LRStruct: **An Enhanced** AList

- Consider the following set of operations:

```
LRStruct L = new LRStruct();
LRStruct M = L;
L.insertFront(new Integer(7));
L.insertFront(new Integer(3));
L.removeFront();
L.removeFront();
```

- See the handout for their effects on the structure.

Program #1: Hangman

```
  ---
 /   \
 |
 |
 |                    - - - - - - - - - - - -
 |
 |        (secret word: p o l y m o r p h i s m)
 |
-+------
```

Program #1: Hangman (cont.)

```
  ---
 /   \
 |    O
 |   -+-
 |    |               p o _ _ m o _ p _ i _ m
 |   / \
 |          Player loses!
 |
-+------
```

Program #1: Hangman (cont.)

- In the hangman game, a character in the target word can be either in the *hidden* state or the *visible* state.
  - When it is *hidden*, it converts to a `String` as "_".
  - When it is *visible*, it converts to a `String` as the `String` consisting of its actual character value.

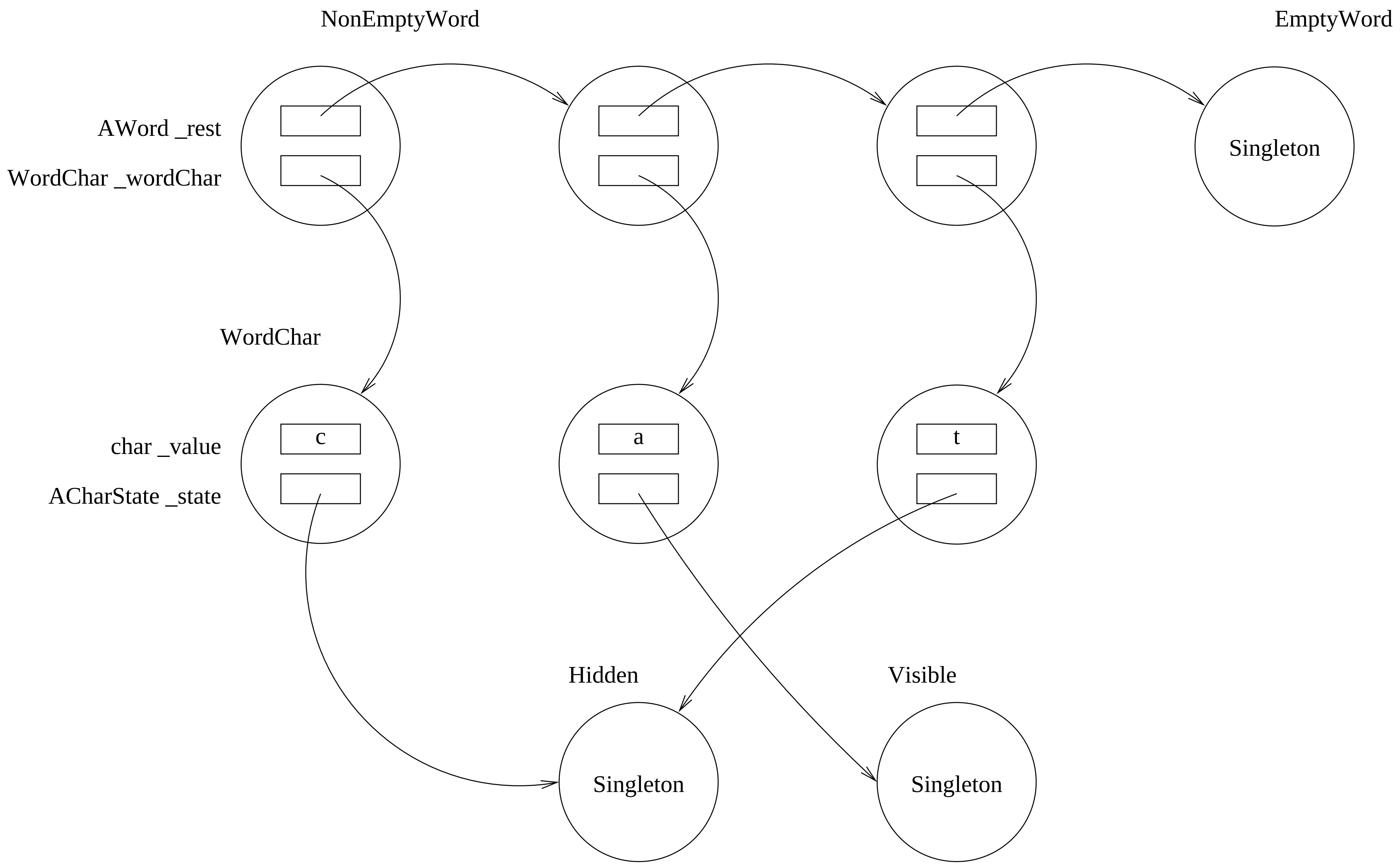
Program #1: Hangman (cont.)

- We can apply the state pattern here to implement hangman characters as objects with states. The pattern calls for the following design steps:

  1. Define class `WordChar` to represent the characters in a hangman word.
  2. Define abstract class `ACharState`.
  3. Define classes `Hidden` and `Visible` as concrete subclasses of `ACharState`.
     - ACharState and its concrete variants represent the states of a `WordChar`.
  4. Define a field in `WordChar` to reference an `ACharState`, its current state.
     - All method calls in `WordChar` are delegated to its state.

Program #1: Hangman (cont.)

Secret Word: cat a is visible c and t are hidden



Program #1: Hangman (cont.)

- The UML diagram on the handout illustrates the above design.
  - This design makes use of the composite pattern, the state pattern, and the singleton pattern.