State Diagram

C++ Object Oriented Programming
Pei-yih Ting
93/04 NTOU CS

Contents

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)

- When using an ifstream object for file input, a read operation for an integer might not succeed 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 of it only when the stack is not empty.
- An editor has two input modes: insert or overwrite. In the insert mode, the characters inputted from the keyboard are inserted right before the cursor. In the overwrite move, the characters inputted 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.

2. The timing of messages to an object with various internal states is important and determines the object's responses.

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 (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 the 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:
   ...
};
```

- **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.

Correct usage:

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

Incorrect usage:

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

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?

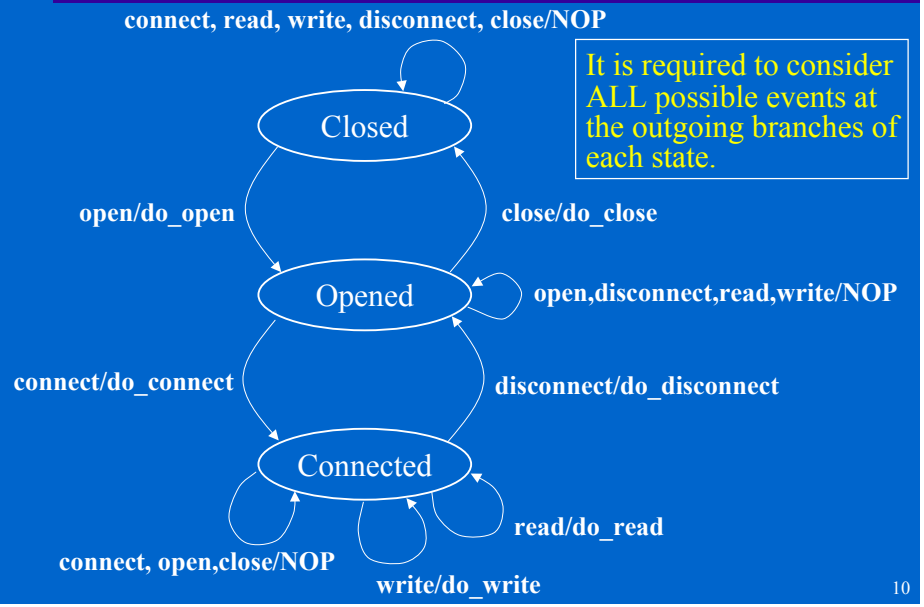
Explicit State

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

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

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

State Diagram

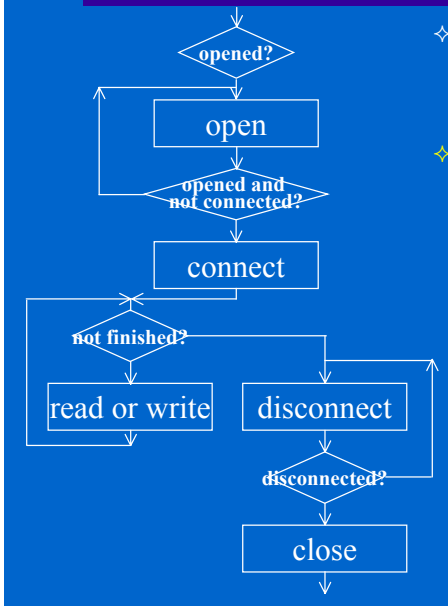


State Diagram (cont'd)

- Advantages:
  - Show only necessary 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



- 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 open but do the connect at the first step? Not connect but do the read/write at the second step?
  - What if there are other possible usage patterns?
    Eg. Opened but find no peer to connect and then close immediately.
    Disconnected but find some other peer to connect.

Implementation of the State Diagram

- Use a single *enum* type of variable to represent the state
  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 code implementation from a state diagram

Implementation of the State Diagram

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

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

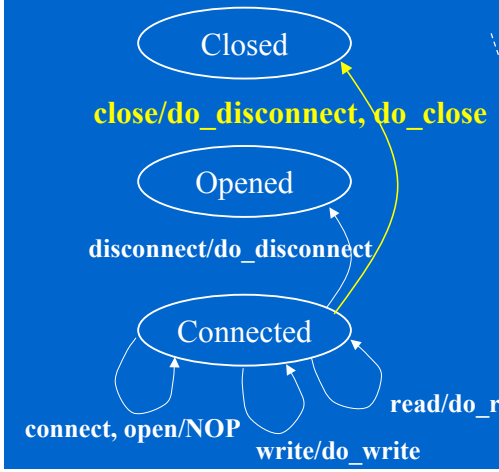
```
void disconnect()
{
   if (m_state == Connected )
   {
      do_disconnect();
      m_state = Opened;
   }
}

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
- 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;
   }
}
```