FSMs and the State Design Pattern

State design pattern, implementing FSMs by the design pattern

Text processing FSM

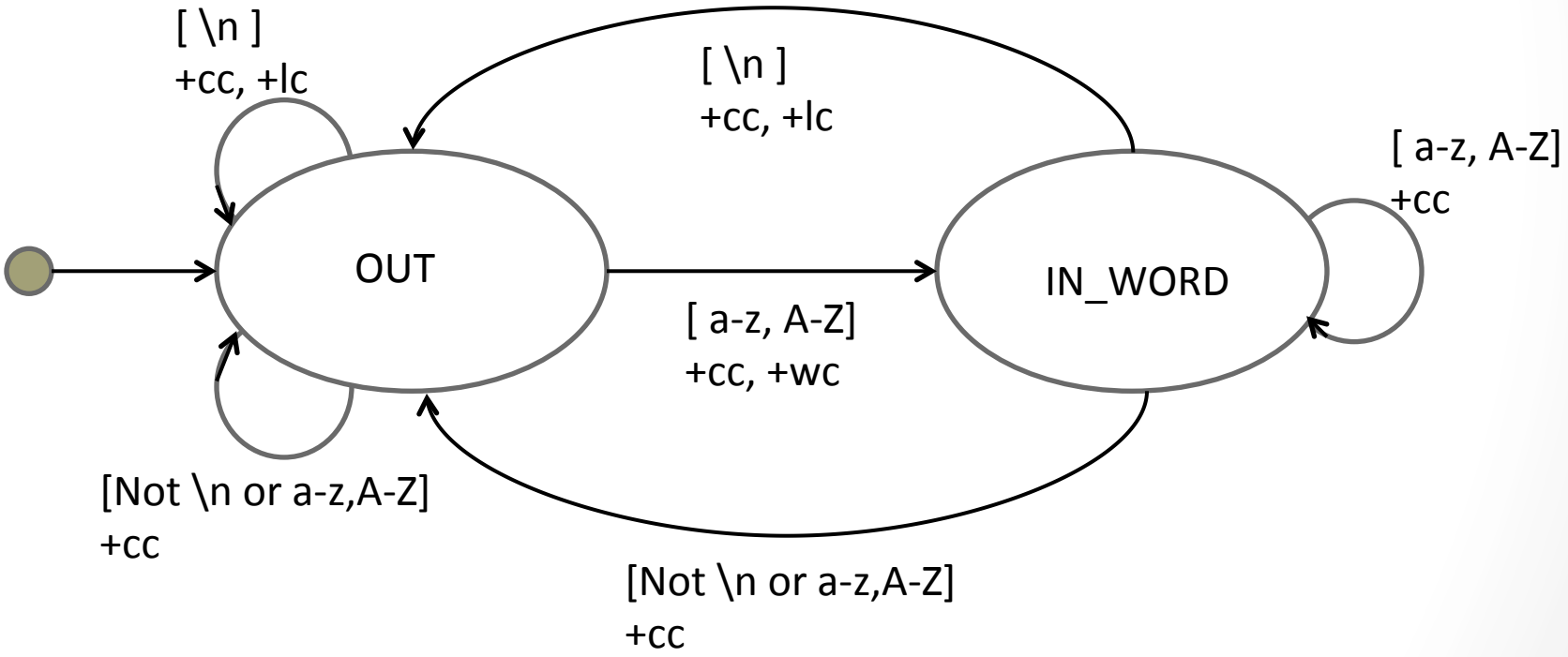
Problem: Counting characters, words, lines.

> " A word is a maximal non-empty sequence of alphabetic characters."

- We can build a FSM as an algorithm to count
  - cc : character count
  - wc: word count
  - lc: line count
- The FSM will have two states: IN_WORD and OUT

FSM diagram

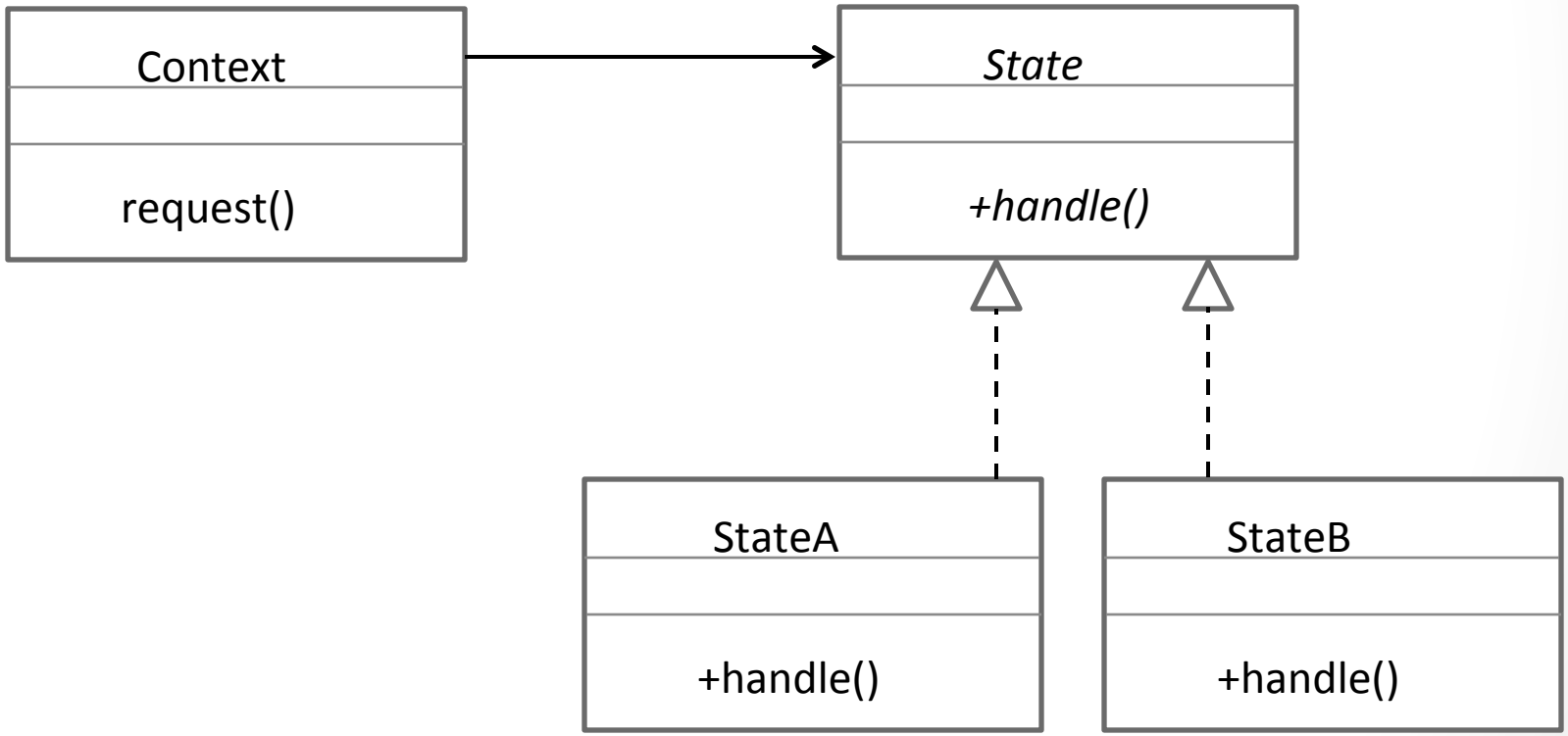
This is the simple FSM diagram from the previous lecture.



Translating FSMs into code

Method 1: Use a while loop to go through each input, with an embedded switch statement to handle the states and transitions.

Method 2: Apply the State Design Pattern to model states and transitions.



State Design Pattern

- Previous examples of FSMs and the FSM handout are not object oriented.
  - Focus is on tracing an FSM with a while-switch idiom for different inputs.
  - Can get complicated with complicated tests. Easy to make mistakes.
- State pattern is an OO solution to FSM implementation
  - The context is an object.
  - The object's behavior depends on state. The object changes its behavior at runtime based on state.
  - Alternative to conditionals with redundant code – treat states as objects that can vary independently from other objects.

Implementing the State pattern

- Define a State interface or abstract class.
  - Declare a method for each transition that the state could support.
- Define a concrete class to implement/extend State for each state in the FSM.
  - Define the behavior of the FSM for that state.
  - Note: These classes can be standalone or they can be inner classes.
- Define the Context
  - This is the part of the pattern that clients will use.
  - Maintains state instances. These state instances delegate behavior to the appropriate state class.

Define the State interface

Transitions are determined by the input. The interface declares the "handle" methods.

```
public interface State {
   /**
    * Do transition for a-z, A-Z
    */
   void handleAlphabetic();
   /**
    * Do transition for \n
    */
   void handleNewline();
   /**
    * Do transition for any other character.
    */
   void handleOther();
}
```

Define the Context

```
public class WordCounter {
   private final State inWord = new InWord();
   private final State out = new Out();
   private State state = out;

   private int wordCount;
   private int lineCount;
   private int characterCount;

   public int getWordCount() { return wordCount; }
   public int getLineCount() { return lineCount; }
   public int getCharacterCount() { return characterCount; }

   // "request" method, which manages the counts, goes here

   // InWord: Inner class goes here
   // Out: Inner class goes here
```

Define first State class

InWord goes inside **WordCounter** (the Context class).

```
private class InWord implements State {
    @Override
    public void handleAlphabetic() {
        characterCount++;
    }
    @Override
    public void handleNewline() {
        state = out;
        lineCount++; characterCount++;
    }
    @Override
    public void handleOther() {
        state = out;
        characterCount++;
    }
}
```

Define second State class

Out goes inside **WordCounter** (the Context class).

```
private class Out implements State {
   @Override
   public void handleAlphabetic() {
      state = inWord;
      characterCount++; wordCount++;
   }
   @Override
   public void handleNewline() {
      lineCount++; characterCount++;
   }
   @Override
   public void handleOther() {
      characterCount++;
   }
}
```

Insert the "request" method

```
public class WordCounter {
   // Counters and State instance variables defined here

   public void countWords(String s) {
      wordCount = 0;
      lineCount = 0;
      characterCount = 0;
      char ch;
      for (int k = 0; k < s.length(); k++) {
         ch = s.charAt(k);
         if (Character.isLetter(ch))
             state.handleAlphabetic();
         else if (ch == '\n')
             state.handleNewline();
         else
             state.handleOther();
      }
      state = out;
   }

   // Inner classes (InWord and Out) go here
```

And now a simple client

```
public class ClientCode {

public static void main(String[] args) {
   WordCounter wc = new WordCounter();
   wc.countWords("One fish.\n Two fish.\n Red fish.\n”);
   System.out.println("lines: " + wc.getLineCount());
   System.out.println("words: " + wc.getWordCount());
   System.out.println("characters: " + wc.getCharacterCount());
 }
}
```

lines: 3
words: 6
characters: 32

References

- Sarah Heckman, CSC 216 Slides.
- Heckman References:
  - “Finite-State Machines,” by Suzanne Balik and Matthias Stallmann
  - Freeman and Freeman, *Head First Design Patterns*, O’Reily, 2004
    - Chapter 10: The State PatternGamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison-Wesley, 1995.