Parallel Notation is better than standard (combined) state machine for this example:

Note: BU is the initial state, and represents NOT BOLD, NOT UNDERLINE & NOT ITALICS.

For n features (n=3) (ON/OFF) we have:

- \(2^n\) states
- \(6 \times 2^n\) transitions

Thus, prefer parallel SM.

Example: TRAFFIC LIGHTS

- We use the same name for events A & B: thus these events must occur in both state machines in parallel.

Thus for A to occur:

- L1 must be in Y,
- L2 must be in R

After A occurs:

- L1 is in R
- L2 is in G

\(\Rightarrow\) We are able to show that both lights cannot be green using coupled events.

Example: TEXT EDITOR

Parallel State Machines

- Can be useful in making concise representations of independent components of a system.

Example:

1. BOLD
2. ITALICS
3. UNDERLINE

For n features (n=3) (ON/OFF) we have:

- n state machines
- \(2^n\) transitions
Abstract Classes vs Interfaces

1. Can provide method body, or choose to not provide method body.
2. Force subclasses to "inherit" from this class; this prevents them from subclassing any other class.
3. Make little sense for functionality; e.g. Comparable®

Interfaces
1. Can only provide method declarations, not method bodies.
2. Create no inheritance constraints. Implementor can implement other interfaces & subclass another class & implement other interfaces.
3. Great for specifying functionality; e.g. Comparable®

Example
interface MyInputStream {
    int read();
    int read(byte[] buf);
}

abstract class MyInputStream {
    abstract public int read(byte[] buf);
    public int read();
    int bytesRead = 0;
    while (bytesRead < buf.length) {
        int next = read();
        if (next == -1) break;
        byte [bytesRead++ = next;]
    }
    return bytesRead;
}

Best Approach:
1. Use both Interfaces & Abstract classes ⇒ FLEXIBILITY for subclasses.
interface MyInputStream {
    int read();
    int read(byte[] buf);
}
abstract class AbstractMyInputStream implements MyInputStream {
    abstract public int read(byte[] buf) {
        if (bytesRead < buf.length) {
            int next = read();
            if (next == -1) break;
            bytesRead += next;
        }
    }
}

Ex: A student is a special type of Person ⇒ Inheritance makes sense.
Ex: To sort objects, we need a "<" operator. Doesn't exist for String or Student:
⇒ Use "Comparable" interface:
interface Comparable {
    int compareTo(Object o);
}
-1 if this < 0
0 if this = 0
1 if this > 0
⇒ can use Comparable to sort objects.

In general:
A is a special kind of B ⇒ consider inheritance.
e.g. Student Person
A is something else; A implements some functionality
⇒ consider interfaces

REUSE CODE
 DECLARE

3 DECOUPLING, MODULE DEPENDENCY DIAGRAM

Backup Solution:
- We want to transfer files to/from:
  1. HARD_DISK
  2. USB
  3. CD
  4. DVD

  e.g. HARD_DISK → USB
  USB → CD
  USB → DVD
  HARD_DISK → HARD_DISK

POOR DESIGN:

```java
class Backup {
    // other stuff
    public void performBackup() {
        // pseudo code:
        while (allFileNotDone) {
            readFile(in, fileNum);
            writeFile(out, fileNum);
            fileNum++;
        }
    }

    private readFile(Medium in, int fileNum) {
        switch (in) {
            case (HARD_DISK):
                // code to read from HARD_DISK
                break;
            case (USB):
                // code to read from USB
                break;
            case (CD):
                // code to read from CD
                break;
            case (DVD):
                // code to read from DVD
                break;
            default:
                throw new IllegalArgumentException();
        }
    }

    private writeFile(Medium out, int fileNum) {
        // Another massive switch
        // statement.
    }

    // we will see performBackup()
    // next.
}
```

What's bad about this design?

1. Ugly code: massive switch statements
2. Module Dependency Diagrams:
   - Our backup software "depends" on DVD, USB, CD, HARD_DISK, because it needs to know about their implementations.
   - If Hard Disks change, or CDs change, we need to change our code and re-release it!
   - If some new Medium (e.g., BLUE Ray/ DROPBox) is created, our code cannot deal with it; we need to change our code and re-release it!
What's wrong with this design?

Long story short:

Our Backup class is strongly coupled with the details of various input/output mediums.

Better Design:

```
class Backup {
    private Medium in;
    private Medium out;

    public Backup(Medium in, Medium out) {
        this.in = in;
        this.out = out;
    }

    public performBackup() {
        // pseudocode:
        while (files still left) {
            in.readFile(fileNum);
            out.writeFile(fileNum);
            fileNum++;
        }
    }
}
```

- Notice how much simpler the code is.
- Our Backup class exports an interface (Medium) & it is someone else's job to create implementations of that interface (third parties).
- If Hard Disks change, or we create new mediums such as BLUERAY or DROPBOX, our code doesn't need to change; though someone needs to implement/change implementations of Medium.

A Now, our Backup class is decoupled from which input/output medium it is using. It only cares about readFile() and writeFile().

Common Pattern:

```
A
B C D
```

Decoupling:

```
A
B C D...
```
**One trick for today:**
Subclass a class to collect information useful for testing!

E.g. class SortedList  
void add (int x) {  
  // add x to list  
  ArrayList<Integer> get() {  
    // code to return sorted list  
  }

To test this class, we could create a test case in which:
- we create a sorted list.
- we add 4, 3, 1, 2 to it.
- we call get and check that we get: 1, 2, 3, 4 in order.

This tests correctness, but what if we want to test performance?

E.g. what if SortedList has a protected sort() method, which calls a protected swap() method, and we want to count the # of swaps?

**Solution:**

```java
class InstrumentedSorted {  
  extends SortedList {  
    private int numSwaps = 0;  
    @Override  
    protected void swap() {  
      numSwaps++;  
      super.swap();
    }
    public getNumSwaps() {  
      return this.numSwaps;
    }
    This test class does not change our original class, but allows us to track the # of swap operations we perform:
    - we create an instrumented sorted list
    - we add items, get back a list & make sure it's sorted.
    - we get # of swaps & make sure they make sense to us.
```