functions)
  - The sequence (order) of the operations being executed
- “Some operations are required to follow other operations” indicates the existence of object’s internal state.
- If a client program does not follow the pre-specified order to use the interface, the object could possibly refuse to respond and enter a special error state.

Object Interface vs. Object State

- The object interface depends also on its current state.
- Object interface (the usage of an object)
  - Public operations (member functions)
  - The sequence (order) of the operations being executed
- "Some operations are required to follow other operations" indicates the existence of object's internal state.
- If a client program does not follow the pre-specified order to use the interface, the object could possibly refuse to respond and enter a special error state.

  Ex.

  A network communication end point

Object Interface vs. Object State

- The object interface depends also on its current state.
- Object interface (the usage of an object)
  - Public operations (member functions)
  - The sequence (order) of the operations being executed
- "Some operations are required to follow other operations" indicates the existence of object's internal state.
- If a client program does not follow the pre-specified order to use the interface, the object could possibly refuse to respond and enter a special error state.

  Ex.

  A network communication end point

Object with States

```
class NetCommStream {
public:
    void open();
    void connect();
    void read();
    void write();
    void disconnect();
    void close();
private:
    ...
};
```

Object with States

```
class NetCommStream {
public:
    void open();
    void connect();
    void read();
    void write();
    void disconnect();
    void close();
private:
    ...
};
```

Correct usage:

```
NetCommStream obj;
obj.open();
obj.connect();
obj.read();
obj.disconnect();
obj.close();
```

Object with States

```
class NetCommStream {
public:
    void open();
    void connect();
    void read();
    void write();
    void disconnect();
    void close();
private:
    ...
};
```

Correct usage:

```
NetCommStream obj;
obj.open();
obj.connect();
obj.read();
obj.disconnect();
obj.close();
```

Incorrect usage:

```
NetCommStream obj;
obj.open();
obj.read();
```

Object with States

```
class NetCommStream {
public:
    void open();
    void connect();
    void read();
    void write();
    void disconnect();
    void close();
private:
    ...
};
```

- **Usage description:**

Correct usage:

```
NetCommStream obj;
obj.open();
obj.connect();
obj.read();
obj.disconnect();
obj.close();
```

Incorrect usage:

```
NetCommStream obj;
obj.open();
obj.read();
```

Object with States

```
class NetCommStream {
public:
    void open();
    void connect();
    void read();
    void write();
    void disconnect();
    void close();
private:
    ...
};
```

- **Usage description:**

  A stream can only be opened (for setting up its own communication interface) when it is not currently opened.

Correct usage:

```
NetCommStream obj;
obj.open();
obj.connect();
obj.read();
obj.disconnect();
obj.close();
```

Incorrect usage:

```
NetCommStream obj;
obj.open();
obj.read();
```

Object with States

```
class NetCommStream {
public:
    void open();
    void connect();
    void read();
    void write();
    void disconnect();
    void close();
private:
    ...
};
```

- **Usage description:**

  A stream can only be opened (for setting up its own communication interface) when it is not currently opened. A stream can only be connected (for building up the connection with a remote machine) when it is opened but not connected

Correct usage:

```
NetCommStream obj;
obj.open();
obj.connect();
obj.read();
obj.disconnect();
obj.close();
```

Incorrect usage:

```
NetCommStream obj;
obj.open();
obj.read();
```

Object with States

```
class NetCommStream {
public:
    void open();
    void connect();
    void read();
    void write();
    void disconnect();
    void close();
private:
    ...
};
```

Correct usage:

```
NetCommStream obj;
obj.open();
obj.connect();
obj.read();
obj.disconnect();
obj.close();
```

- **Usage description:**

  A stream can only be opened (for setting up its own communication interface) when it is not currently opened. A stream can only be connected (for building up the connection with a remote machine) when it is opened but not connected. A stream object can be read / write / disconnected only when it is connected properly.

Incorrect usage:

```
NetCommStream obj;
obj.open();
obj.read();
```

Intuitive Implementation

- Using *bool* variables to keep various kinds of states

Intuitive Implementation

- Using *bool* variables to keep various kinds of states

```
void open() {
    if (!m_fOpen) {
        m_fOpen = true;
        do_open();
    }
}
```

Intuitive Implementation

- Using *bool* variables to keep various kinds of states

```
void open() {
    if (!m_fOpen) {
        m_fOpen = true;
        do_open();
    }
}

void connect() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fConnected = true;
        do_connect();
    }
}
```

Intuitive Implementation

- Using *bool* variables to keep various kinds of states

```
void open() {
    if (!m_fOpen) {
        m_fOpen = true;
        do_open();
    }
}

void connect() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fConnected = true;
        do_connect();
    }
}

void read() {
    if (m_fConnected)
        do_read();
}
```

Intuitive Implementation

- Using *bool* variables to keep various kinds of states

```
void open() {
    if (!m_fOpen) {
        m_fOpen = true;
        do_open();
    }
}

void connect() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fConnected = true;
        do_connect();
    }
}

void read() {
    if (m_fConnected)
        do_read();
}
```

```
void disconnect() {
    if (m_fConnected) {
        m_fConnected = false;
        do_disconnect();
    }
}
```

Intuitive Implementation

- Using *bool* variables to keep various kinds of states

```
void open() {
    if (!m_fOpen) {
        m_fOpen = true;
        do_open();
    }
}

void connect() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fConnected = true;
        do_connect();
    }
}

void read() {
    if (m_fConnected)
        do_read();
}
```

```
void disconnect() {
    if (m_fConnected) {
        m_fConnected = false;
        do_disconnect();
    }
}

void close() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fOpen = false;
        do_close();
    }
}
```

Intuitive Implementation

- Using *bool* variables to keep various kinds of states

```
void open() {
    if (!m_fOpen) {
        m_fOpen = true;
        do_open();
    }
}

void connect() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fConnected = true;
        do_connect();
    }
}

void read() {
    if (m_fConnected)
        do_read();
}
```

```
void disconnect() {
    if (m_fConnected) {
        m_fConnected = false;
        do_disconnect();
    }
}

void close() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fOpen = false;
        do_close();
    }
}

void write() {
    if (m_fConnected)
        do_write();
}
```

Intuitive Implementation

- Using *bool* variables to keep various kinds of states

```
void open() {
    if (!m_fOpen) {
        m_fOpen = true;
        do_open();
    }
}

void connect() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fConnected = true;
        do_connect();
    }
}

void read() {
    if (m_fConnected)
        do_read();
}
```

```
void disconnect() {
    if (m_fConnected) {
        m_fConnected = false;
        do_disconnect();
    }
}

void close() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fOpen = false;
        do_close();
    }
}

void write() {
    if (m_fConnected)
        do_write();
}
```

Two flags are used in the above implementation. **4 different states**?

Intuitive Implementation

- Using *bool* variables to keep various kinds of states

```
void open() {
    if (!m_fOpen) {
        m_fOpen = true;
        do_open();
    }
}

void connect() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fConnected = true;
        do_connect();
    }
}

void read() {
    if (m_fConnected)
        do_read();
}
```

```
void disconnect() {
    if (m_fConnected) {
        m_fConnected = false;
        do_disconnect();
    }
}

void close() {
    if ((m_fOpen)&&(!m_fConnected)) {
        m_fOpen = false;
        do_close();
    }
}

void write() {
    if (m_fConnected)
        do_write();
}
```

implicit and vague

Two flags are used in the above implementation. **4 different states**?

Explicit State

- Two *bool* variables m_fOpen and m_fConnected define 4 legal states

Explicit State

- Two *bool* variables m_fOpen and m_fConnected define 4 legal states; but only 3 of them are meaningful to this application

Explicit State

- Two *bool* variables m_fOpen and m_fConnected define 4 legal states; but only 3 of them are meaningful to this application

m_fOpen	m_fConnected	State
false	false	**Closed**
false	true	~~X~~
true	false	**Opened**
true	true	**Connected**

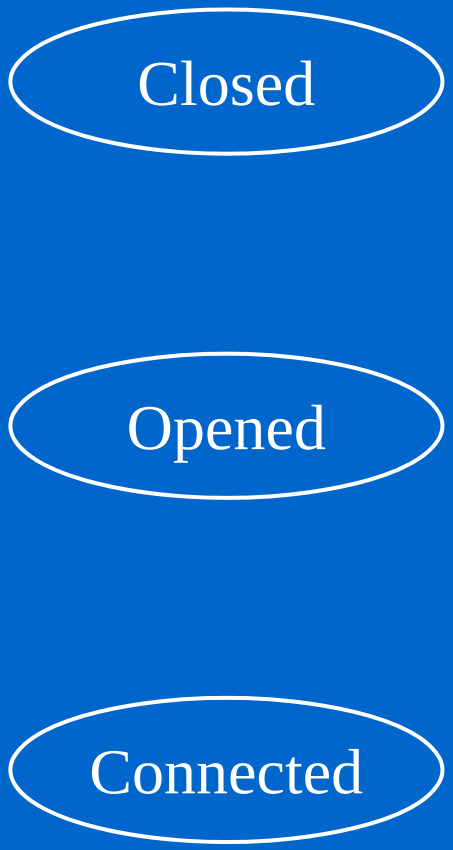
Explicit State

- Two *bool* variables m_fOpen and m_fConnected define 4 legal states; but only 3 of them are meaningful to this application

m_fOpen	m_fConnected	State
false	false	**Closed**
false	true	~~X~~
true	false	**Opened**
true	true	**Connected**

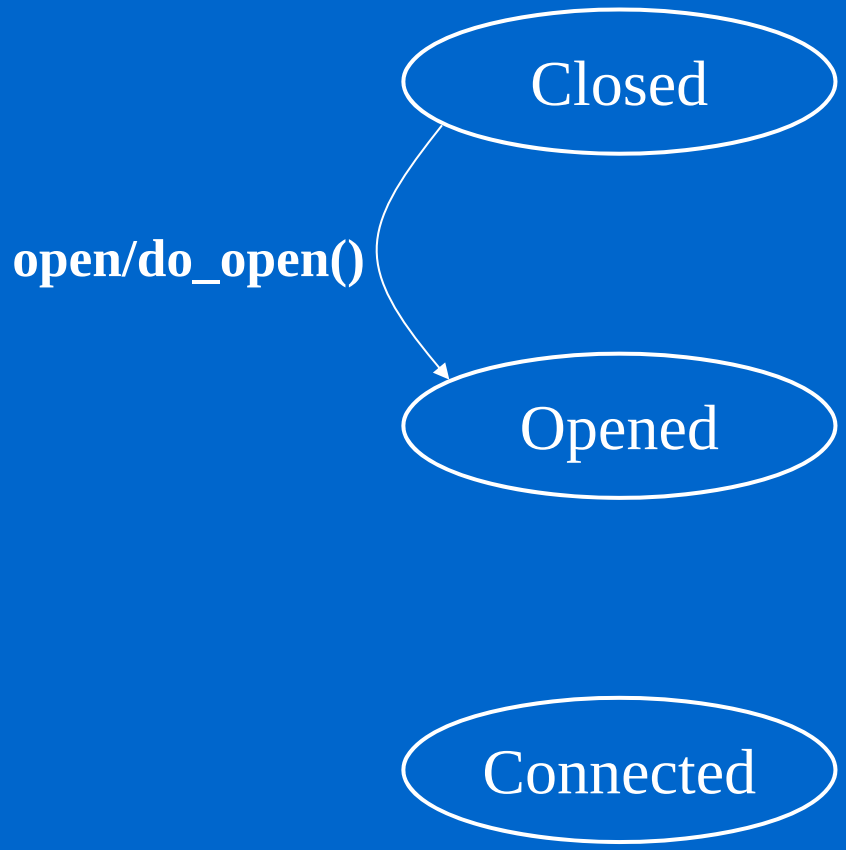
- There are six possible events (messages) to this object
  - **open**
  - **connect**
  - **read**
  - **write**
  - **disconnect**
  - **close**

State Diagram



Consider ALL possible events at the outgoing branches of each state.

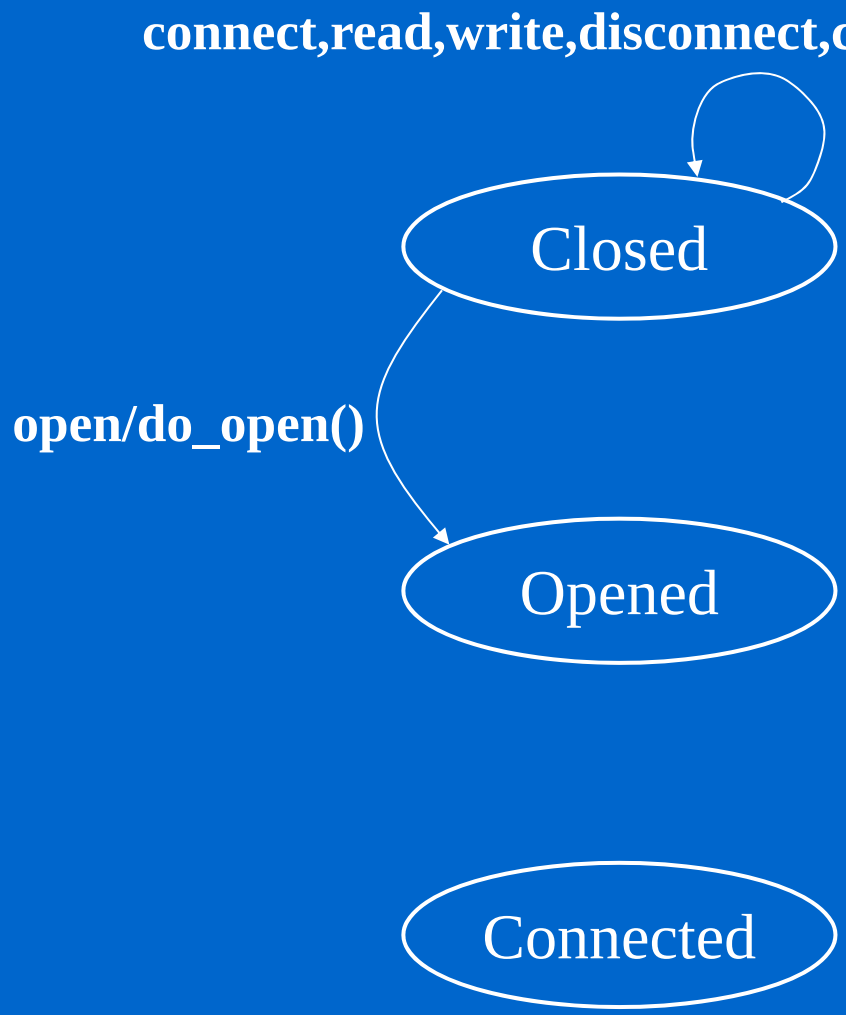
State Diagram



Consider ALL possible events at the outgoing branches of each state.

Finite State Machine

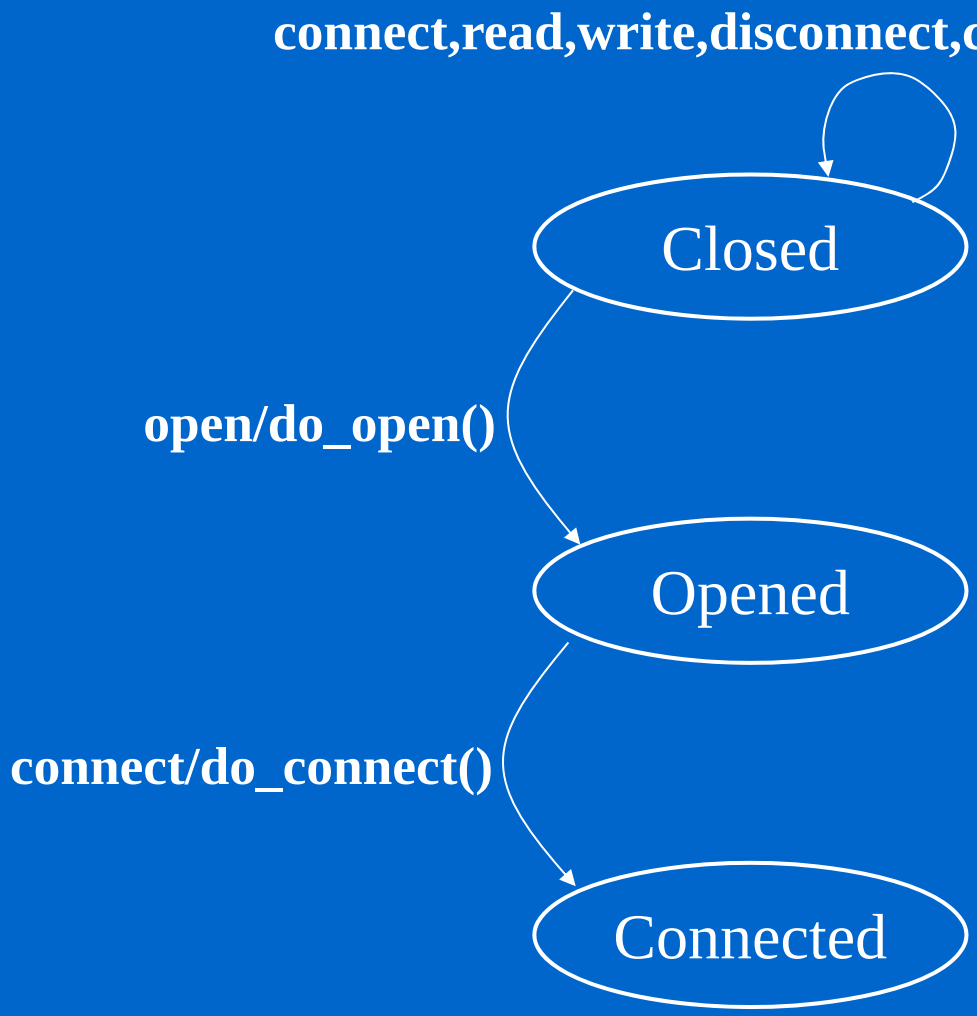
State Diagram



Consider ALL possible events at the outgoing branches of each state.

Finite State Machine

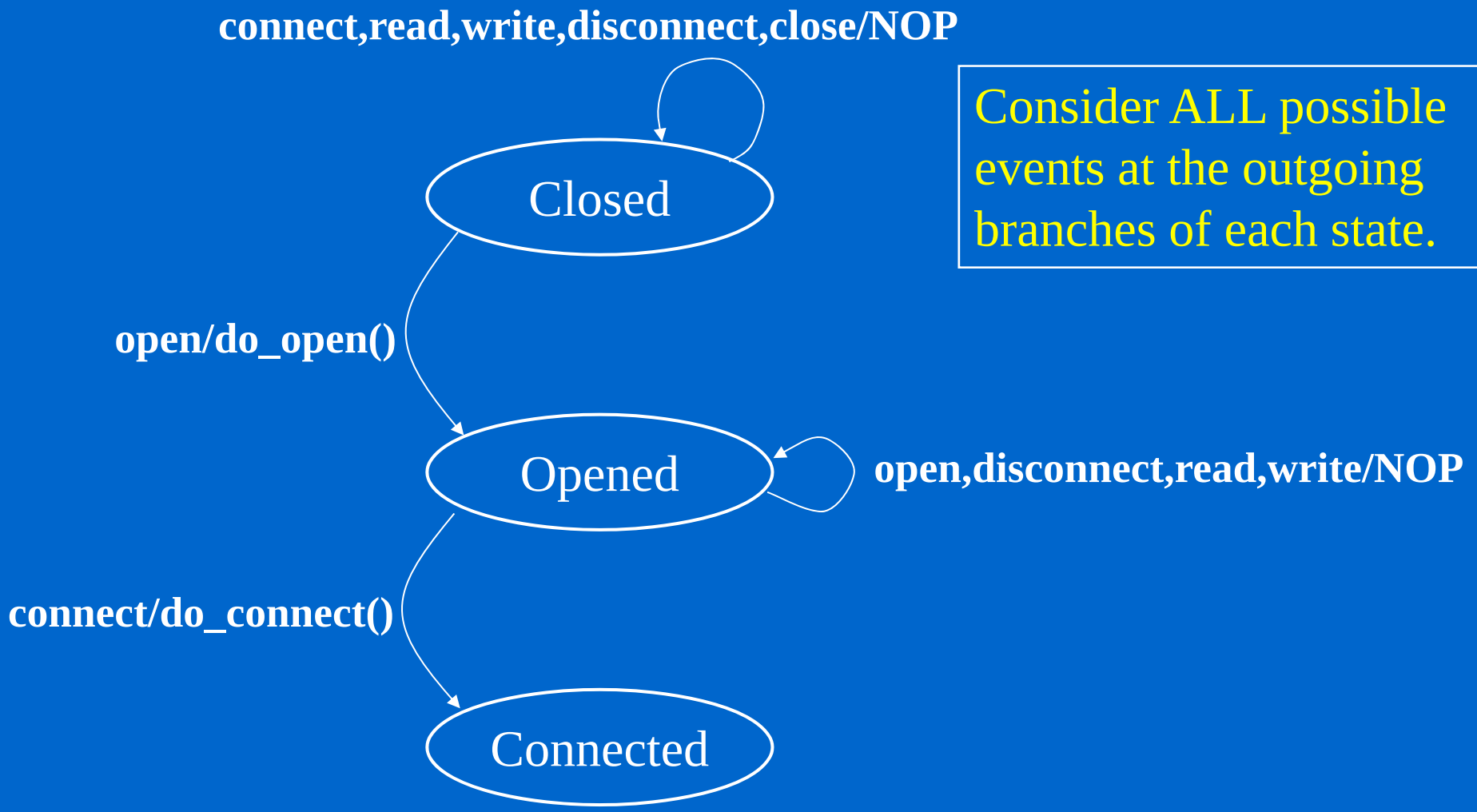
State Diagram



Consider ALL possible events at the outgoing branches of each state.

Finite State Machine

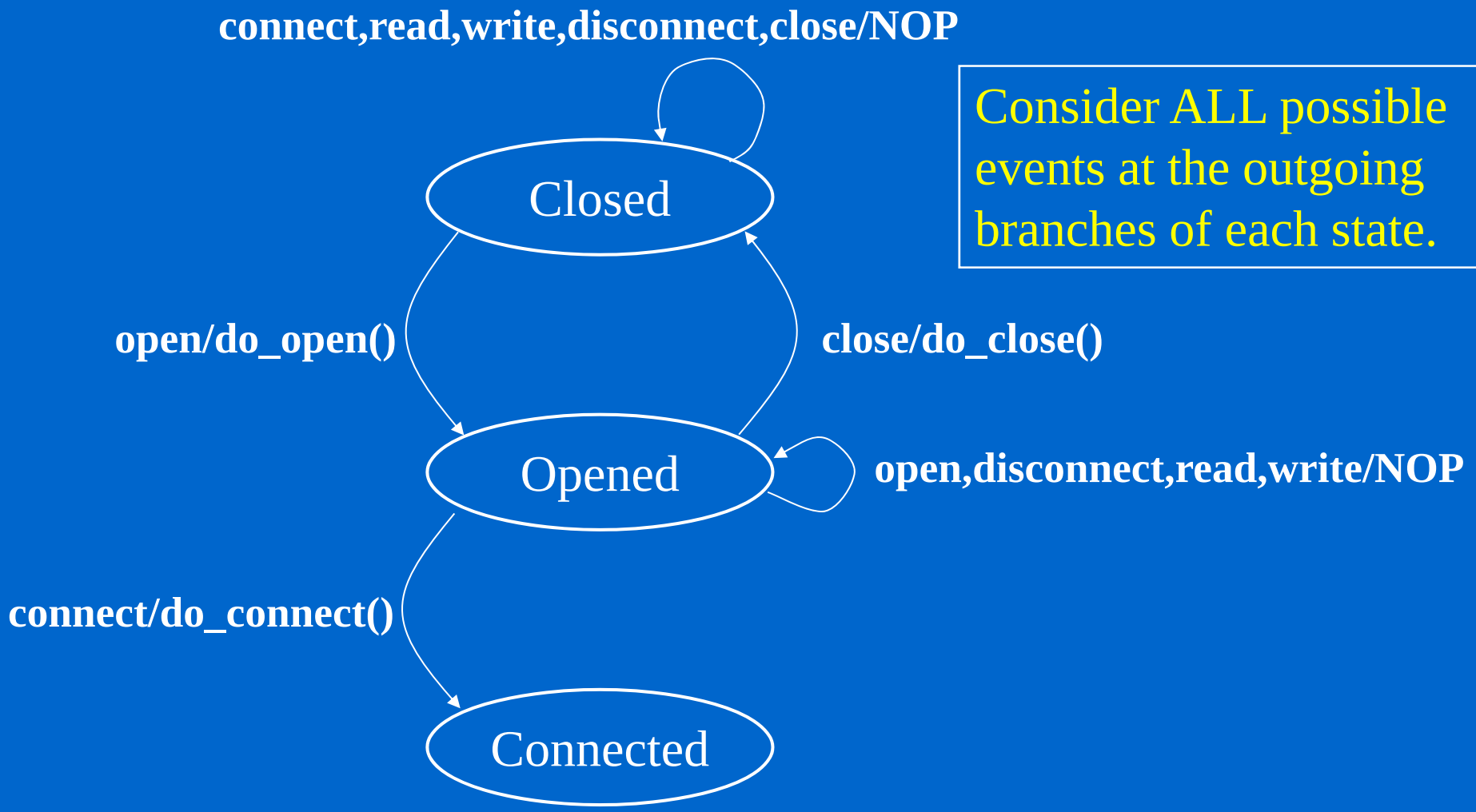
State Diagram



Consider ALL possible events at the outgoing branches of each state.

Finite State Machine

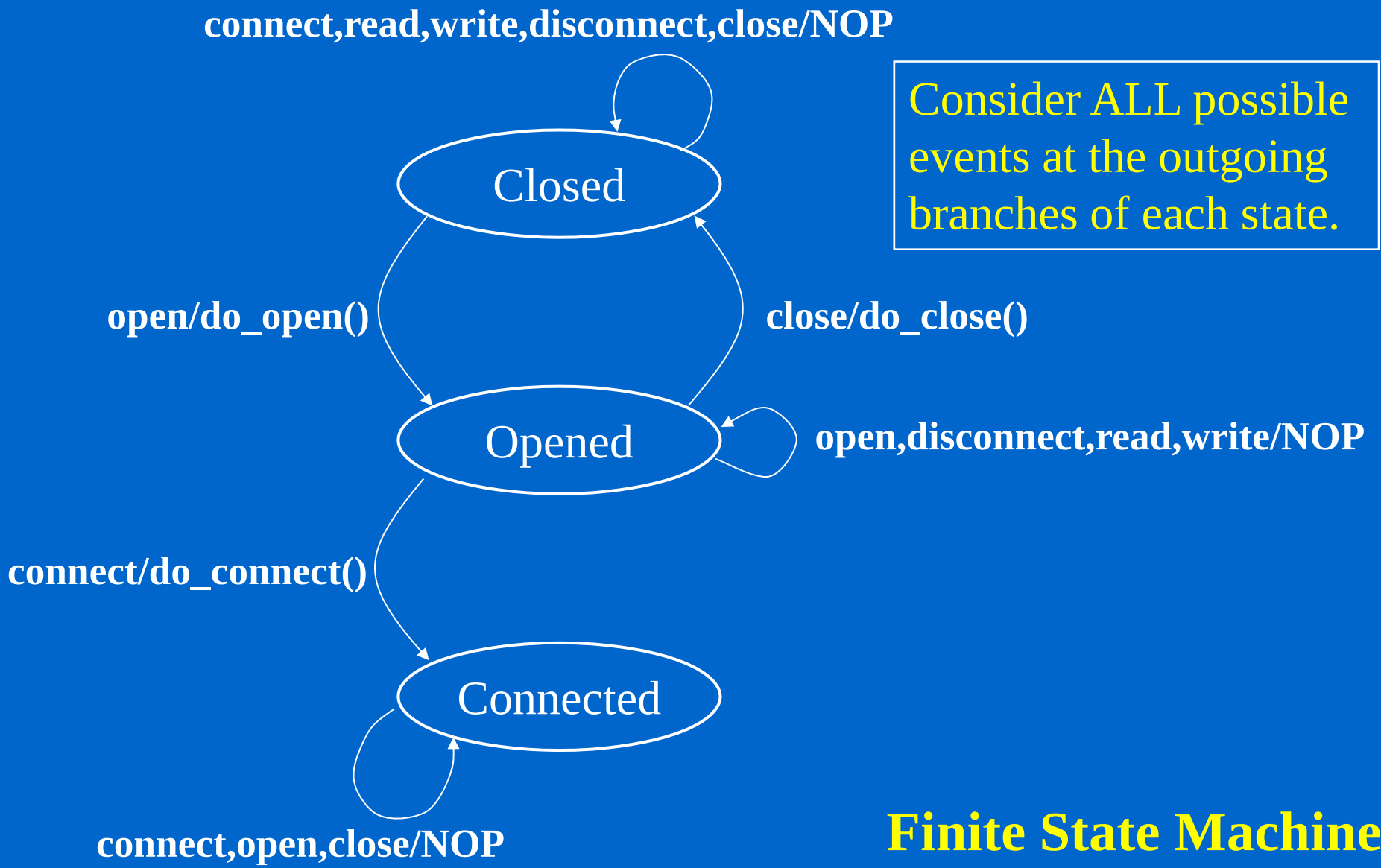
State Diagram



Consider ALL possible events at the outgoing branches of each state.

Finite State Machine

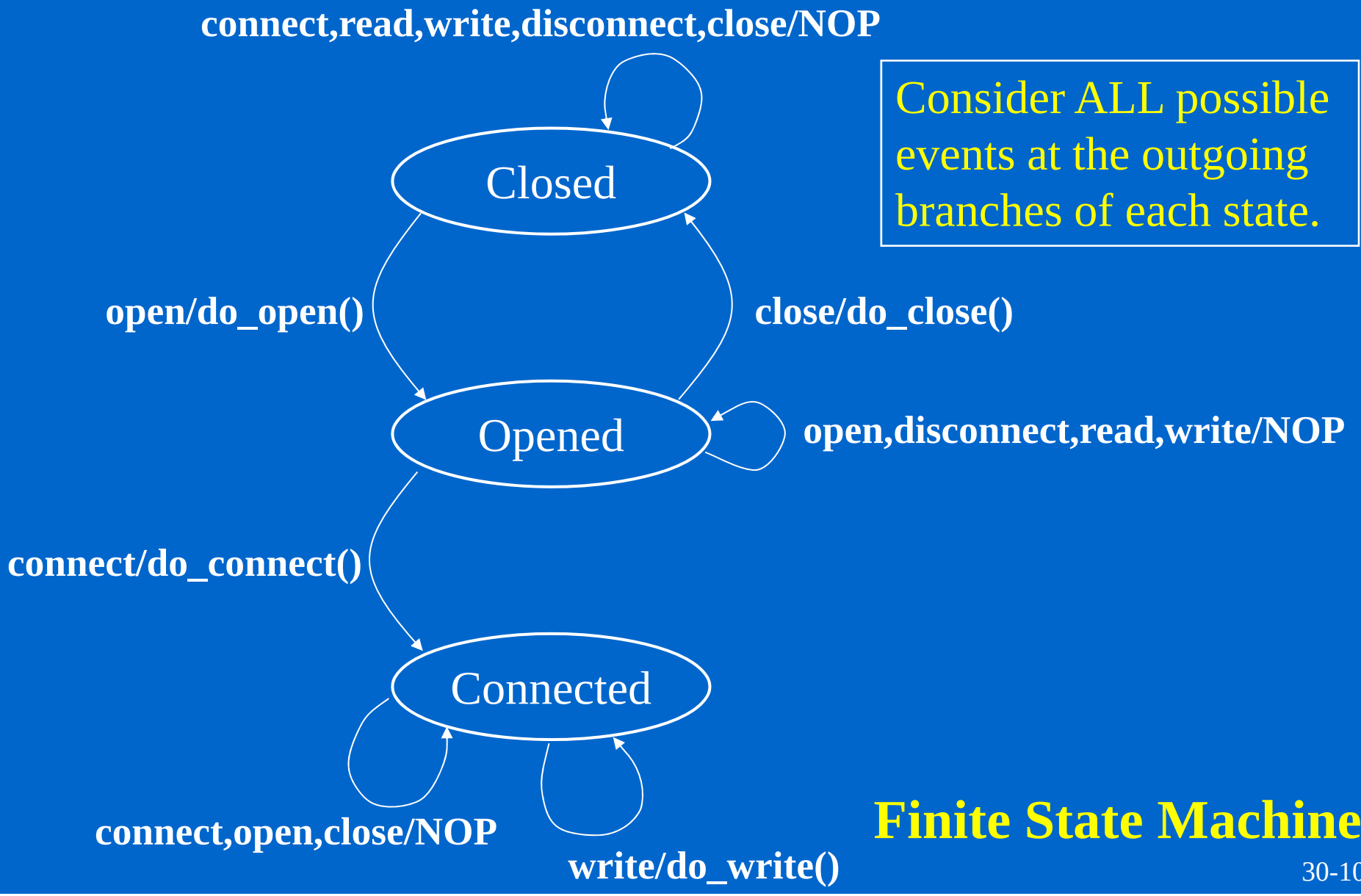
State Diagram



Consider ALL possible events at the outgoing branches of each state.

Finite State Machine

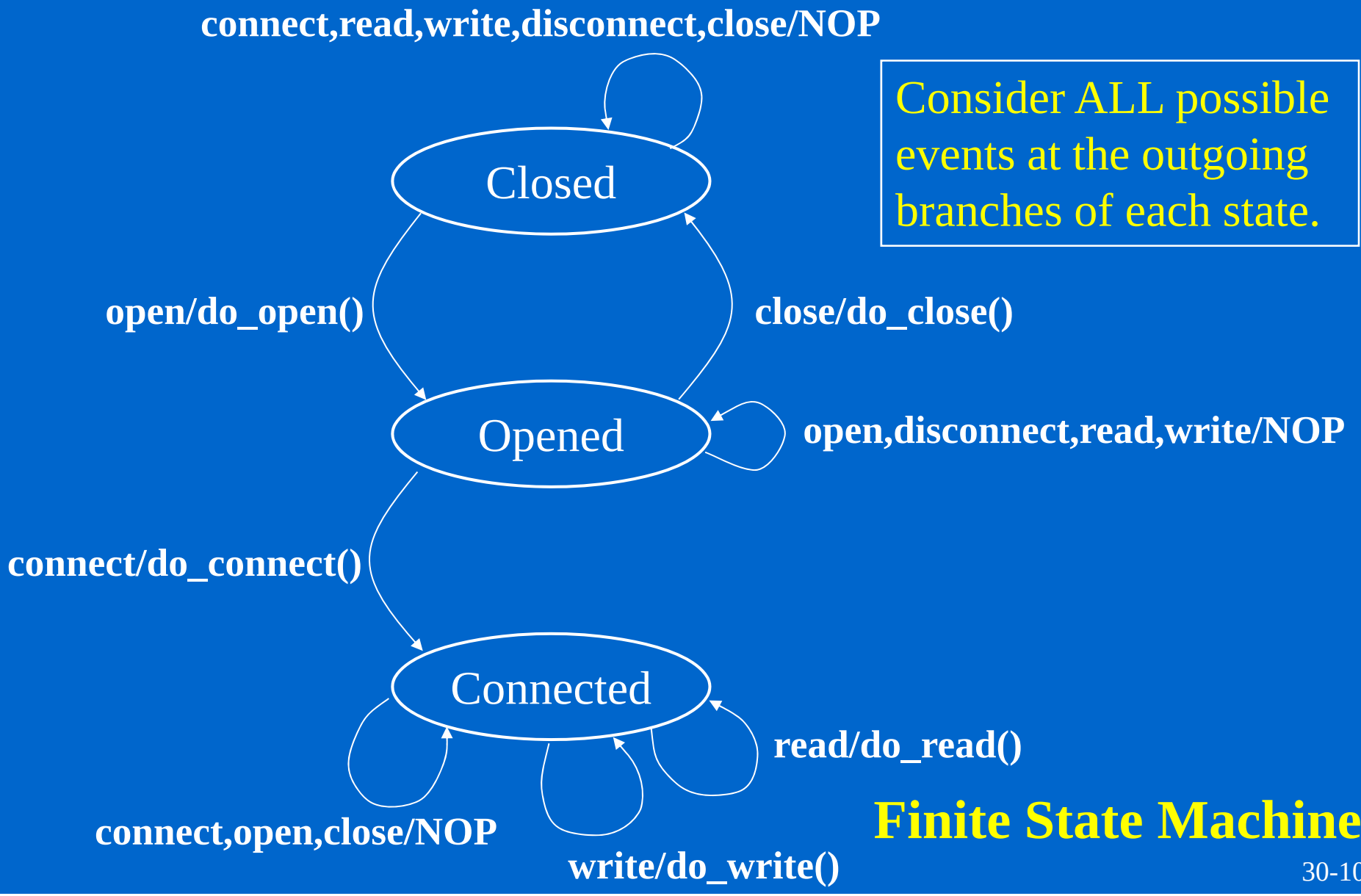
State Diagram



Consider ALL possible events at the outgoing branches of each state.

Finite State Machine

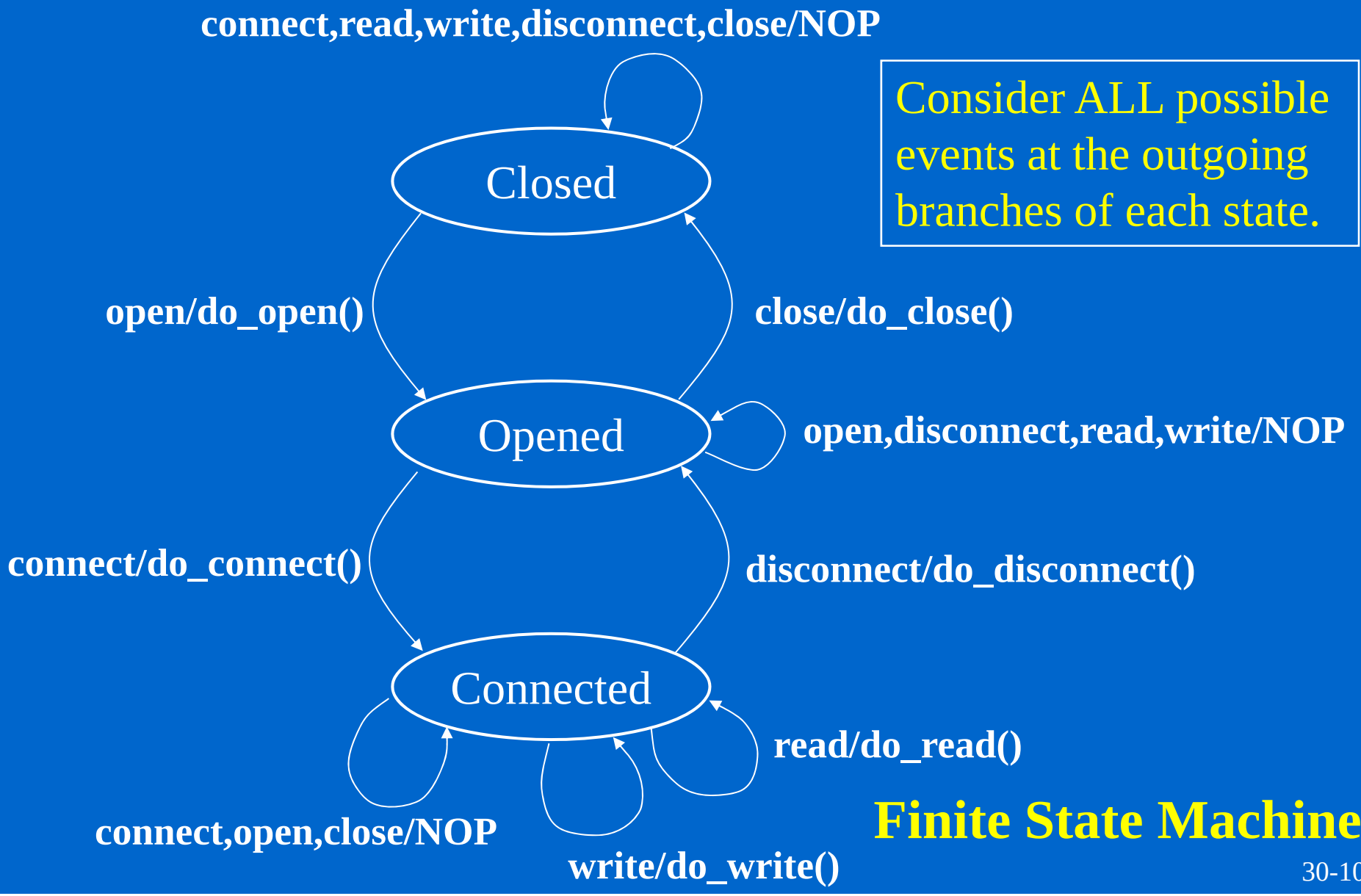
State Diagram



Consider ALL possible events at the outgoing branches of each state.

Finite State Machine

State Diagram



Consider ALL possible events at the outgoing branches of each state.

Finite State Machine

State Diagram (cont'd)

- Advantages:

State Diagram (cont'd)

- Advantages:
  - Show only valid states in the diagram

State Diagram (cont'd)

- Advantages:
  - Show only valid states in the diagram
  - Label each state with meaningful words

State Diagram (cont'd)

- Advantages:
  - Show only valid states in the diagram
  - Label each state with meaningful words
  - Allow programmer to consider the full set of events at each state

State Diagram (cont'd)

- Advantages:
  - Show only valid states in the diagram
  - Label each state with meaningful words
  - Allow programmer to consider the full set of events at each state
  - Simplify the considerations of server program logics.

State Diagram (cont'd)

- Advantages:
  - Show only valid states in the diagram
  - Label each state with meaningful words
  - Allow programmer to consider the full set of events at each state
  - Simplify the considerations of server program logics. A state diagram for the server object shows a lot more design information than a control flow diagram for the server

State Diagram (cont'd)

- Advantages:
  - Show only valid states in the diagram
  - Label each state with meaningful words
  - Allow programmer to consider the full set of events at each state
  - Simplify the considerations of server program logics. A state diagram for the server object shows a lot more design information than a control flow diagram for the server. (The server control flow diagram is incomplete and fragmented without the client control flow diagram.)

State Diagram (cont'd)

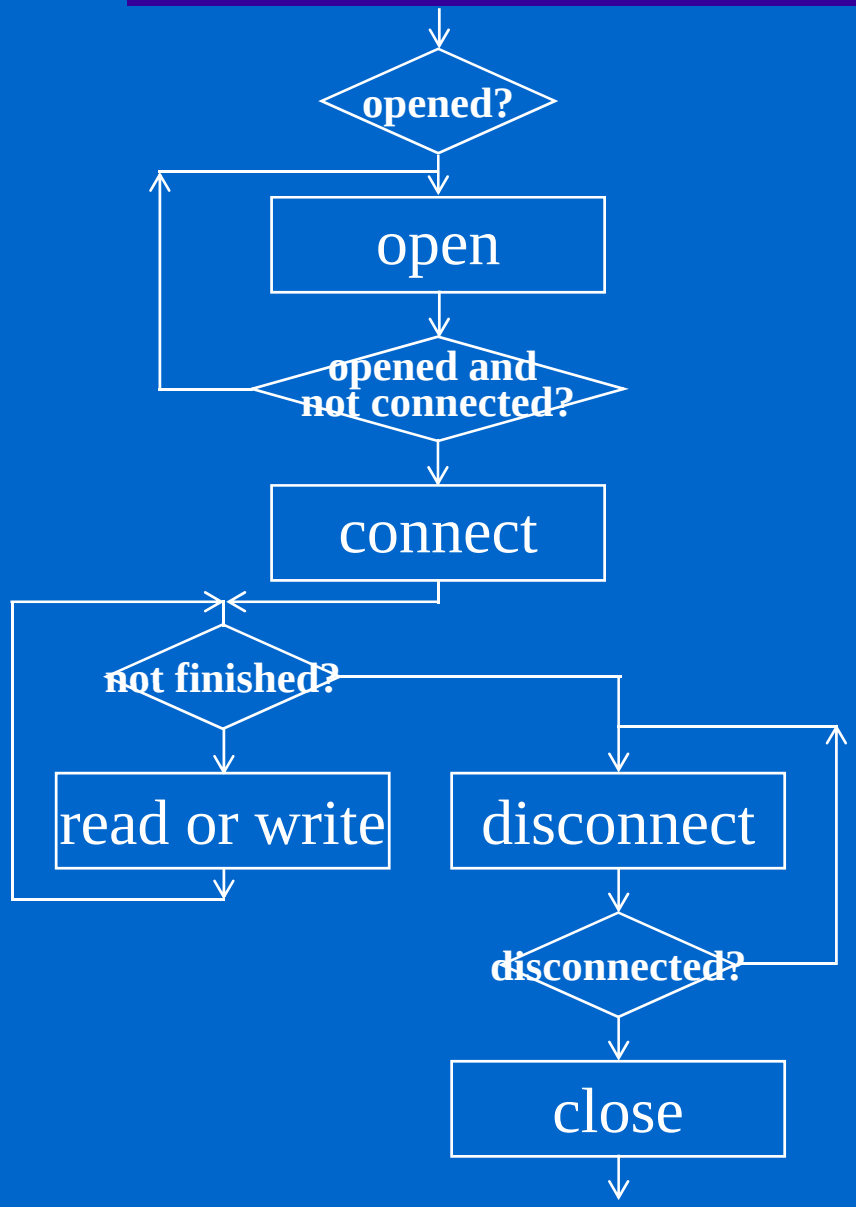
- Advantages:
  - Show only valid states in the diagram
  - Label each state with meaningful words
  - Allow programmer to consider the full set of events at each state
  - Simplify the considerations of server program logics. A state diagram for the server object shows a lot more design information than a control flow diagram for the server. (The server control flow diagram is incomplete and fragmented without the client control flow diagram.)
- A control flow diagram of the client simply does not show all possible ways of usages.

State Diagram (cont'd)

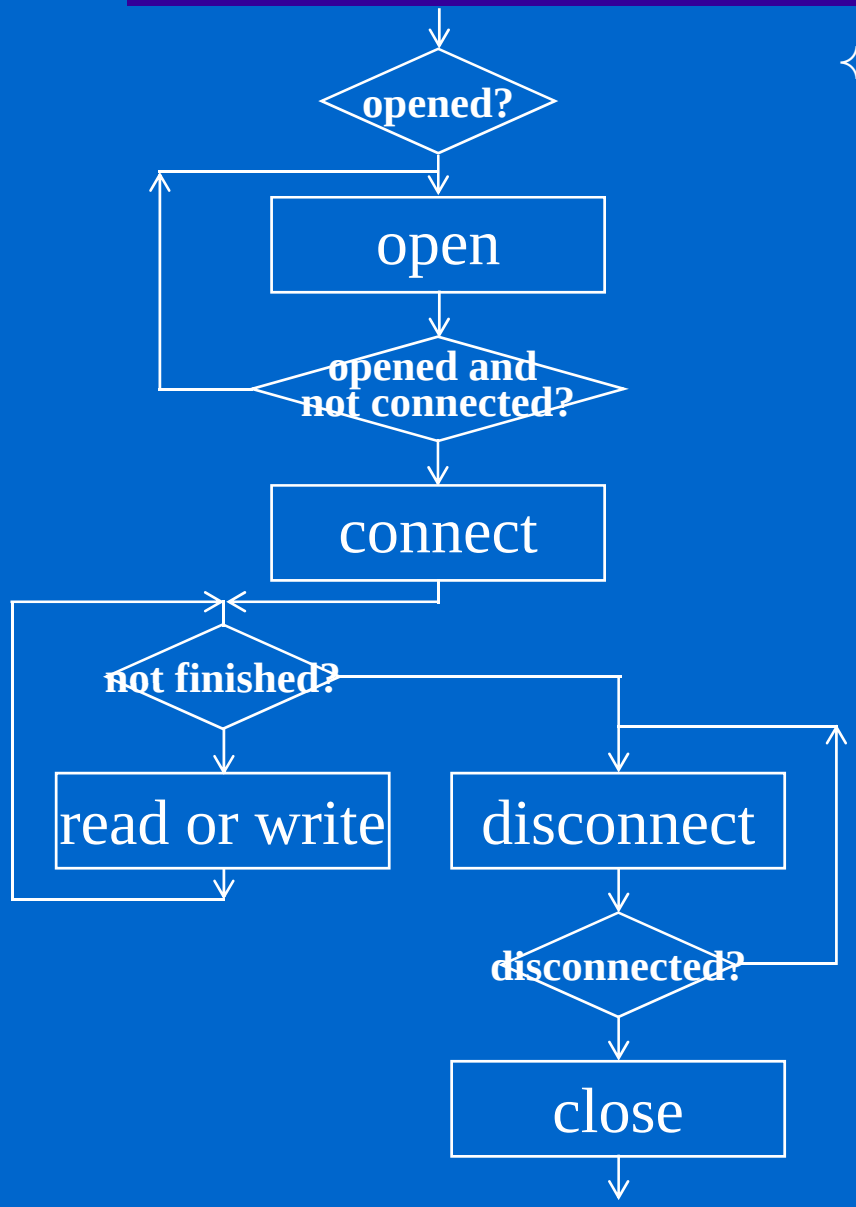
- Advantages:
  - Show only valid states in the diagram
  - Label each state with meaningful words
  - Allow programmer to consider the full set of events at each state
  - Simplify the considerations of server program logics. A state diagram for the server object shows a lot more design information than a control flow diagram for the server. (The server control flow diagram is incomplete and fragmented without the client control flow diagram.)
- A control flow diagram of the client simply does not show all possible ways of usages.

See the following example…

Control Flow Diagram of A Client

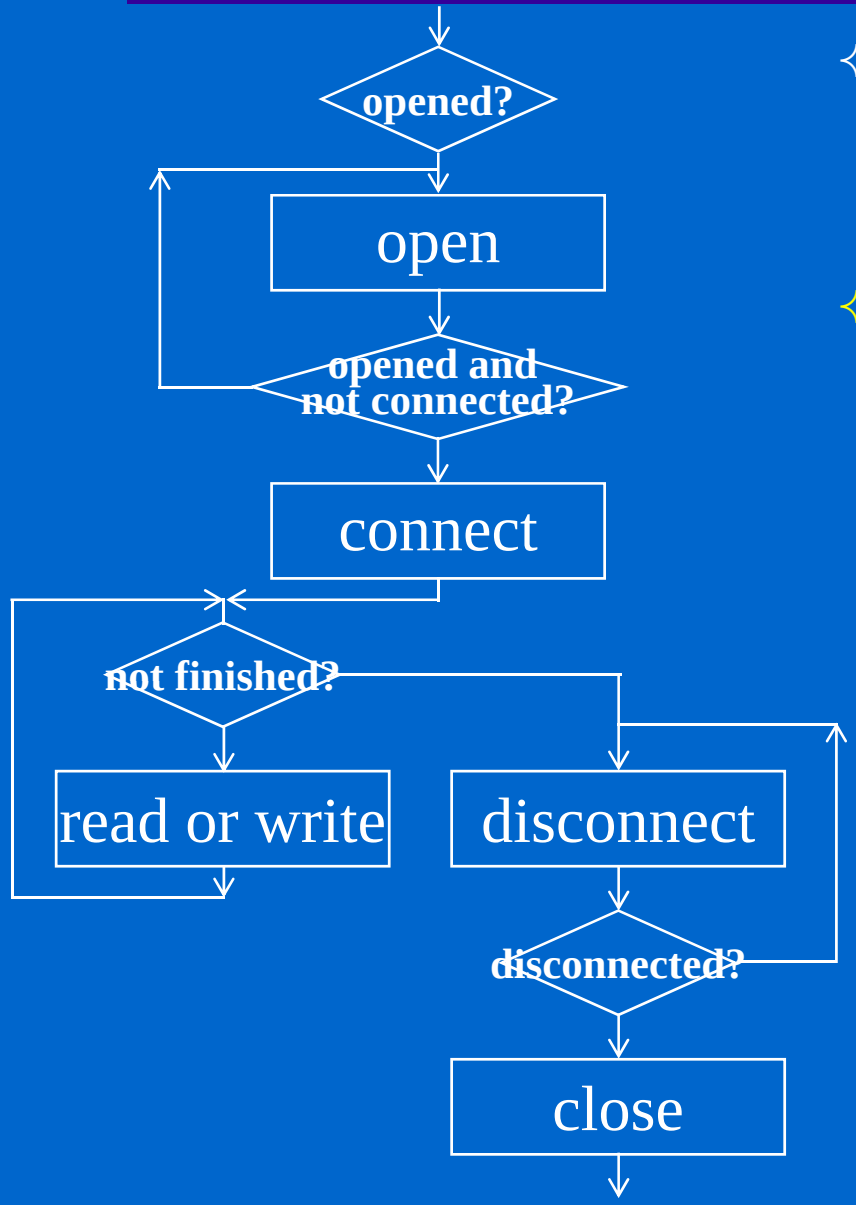


Control Flow Diagram of A Client



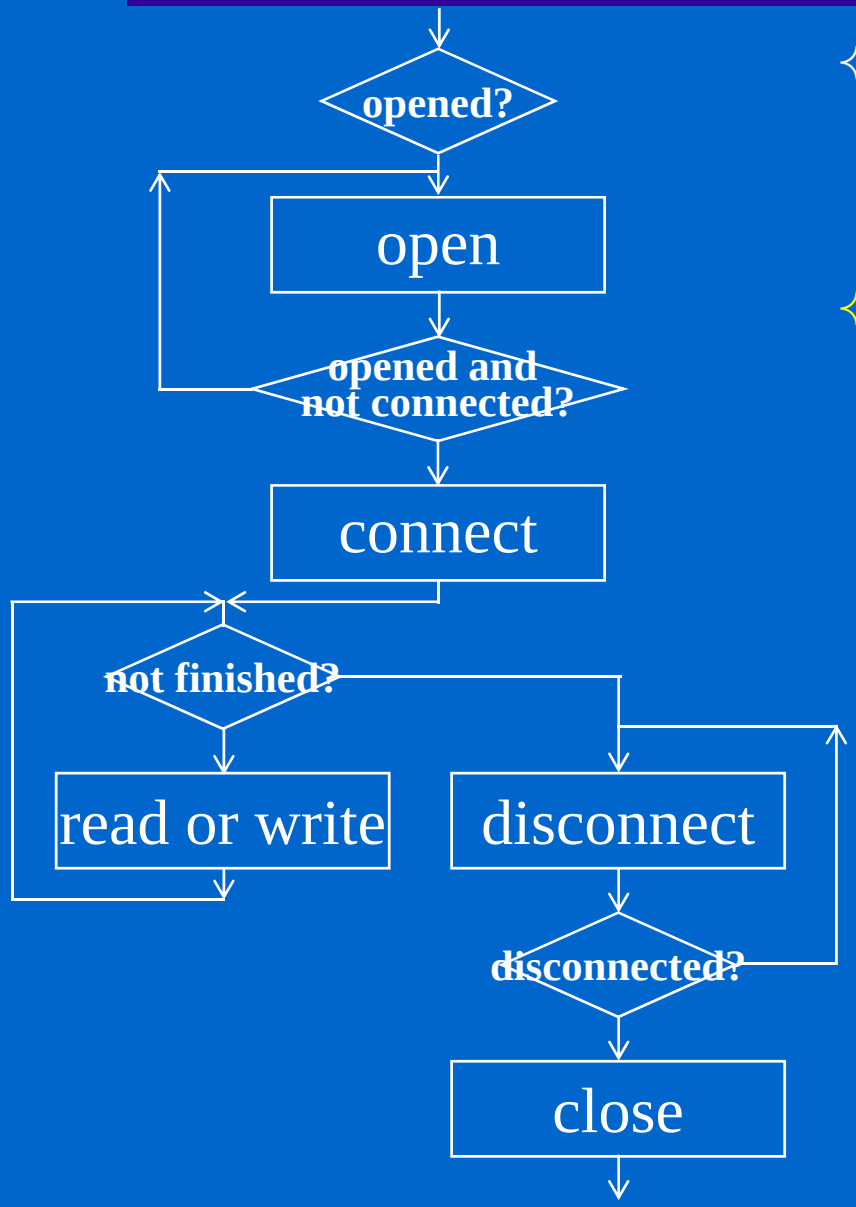
- This is the control flow for the TYPICAL / CORRECT usage of this NetCommStream object.

Control Flow Diagram of A Client



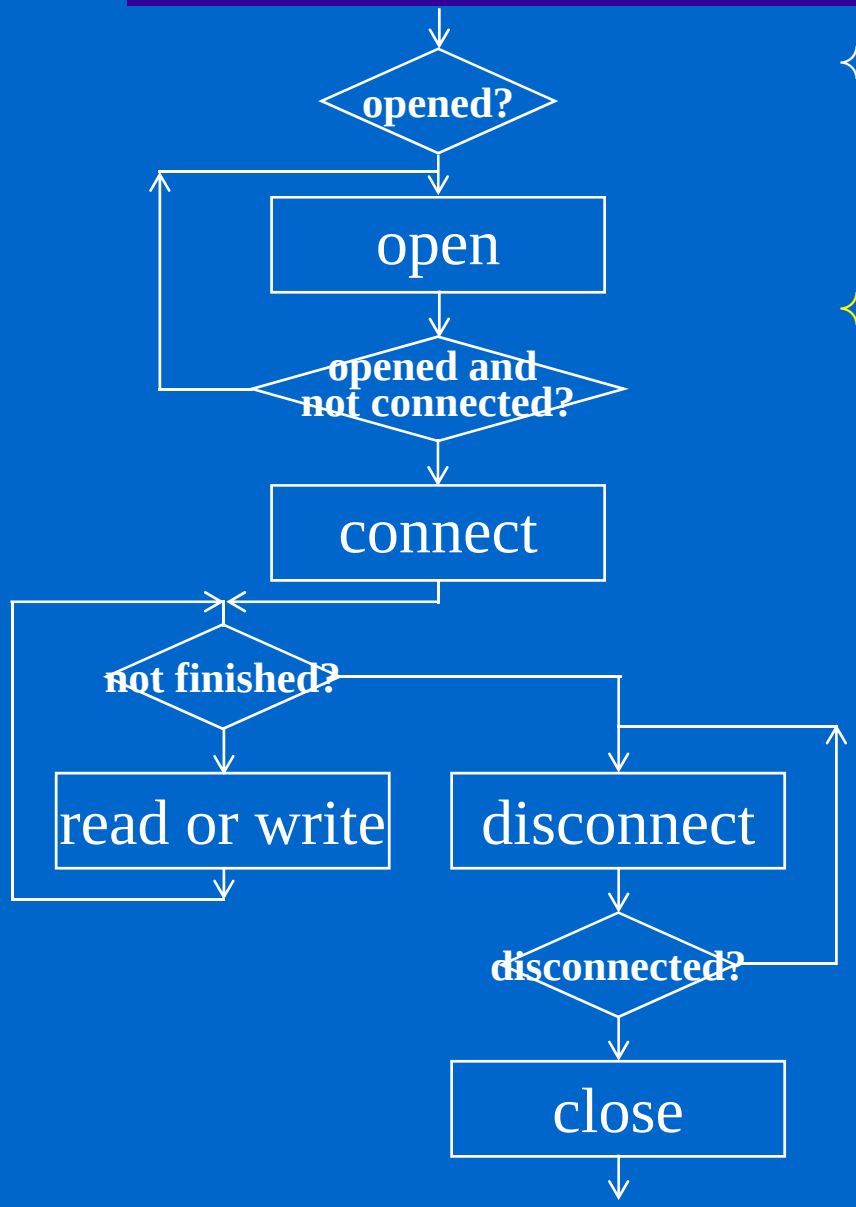
- This is the control flow for the TYPICAL / CORRECT usage of this NetCommStream object.
- Problems:

Control Flow Diagram of A Client



- This is the control flow for the TYPICAL / CORRECT usage of this NetCommStream object.
- Problems:
  - What if the client does not follow this advised procedure?
    Eg. Not opened but do the connect at the first step? Not connected but do the read/write at the second step?

Control Flow Diagram of A Client



- This is the control flow for the TYPICAL / CORRECT usage of this NetCommStream object.
- Problems:
  - What if the client does not follow this advised procedure?
    Eg. Not opened but do the connect at the first step? Not connected but do the read/write at the second step?
  - What if there are other possible usage patterns?
    e.g. Opened but find no peer to connect and then close immediately.
    Disconnected but find some other peer to connect.

Implementation of an FSM

- Use a single *enum* type of variable to represent the state of the system

  enum InternalStates {***Closed***, ***Opened***, ***Connected***};
  InternalStates **m_state**;

- In an OO system, objects communicate with each other through events. Take the event open and its handler open() as example:

  1. For each open message of each state in the diagram
  2. Implement the response in open()

```
void open() {
   if (m_state == Closed) {
      do_open();
      m_state = Opened;
   }
   else if (m_state == Opened) ;
   else if (m_state == Connected) ;
}
```

A systematic way of implementation for a state diagram

Implementation (cont'd)

```
void close() {
    if (m_state == Opened ) {
        do_close();
        m_state = Closed;
    }
}
```

Implementation (cont'd)

```
void close() {
    if (m_state == Opened ) {
        do_close();
        m_state = Closed;
    }
}

void connect() {
    if (m_state == Opened ) {
        do_connect();
        m_state = Connected;
    }
}
```

Implementation (cont'd)

```
void close() {
    if (m_state == Opened ) {
        do_close();
        m_state = Closed;
    }
}

void connect() {
    if (m_state == Opened ) {
        do_connect();
        m_state = Connected;
    }
}

void disconnect() {
    if (m_state == Connected ) {
        do_disconnect();
        m_state = Opened;
    }
}
```

Implementation (cont'd)

```
void close() {
    if (m_state == Opened ) {
        do_close();
        m_state = Closed;
    }
}

void connect() {
    if (m_state == Opened ) {
        do_connect();
        m_state = Connected;
    }
}

void disconnect() {
    if (m_state == Connected ) {
        do_disconnect();
        m_state = Opened;
    }
}
```

```
void read() {
    if (m_state == Connected )
        do_read();
}
```

Implementation (cont'd)

```
void close() {
    if (m_state == Opened ) {
        do_close();
        m_state = Closed;
    }
}

void connect() {
    if (m_state == Opened ) {
        do_connect();
        m_state = Connected;
    }
}

void disconnect() {
    if (m_state == Connected ) {
        do_disconnect();
        m_state = Opened;
    }
}
```

```
void read() {
    if (m_state == Connected )
        do_read();
}

void write() {
    if (m_state == Connected )
        do_write();
}
```

Modification over State Diagram

- If the system specification is modified such that it is allowed to close at the Connected state

Modification over State Diagram

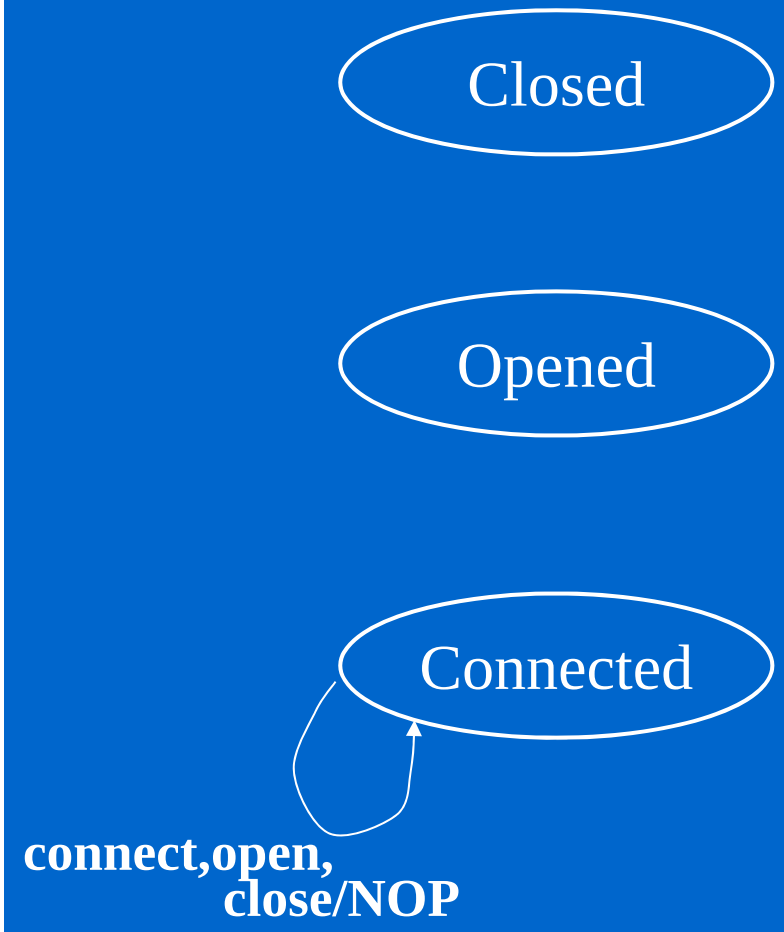
- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly

Modification over State Diagram

- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly

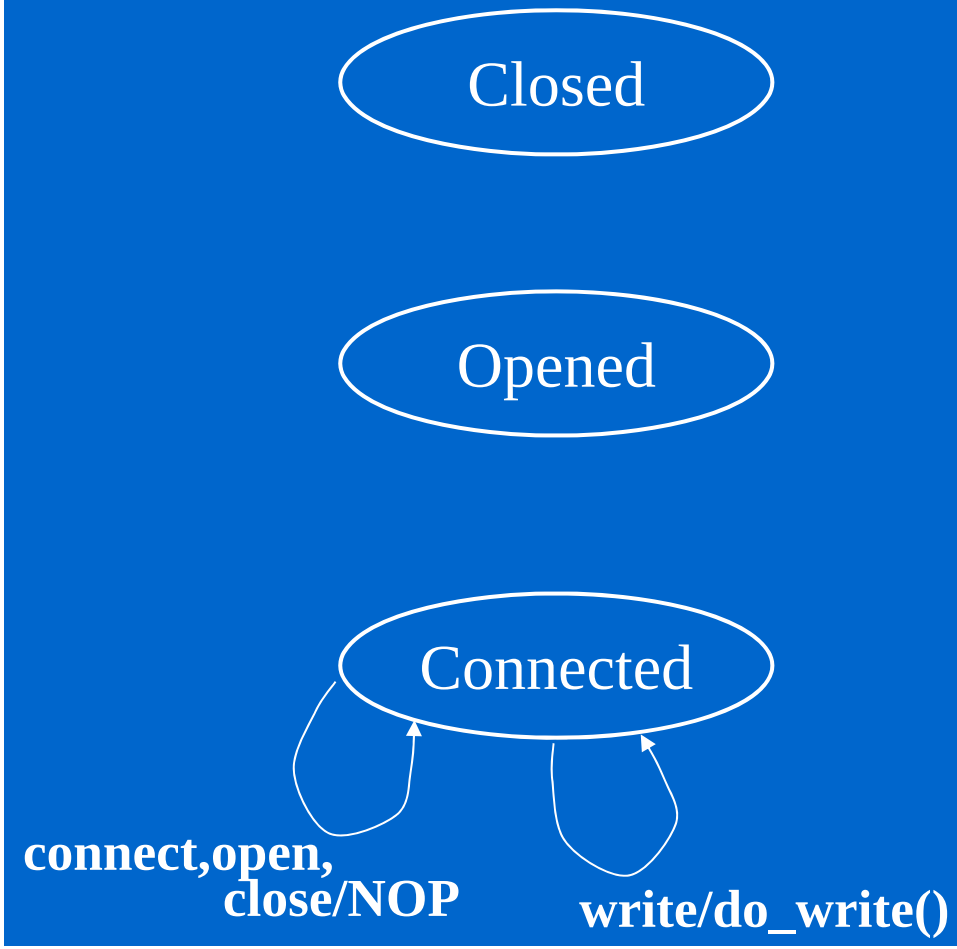
Modification over State Diagram

- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly



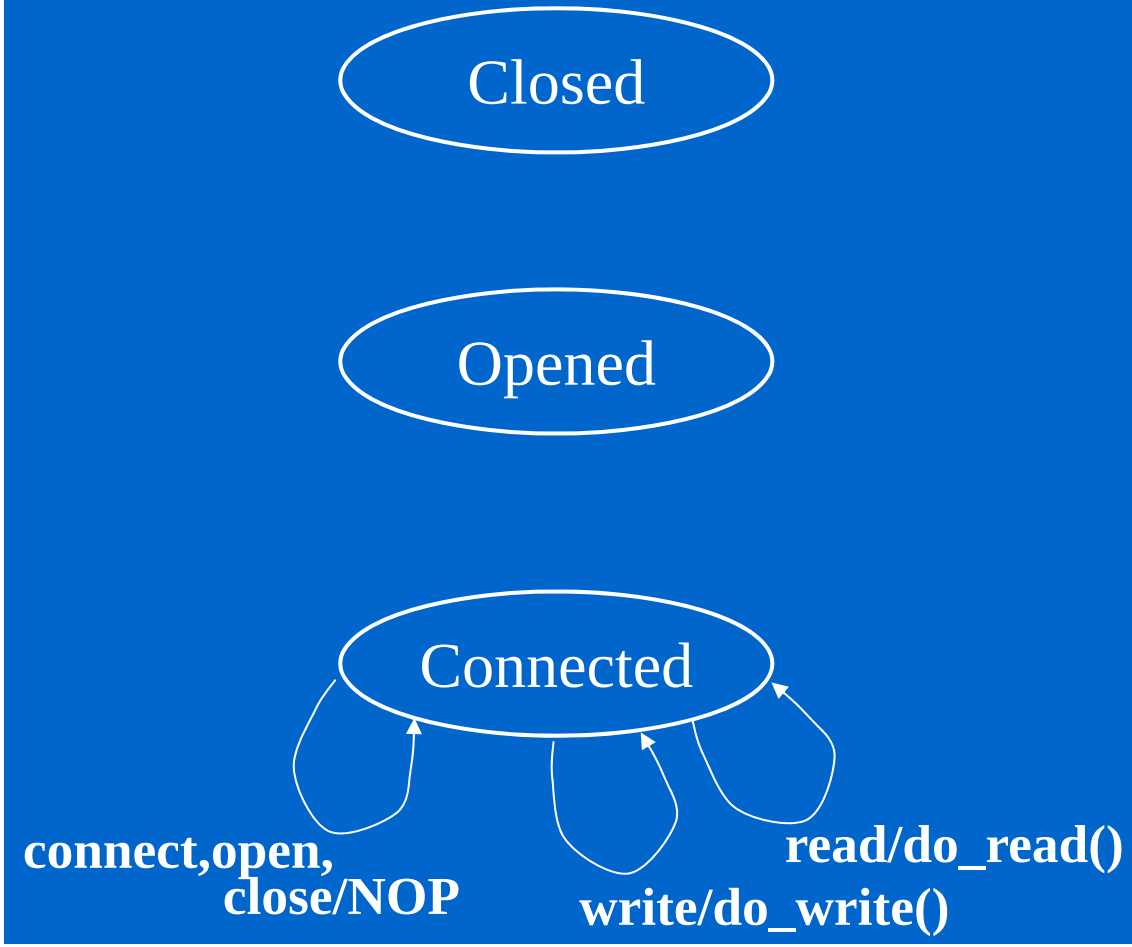
Modification over State Diagram

- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly



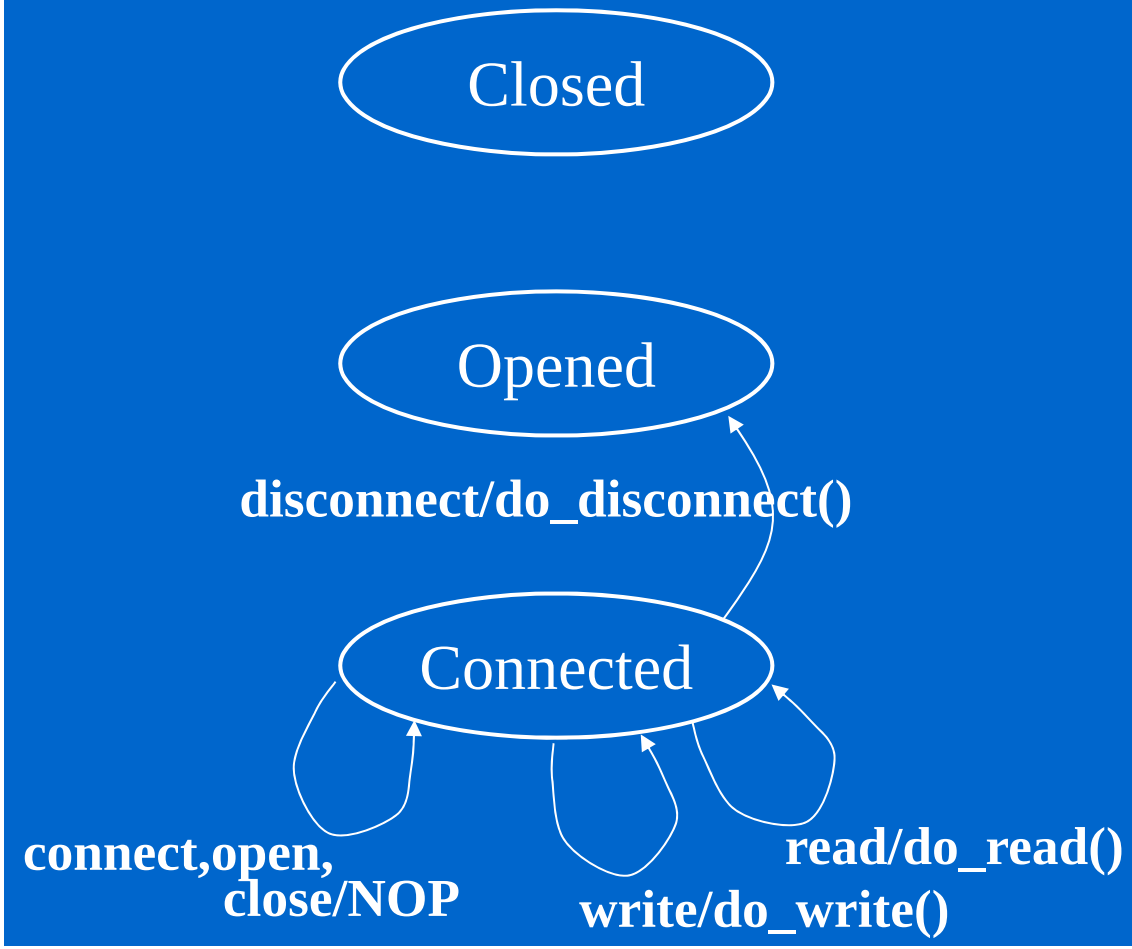
Modification over State Diagram

- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly



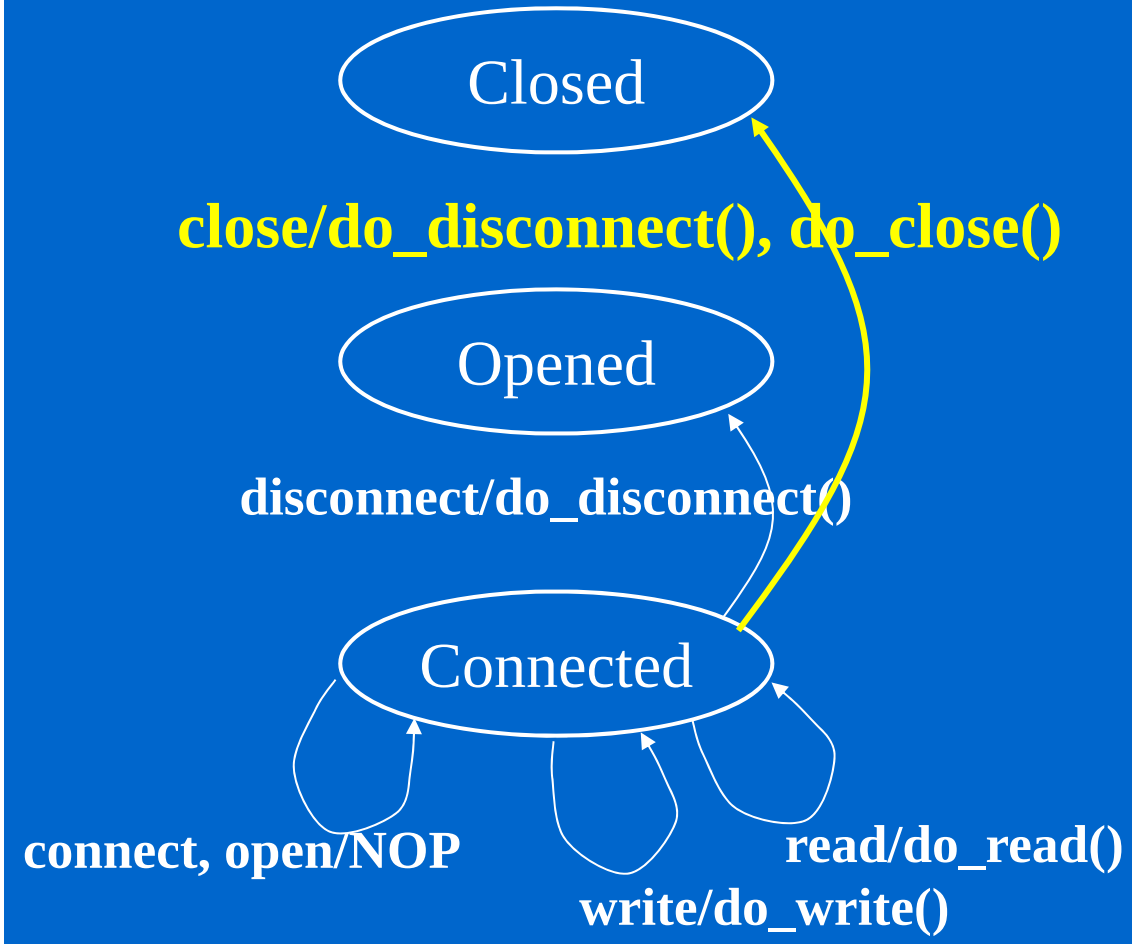
Modification over State Diagram

- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly



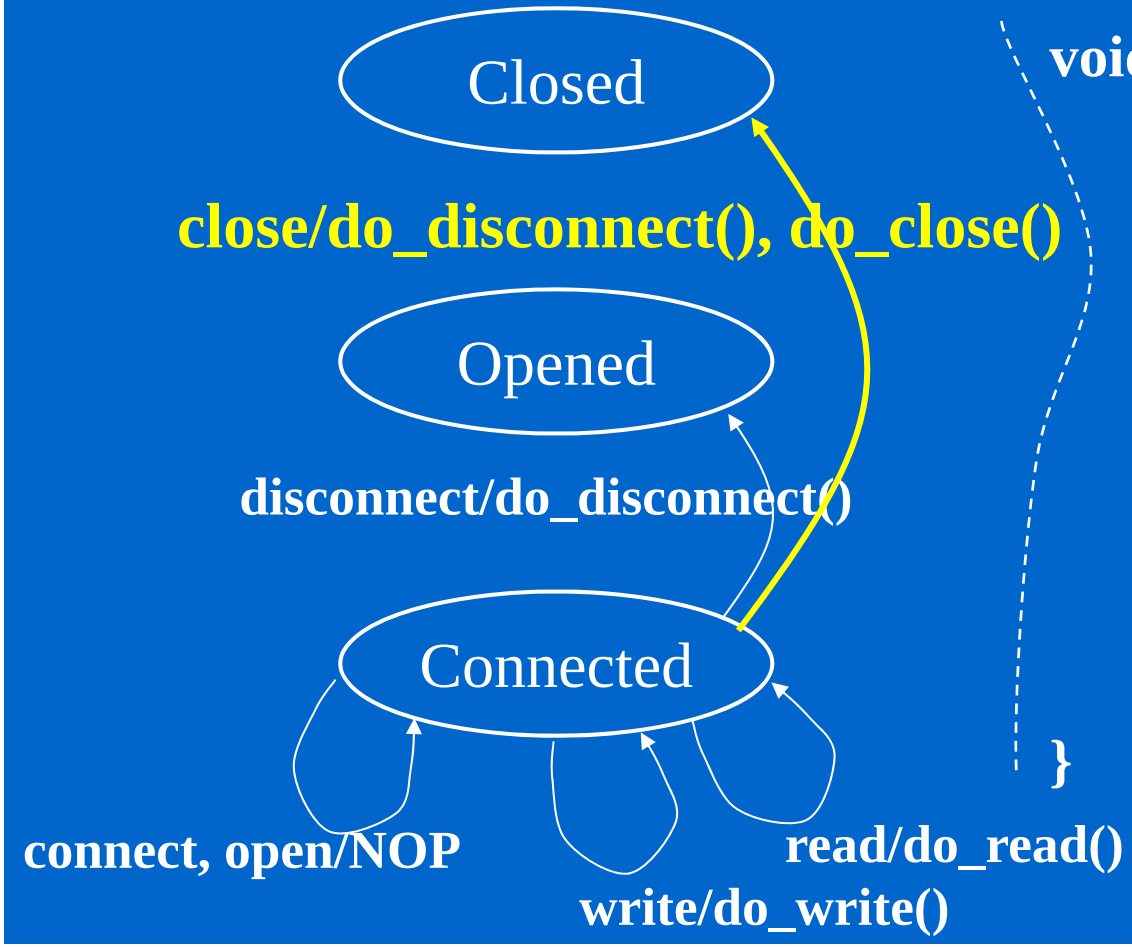
Modification over State Diagram

- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly



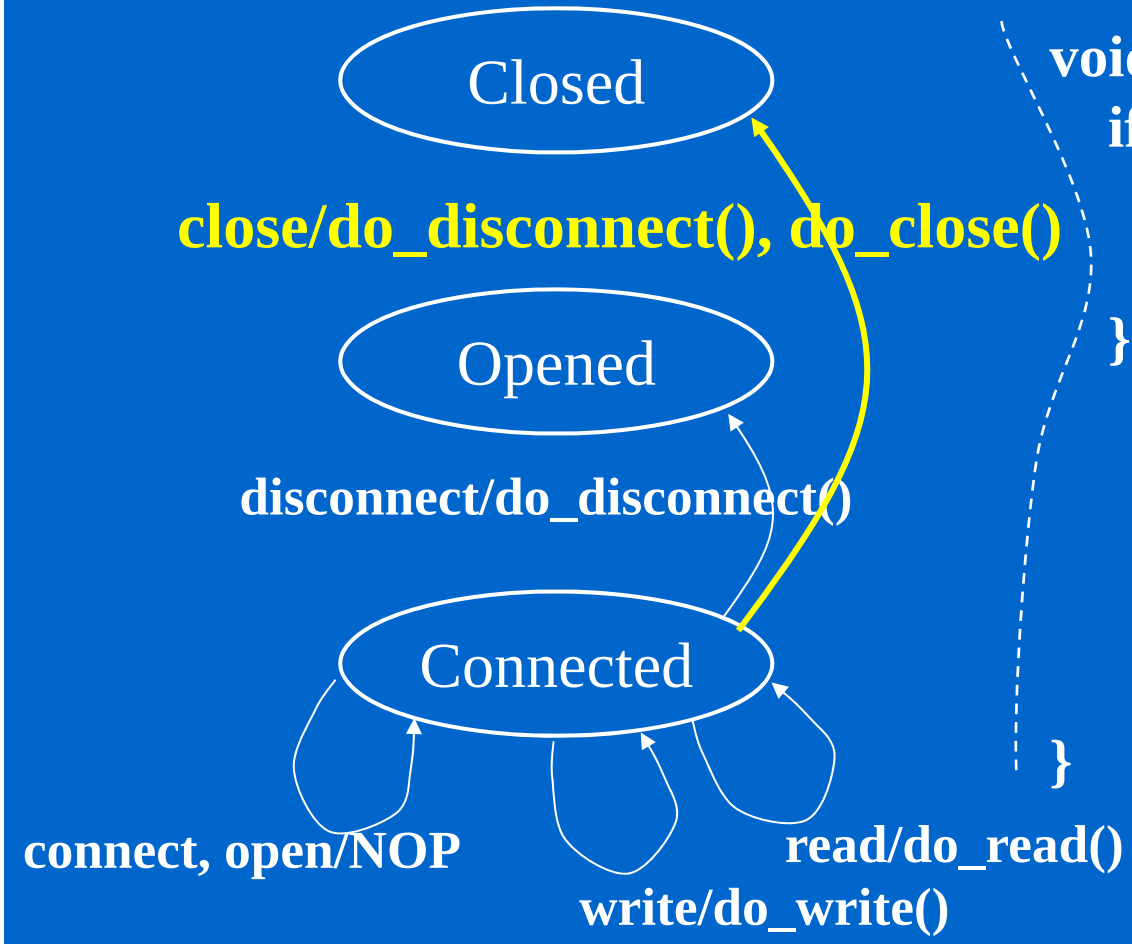
Modification over State Diagram

- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly



Modification over State Diagram

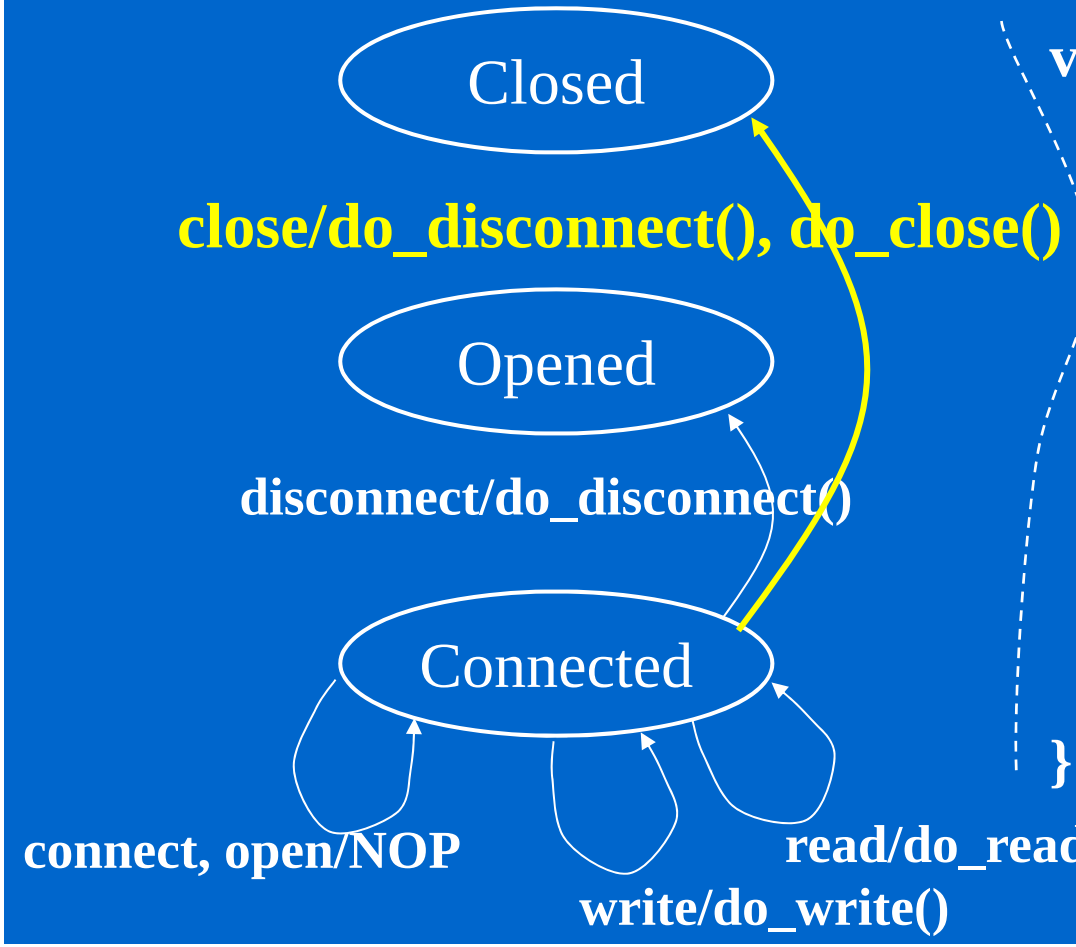
- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly




```
void close() {
    if (m_state == Opened) {
        do_close();
        m_state = Closed;
    }
}
```

Modification over State Diagram

- If the system specification is modified such that it is allowed to close at the Connected state
- It is a good idea to change the design on the state diagram directly




```
void close() {
    if (m_state == Opened) {
        do_close();
        m_state = Closed;
    }
    else if (m_state == Connected ) {
        do_disconnect();
        do_close();
        m_state = Closed;
    }
}
```

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

  enum **State** {***Closed***, ***Opened***, ***Connected***};

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

  enum **State** {***Closed***, ***Opened***, ***Connected***};
  enum **Event** {***open***, ***connect***, ***read***, ***write***, ***disconnect***, ***close***};

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

  enum **State** {***Closed***, ***Opened***, ***Connected***};
  enum **Event** {***open***, ***connect***, ***read***, ***write***, ***disconnect***, ***close***};

- Implement all actions as functions

  void do_open() { … }
  void do_connect() { … }
  void do_read() { … }
  void do_close() { … }
  void do_disconnect() { … }
  void do_write() { … }

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

  enum **State** {***Closed***, ***Opened***, ***Connected***};
  enum **Event** {***open***, ***connect***, ***read***, ***write***, ***disconnect***, ***close***};

- Implement all actions as functions

  void do_open() { … }
  void do_connect() { … }
  void do_read() { … }
  void do_close() { … }
  void do_disconnect() { … }
  void do_write() { … }

- Use a static **state** variable inside the Transition function

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

  enum **State** {***Closed***, ***Opened***, ***Connected***};
  enum **Event** {***open***, ***connect***, ***read***, ***write***, ***disconnect***, ***close***};

- Implement all actions as functions

  void do_open() { … }
  void do_connect() { … }
  void do_read() { … }
  void do_close() { … }
  void do_disconnect() { … }
  void do_write() { … }

- Use a static **state** variable inside the Transition function

  void Transition(Event **event**) {

  }

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

  enum **State** {***Closed***, ***Opened***, ***Connected***};
  enum **Event** {***open***, ***connect***, ***read***, ***write***, ***disconnect***, ***close***};

- Implement all actions as functions

  void do_open() { … }
  void do_connect() { … }
  void do_read() { … }
  void do_close() { … }
  void do_disconnect() { … }
  void do_write() { … }

- Use a static **state** variable inside the Transition function

```
void Transition(Event event) {
  static State state = Closed;
                                    }
```

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

```
enum State {Closed, Opened, Connected};
enum Event {open, connect, read, write, disconnect, close};
```

- Implement all actions as functions

```
void do_open() { … }          void do_close() { … }
void do_connect() { … }       void do_disconnect() { … }
void do_read() { … }          void do_write() { … }
```

- Use a static **state** variable inside the Transition function

```
void Transition(Event event) {
    static State state = Closed;
    switch (state) {
    }
}
```

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

  enum **State** {***Closed***, ***Opened***, ***Connected***};
  enum **Event** {***open***, ***connect***, ***read***, ***write***, ***disconnect***, ***close***};

- Implement all actions as functions

  void do_open() { … }
  void do_connect() { … }
  void do_read() { … }
  void do_close() { … }
  void do_disconnect() { … }
  void do_write() { … }

- Use a static **state** variable inside the Transition function

```
void Transition(Event event) {
    static State state = Closed;
    switch (state) {
    case Closed:
    case Opened:
    case Connected:
    }
}
```

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

```
enum State {Closed, Opened, Connected};
enum Event {open, connect, read, write, disconnect, close};
```

- Implement all actions as functions

```
void do_open() { … }          void do_close() { … }
void do_connect() { … }       void do_disconnect() { … }
void do_read() { … }          void do_write() { … }
```

- Use a static **state** variable inside the Transition function

```
void Transition(Event event) {
    static State state = Closed;
    switch (state) {
    case Closed:
        switch (event) {
        case open:
            do_open(); state = Opened;
        } break;
    case Opened:
                                    case Connected:
                                    }
}
```

Second Implementation

- Use two ***enum*** types for the **state** of the system and the **event**

```
enum State {Closed, Opened, Connected};
enum Event {open, connect, read, write, disconnect, close};
```

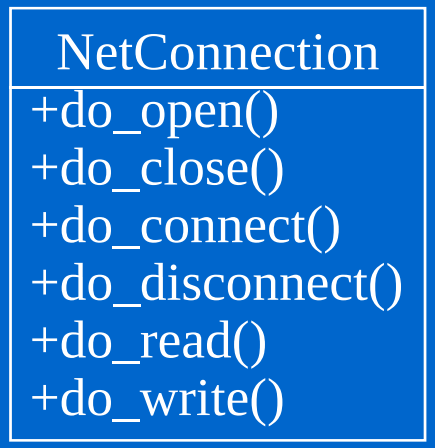
- Implement all actions as functions

```
void do_open() { … }          void do_close() { … }
void do_connect() { … }       void do_disconnect() { … }
void do_read() { … }          void do_write() { … }
```

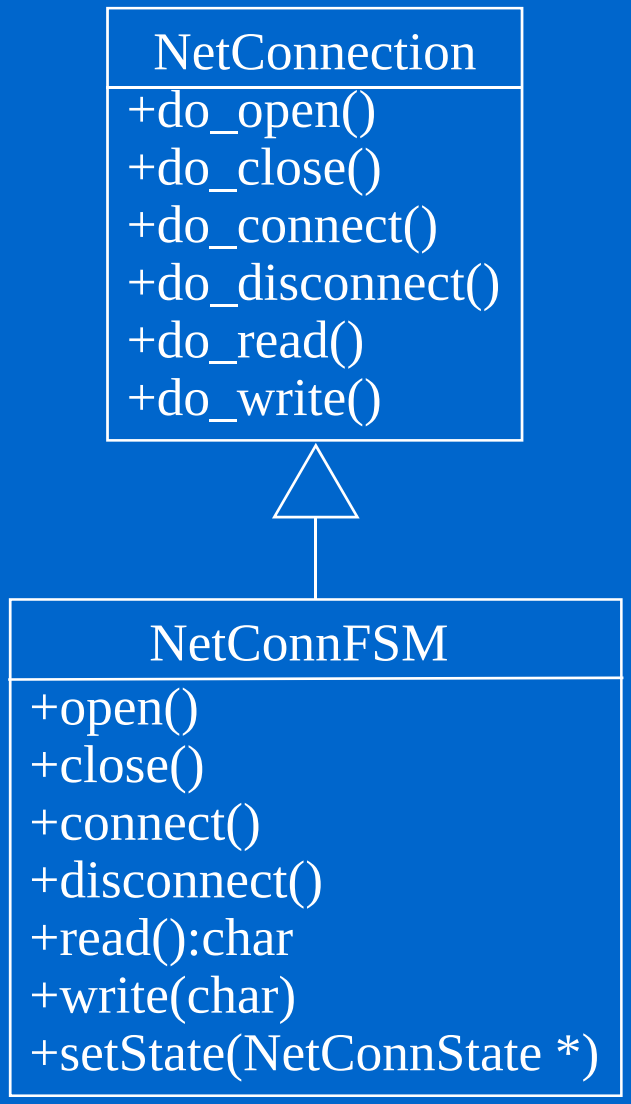
- Use a static **state** variable inside the Transition function

```
void Transition(Event event) {
    static State state = Closed;
    switch (state) {
    case Closed:
        switch (event) {
        case open:
            do_open(); state = Opened;
        } break;
    case Opened:
        switch (event) {
        case connect:
            …
        case close:
            …
        } break;
    case Connected:
        …
    }
}
```

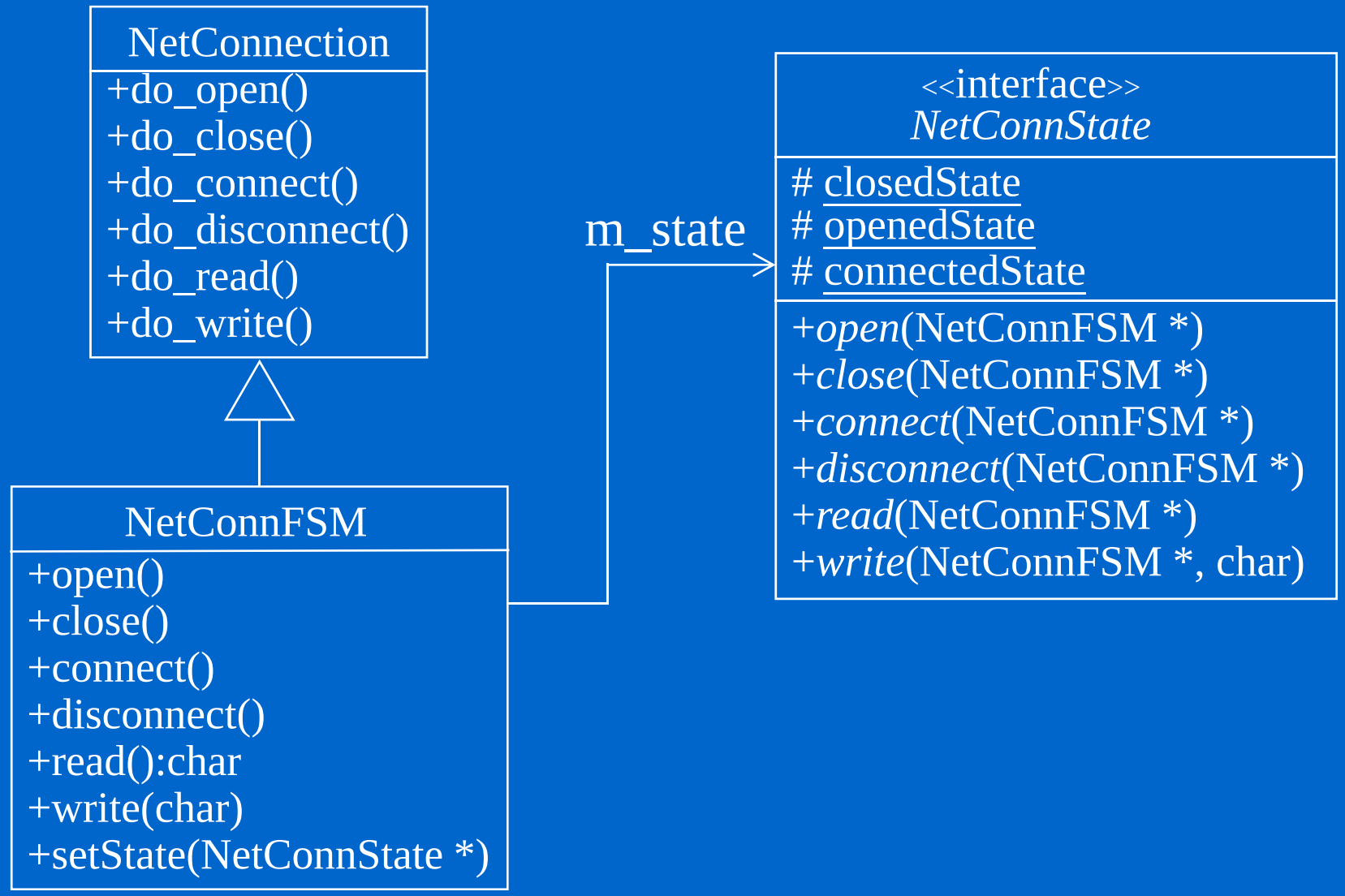
OO Way – the State Pattern



OO Way – the State Pattern

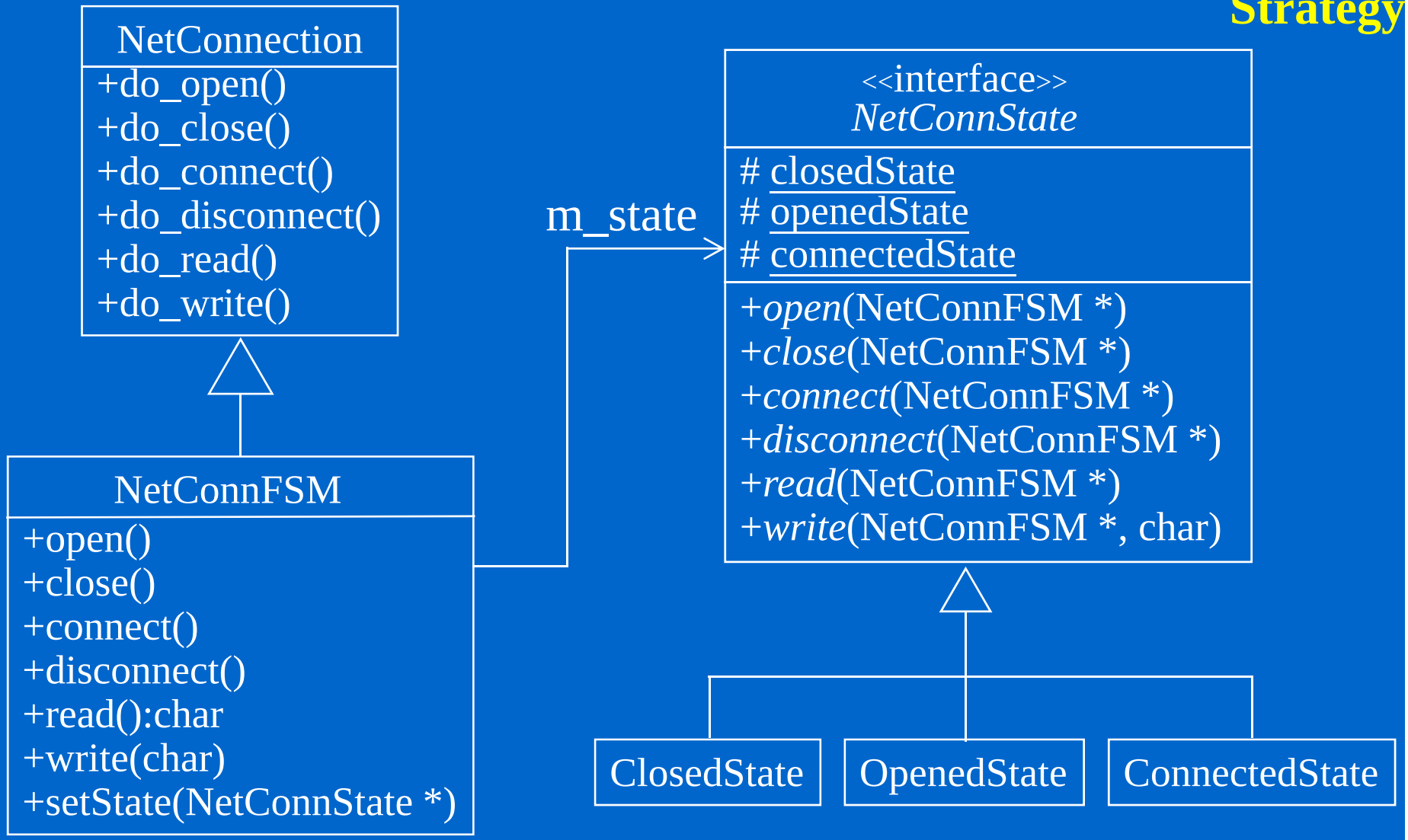


OO Way – the State Pattern



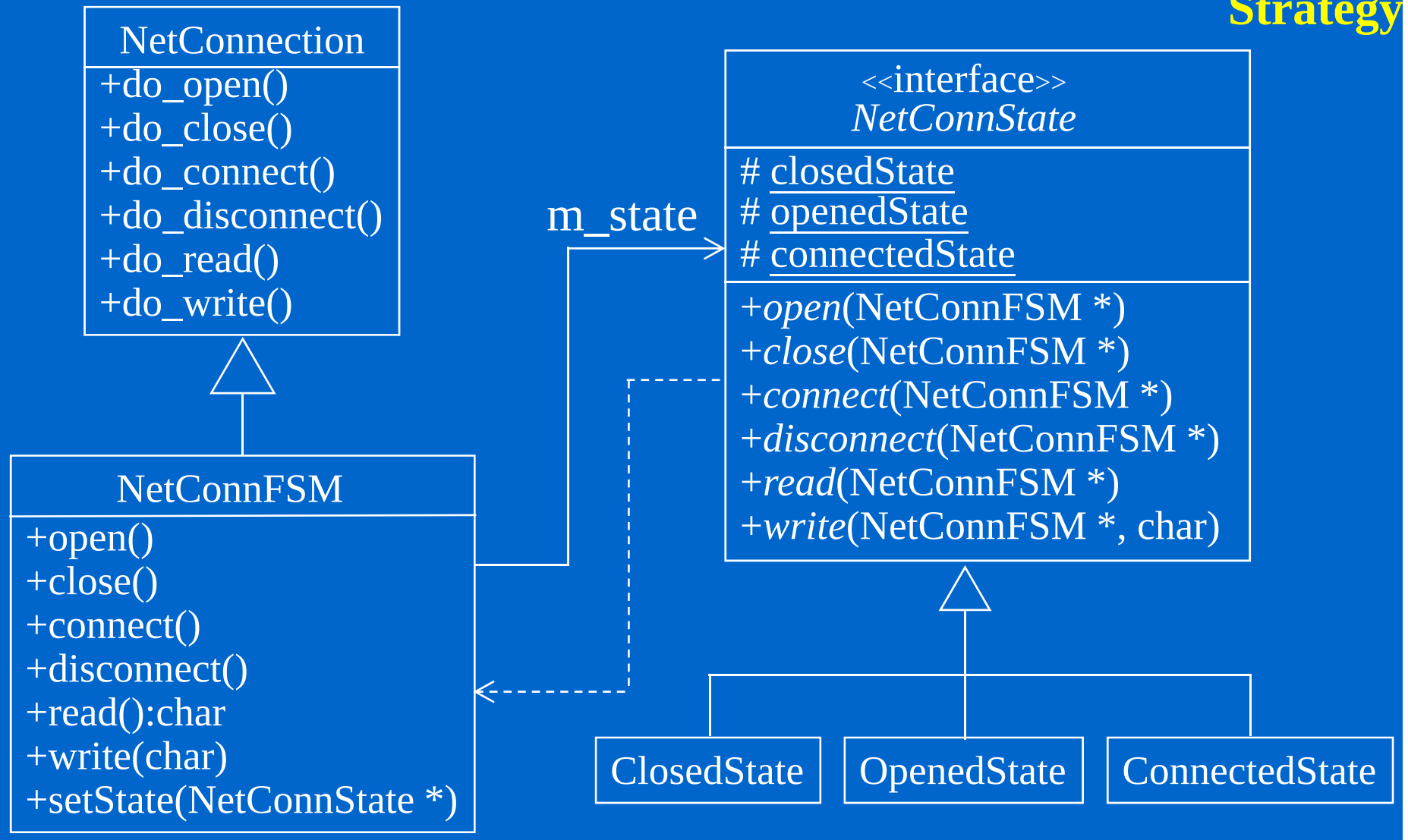
OO Way – the State Pattern

Strategy



OO Way – the State Pattern

Strategy



Implementation (1/4)

- Actual network operations

Implementation (1/4)

- Actual network operations

```
class NetConnection {
public:
    void do_open() { … }          void do_close() { … }
    void do_connect() { … }       void do_disconnect() { … }
    char do_read() { … }          void do_write(char x) { … }
};
```

Implementation (1/4)

- Actual network operations

```
class NetConnection {
public:
    void do_open() { … }          void do_close() { … }
    void do_connect() { … }       void do_disconnect() { … }
    char do_read() { … }          void do_write(char x) { … }
};
```

- Interface for all states

Implementation (1/4)

- Actual network operations

```
class NetConnection {
public:
    void do_open() { … }          void do_close() { … }
    void do_connect() { … }       void do_disconnect() { … }
    char do_read() { … }          void do_write(char x) { … }
};
```

- Interface for all states

```
class NetConnState {
public:
    virtual void open(NetConnFSM *) = 0;  virtual void close(NetConnFSM *);
    virtual void connect(NetConnFSM *);   virtual void disconnect(NetConnFSM *);
    virtual char read(NetConnFSM *);      virtual void write(NetConnFSM *, char);
protected:
    static ClosedState closedState;
    static OpenedState openedState;
    static ConnectedState connectedState;
};
```

Implementation (1/4)

- Actual network operations

```
class NetConnection {
public:
    void do_open() { … }          void do_close() { … }
    void do_connect() { … }       void do_disconnect() { … }
    char do_read() { … }          void do_write(char x) { … }
};
```

- Interface for all states

```
class NetConnState {
public:
    virtual void open(NetConnFSM *) = 0;  virtual void close(NetConnFSM *);
    virtual void connect(NetConnFSM *);   virtual void disconnect(NetConnFSM *);
    virtual char read(NetConnFSM *);      virtual void write(NetConnFSM *, char);
protected:
    static ClosedState closedState;
    static OpenedState openedState;
    static ConnectedState connectedState;
};
```

```
ClosedState NetConnState::closedState;
OpenedState NetConnState::openedState;
ConnectedState NetConnState::
                    connectedState;
```

Singleton

Implementation (2/4)

- the Finite State Machine

Implementation (2/4)

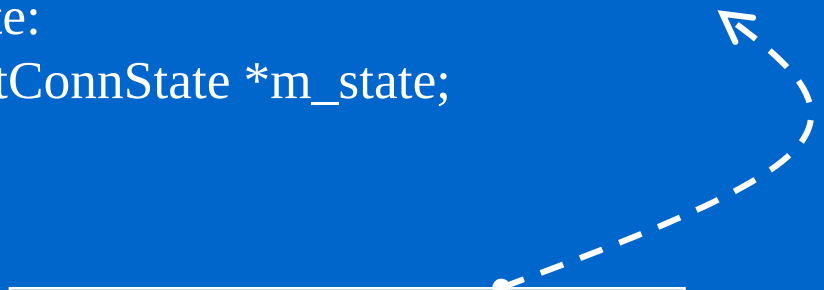
- the Finite State Machine

```
class NetConnFSM: public NetConnection {
public:
    NetConnFSM():m_state(&closedState) {}
    void setState(NetConnState *s) { m_state = s; }
    void open() { m_state->open(this); }
    void close() { m_state->close(this); }
    void connect() { m_state->connect(this); }
    void disconnect() { m_state->disconnect(this); }
    char read() { return m_state->read(this); }
    void write(char x) { m_state->write(this, x); }
private:
    NetConnState *m_state;
};
```

Implementation (2/4)

- the Finite State Machine

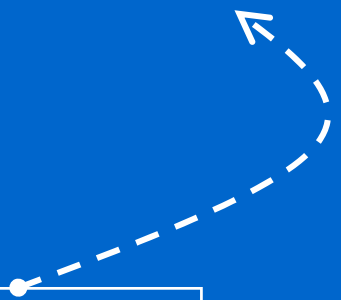
```
class NetConnFSM: public NetConnection {
public:
    NetConnFSM():m_state(&closedState) {}
    void setState(NetConnState *s) { m_state = s; }
    void open() { m_state->open(this); }
    void close() { m_state->close(this); }
    void connect() { m_state->connect(this); }
    void disconnect() { m_state->disconnect(this); }
    char read() { return m_state->read(this); }
    void write(char x) { m_state->write(this, x); }
private:
    NetConnState *m_state;
};
```



Implementation (2/4)

- the Finite State Machine

```
class NetConnFSM: public NetConnection {
public:
    NetConnFSM():m_state(&closedState) {}
    void setState(NetConnState *s) { m_state = s; }
    void open() { m_state->open(this); }
    void close() { m_state->close(this); }
    void connect() { m_state->connect(this); }
    void disconnect() { m_state->disconnect(this); }
    char read() { return m_state->read(this); }
    void write(char x) { m_state->write(this, x); }
private:
    NetConnState *m_state;
};
```



Strategy, delegation of events
Closed for modification

- Usage:

Implementation (2/4)

- the Finite State Machine

```
class NetConnFSM: public NetConnection {
public:
    NetConnFSM():m_state(&closedState) {}
    void setState(NetConnState *s) { m_state = s; }
    void open() { m_state->open(this); }
    void close() { m_state->close(this); }
    void connect() { m_state->connect(this); }
    void disconnect() { m_state->disconnect(this); }
    char read() { return m_state->read(this); }
    void write(char x) { m_state->write(this, x); }
private:
    NetConnState *m_state;
};
```

- Usage:

```
void main() {
    NetConnFSM conn_obj;
    conn_obj.open();
    conn_obj.connect();
    int x=conn_obj.read();
        …
    conn_obj.write(x);
        …
    conn_obj.disconnect();
    conn_obj.close();
}
```

Implementation (2/4)

- the Finite State Machine

```
class NetConnFSM: public NetConnection {
public:
    NetConnFSM():m_state(&closedState) {}
    void setState(NetConnState *s) { m_state = s; }
    void open() { m_state->open(this); }
    void close() { m_state->close(this); }
    void connect() { m_state->connect(this); }
    void disconnect() { m_state->disconnect(this); }
    char read() { return m_state->read(this); }
    void write(char x) { m_state->write(this, x); }
private:
    NetConnState *m_state;
};
```

Strategy, delegation of events
Closed for modification

- Usage:

```
void main() {
    NetConnFSM conn_obj;
    conn_obj.open();
    conn_obj.connect();
    int x=conn_obj.read();
      …
    conn_obj.write(x);
      …
    conn_obj.disconnect();
    conn_obj.close();
}
```

real event sequence

Implementation (3/4)

- Actual states

```
class ClosedState: public NetConnState {
public:
    void open(NetConnFSM *fsm) {
        fsm->do_open();
        fsm->setState(&openedState);
    }
    void connect(NetConnFSM *) {}
    char read(NetConnFSM *) { return 0; }
};
```

```
void close(NetConnFSM *) {}

void disconnect(NetConnFSM *) {}
void write(NetConnFSM *, char) {}
```

Implementation (3/4)

- Actual states

```
class ClosedState: public NetConnState {
public:
    void open(NetConnFSM *fsm) {
        fsm->do_open();
        fsm->setState(&openedState);
    }
    void connect(NetConnFSM *) {}
    char read(NetConnFSM *) { return 0; }
};
```

```
void close(NetConnFSM *) {}

void disconnect(NetConnFSM *) {}
void write(NetConnFSM *, char) {}
```

```
class OpenedState: public NetConnState {
public:
    void open(NetConnFSM *) {}
    void close(NetConnFSM *fsm) {
        fsm->do_close();
        fsm->setState(&closedState);
    }
    char read(NetConnFSM *) { return 0; }
    void write(NetConnFSM *, char) {}
};
```

```
void connect(NetConnFSM *fsm) {
    fsm->do_connect();
    fsm->setState(&connectedState);
}
void disconnect(NetConnFSM *) {}
```

Implementation (4/4)

```
class ConnectedState: public NetConnState {
public:
    void open(NetConnFSM *) {}
    void close(NetConnFSM *) {}
    void connect(NetConnFSM *) {}
    void disconnect(NetConnFSM *fsm) {
        fsm->do_disconnect();
        fsm->setState(&openedState);
    }
    char read(NetConnFSM *fsm) {
        return fsm->do_read();
    }
    void write(NetConnFSM *fsm, char x) {
        fsm->do_write(x);
    }
};
```

Implementation (4/4)

```
class ConnectedState: public NetConnState {
public:
    void open(NetConnFSM *) {}
    void close(NetConnFSM *) {}
    void connect(NetConnFSM *) {}
    void disconnect(NetConnFSM *fsm) {
        fsm->do_disconnect();
        fsm->setState(&openedState);
    }
    char read(NetConnFSM *fsm) {
        return fsm->do_read();
    }
    void write(NetConnFSM *fsm, char x) {
        fsm->do_write(x);
    }
};
```

- Implemented through the “Strategy” and “Singleton” patterns

Implementation (4/4)

```
class ConnectedState: public NetConnState {
public:
    void open(NetConnFSM *) {}
    void close(NetConnFSM *) {}
    void connect(NetConnFSM *) {}
    void disconnect(NetConnFSM *fsm) {
        fsm->do_disconnect();
        fsm->setState(&openedState);
    }
    char read(NetConnFSM *fsm) {
        return fsm->do_read();
    }
    void write(NetConnFSM *fsm, char x) {
        fsm->do_write(x);
    }
};
```

- Implemented through the “Strategy” and “Singleton” patterns
- Object Mentor Finite State Machine Compiler for Java/C++ code
  http://www.objectmentor.com/resources/bin/smcJava.zip