The State Pattern

Graeme Geldenhuys

2009-03-20

This month we will look at the State Pattern - one of the lesser know design patterns in the GoF (Gang of Four) Design Patterns book. We use the State pattern when an object's behaviour changes at run-time, due to its internal state. I will highlight a few examples and work through a detailed example of parsing a CSV file by creating an Infinite State Machine using the State Pattern.

Introduction

In this article I am going to create a CSV parser. If you don't know, CSV is an acronym for Comma Separated Value and is a text file with the following format.

```
FieldOne,"Field two, which can have commas.",12345,Field4
"Field1","Field two in record two.",98765,"Field4Again"
```

Each field is separated by a comma. Text fields containing a single word can be enclosed in quotes, but this is not mandatory thought it is recommended so CSV parsers can clearly distinguish between text fields and other type of fields. Number fields are not enclosed in quotes. Every line in the text file represents a single record of data.

So the big question is when do we know we can use the State Pattern. The answer is simple - when you see code that has large conditional statements like nested *if..else blocks* or large *case statements* in your code, then you have an ideal candidate for the State Pattern.

The definition of the State Pattern as defined in the Design Patterns[1] book is as follows.

> *"Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class." — Design Patterns Book*

This is a slight exaggeration as most programming language don't support classes chan-

1 Eric Gamma, Richard Helm, Ralph Johnson & John Vlissides: Design Patterns: Elements of Reusable Object- Oriented Software. Addison-Wesley 1994. ISBN 0-201-63361-2, Page 305.

ging their actual class types at run-time, but for the Object Pascal language we can easily mimic that behaviour as I will show later.

To help understand the State Pattern a bit better I will refer to the UML diagram of our CSV Parser shown in Figure 1.

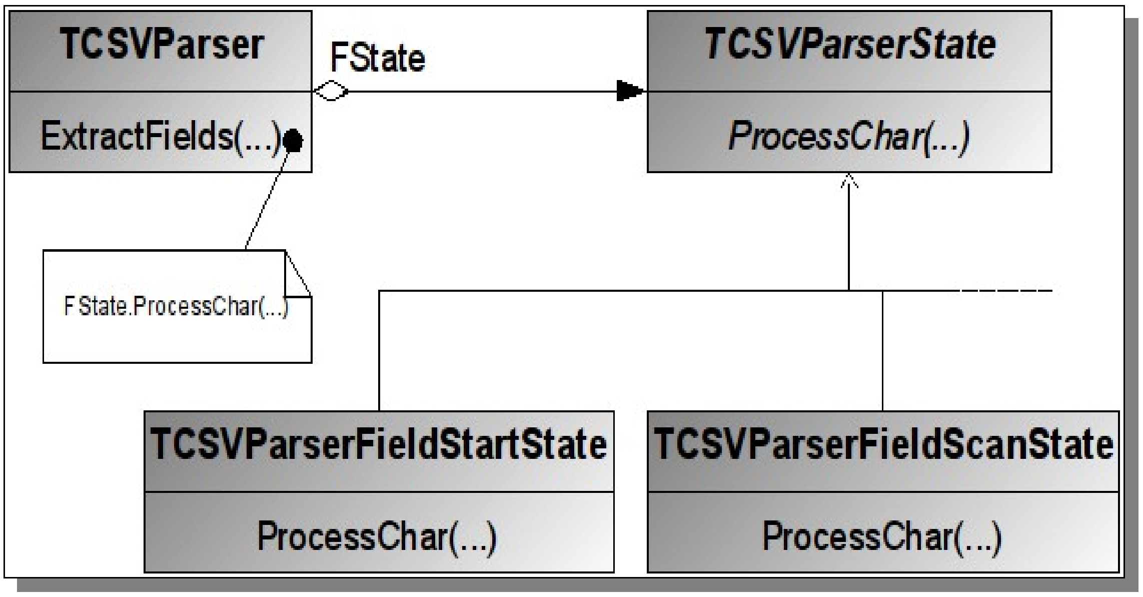

Figure 1: UML class diagram showing our CSV Parser.

To summarise what the UML class diagram tells us: The `TCSVParser` is our primary class in our CSV Parser project and is visible to its clients (other classes using this class to parse a CSV file). The `TCSVParserState` is an abstract class which declares a common interface which the concrete subclasses will implement. The abstract class and its concrete subclasses are hidden from the clients allowing us to change them at will. The concrete subclasses represent specific actions for specific states that the CSV Parser can be in. The `TCSVParser` keeps track of the current state by storing the concrete instance of that state in the `FState` field variable.

As you can see from the UML diagram, every state is represented by a subclass. This is not require, but does explain the State Pattern more clearly. Some developers will take a few shortcuts here and rather implement the State as an enumerated type. You will also have noticed that each state subclass must implement the abstract `ProcessChar(...)` method.

I would also like to point out that primary class (TCSVParser in our example) may pass itself as a parameter to the State object handling the current task. This lets the State object query the primary class for information in needed.

Another important point to make is that the primary class or any of the state subclasses can decide which state succeeds the current state and under what circumstances. This is also why it is handy to be able to reference the primary class from the state subclasses.

Implementation

As I mentioned earlier, we are going to implement a Finite State Machine using the State Pattern. Figure 2 shows a visual representation of our CSV Parser that we are going to create and under what circumstances we change states. Each state will be represented by a class and in our implementation each state class will decided the next state based on the current information being processed.

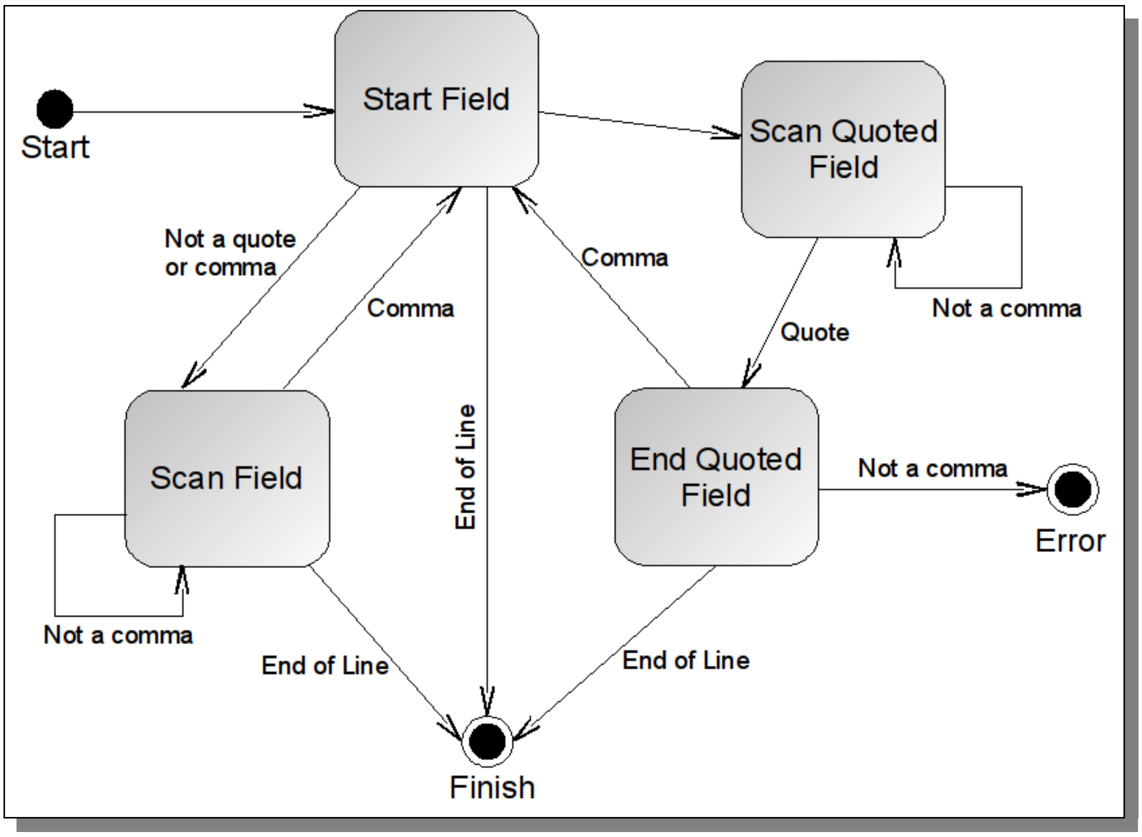

Figure 2: Finite State Machine for our CSV Parser.

Without any further delay it is time to look at some code. I am first going to work through the main CSV Parser class called `TCSVParser`, then I'll work through the state classes.

```
TCSVParser = class(TObject)
private
  FCurrentLine: string;
  FState: TCSVParserState;
  { cached state classes for performance }
  FFieldStartState: TCSVParserFieldStartState;
  FScanFieldState: TCSVParserScanFieldState;
  FScanQuotedState: TCSVParserScanQuotedState;
  FEndQuotedState: TCSVParserEndQuotedState;
  FGotErrorState: TCSVParserGotErrorState;
  { Fields used during parsing }
  FCurrentField: string;
  FFieldList: TStrings;
  function GetState: TParserStateClass;
  procedure SetState(const Value: TParserStateClass);
protected
  procedure AddCharToCurrentField(Ch: char);
  procedure AddCurrentFieldToList;
  property State: TParserStateClass
      read GetState write SetState;
public
  constructor Create;
  destructor Destroy; override;
  procedure ExtractFields(const S: string;
      var pFieldList: TStrings);
  property CurrentLine: string read FCurrentLine;
end;
```

The first thing you will notice is that we have cached state objects in private fields. This is simply done for performance reasons. When parsing large CSV files, the state will change frequently. So using cached state objects will give us much better performance compared to creating and destroying state objects over and over. For smaller files or when we use the State pattern where the internal state does not change that frequent we would rather opt to create and destroy the state objects as we need them.

We also have a protected `State` property which keeps track of which state the parser is currently in. This simply points to one of the state object instances. The state subclasses will also use this property to change the internal state of the parser. More on how or when they do this a little bit later. The two private fields `FFieldList` and `FCurrent-Field` are used to store the fields we have already processed or storing the characters of the field we are currently processing.

Then the last important method is the `ExtractFields` method which you call to start the whole parsing process. Below is a snippet of that method and shows how we iterate through all the characters of the string we are processing. We that pass each character to the state object currently stored in the `FState` field so it can be processed.

```
{ Read through all the characters in the string }
for i := 1 to Length(s) do
begin
  { Get the next character }
  Ch := s[i];
  FState.ProcessChar(Ch, i);
end;
```

Now that we have looked at the main parser object, it is time we take a look at the state objects. To make things easier and reuse as much code as possible, we define an abstract state class which does any common tasks for us.

```
{ Abstract State object }
TCSVParserState = class(TObject)
private
  FParser: TCSVParser;
  procedure ChangeState(NewState: TParserStateClass);
  procedure AddCharToCurrentField(Ch: char);
  procedure AddCurrentFieldToList;
public
  constructor Create(AParser: TCSVParser);
  { Must be implemented in the concrete sub-classes to
    handle the input character and decide on the next
    state. }
  procedure ProcessChar(Ch: AnsiChar; Pos: integer);
      virtual; abstract;
end;
```

If you look at the constructor you can see we pass in as a parameter the main CSV Parser instance. This is not required, but is quite common so that any of the State objects can reference back to the parser if they need to know in what context they are being used, or if they want to change the internal state of the CSV Parser.

For the state subclasses to actually change the state of the CSV Parser they call the `ChangeState(NewState)` method. The parameter for `ChangeState` is of type `TParserStateClass` and is defined as follows.

```
TParserStateClass = class of TCSVParserState;
```

This allows the state subclasses to pass in the new state class type instead of the new state object instance. This simply minimises dependencies between state subclasses. The `ProcessChar(...)` method is declared as abstract and needs to be implemented by the concrete subclasses to define how they will process each character.

The Interface section for the rest of the state objects look similar to the following:

```
TCSVParserFieldStartState = class(TCSVParserState)
public
  procedure ProcessChar(Ch: AnsiChar; Pos: integer);
      override;
end;
```

They simply override the `ProcessChar(...)` method and implement their specific behaviour. In Table 1 I listed all the concrete state classes that we will need for the CSV Parser and what the function is of each one.

Concrete State Class	**Description**
TCSVParserFieldStartState	A concrete state object used when starting a new field
TCSVParserScanFieldState	A concrete state object used while scanning a field
TCSVParserScanQuotedState	A concrete state object used while scanning double quoted fields
TCSVParserEndQuotedState	A concrete state object used when we found the ending double quote
TCSVParserGotErrorState	A concrete state object used when some error occurred like an invalid CSV structure

Table 1: The function of each concrete state class.

Below is the implementations of the five state classes. As you can see the implementations is pretty straight forward. Each class simply looks at the character being processed and decides what action to take and if required what state to change to next. `TCSV-ParserGotErrorState` is the only class that does something different, because it's a slightly special case. If that state is reached, it means that an error occurred in the parsing of the CSV and we simply raise an exception to notify the users of the issue. Obviously this doesn't mean the CSV parsing needs to stop for the whole file. The user can maybe decide to log all errors so can can manually be fixed up and the reparse them at a later date.

```
procedure TCSVParserFieldStartState.ProcessChar(Ch: AnsiChar;
    Pos: integer);
begin
  if (Ch = '"') then
     ChangeState(TCSVParserScanQuotedState)
  else if (Ch = ',') then
    AddCurrentFieldToList
  else
  begin
    AddCharToCurrentField(Ch);
    ChangeState(TCSVParserScanFieldState);
  end;
end;

procedure TCSVParserScanFieldState.ProcessChar(Ch: AnsiChar;
    Pos: integer);
begin
  if (Ch = ',') then
  begin
    AddCurrentFieldToList;
    ChangeState(TCSVParserFieldStartState);
  end
  else
    AddCharToCurrentField(Ch);
end;

procedure TCSVParserScanQuotedState.ProcessChar(Ch: AnsiChar;
    Pos: integer);
begin
  if (Ch = '"') then
    ChangeState(TCSVParserEndQuotedState)
  else
    AddCharToCurrentField(Ch);
end;

procedure TCSVParserEndQuotedState.ProcessChar(Ch: AnsiChar;
    Pos: integer);
begin
  if (Ch = ',') then
  begin
    AddCurrentFieldToList;
    ChangeState(TCSVParserFieldStartState);
  end
  else
    ChangeState(TCSVParserGotErrorState);
end;

procedure TCSVParserGotErrorState.ProcessChar(Ch: AnsiChar;
    Pos: integer);
const
  err = 'Error in line at position %d: ' + #10 + '<%s>';
begin
  raise Exception.Create(
      Format(err, [Pos, FParser.CurrentLine]));
end;
```

If you take a closer look at the implementations of `TCSVParserFieldStartState` and `TCSVParserScanQuotedState` you will see exactly what is meant by implementing specific behaviour depending on the state. In both these classes they handle the " (quote) character, but in very different ways. In `TCSVParserFieldStartState` the is interpreted as the beginning of a field and the parser's state is changed accordingly. In `TCSVParserScanQuotedState` the " (quote) character signifies the end of the field and this time the parser's state is changed to a different state as before. The same characte is being processed, but the state of the parser afterwards is very different.

Conclusion

That concludes our implementation of the CSV Parser. What remains now is getting an instance of the parser. For that we go back to the *Singleton*[2] design pattern which has already been covered in a previous article.

We have a global method returning an instance of the CSV Parser. If the instance did not already exist, it creates one on the fly. The CSV Parser instance is stored in a unit wide variable `uCSVParser` and is freed in the finalization section when the application is terminated.

```
function gCSVParser: TCSVParser;
begin
  if uCSVParser = nil then
    uCSVParser := TCSVParser.Create;
  Result := uCSVParser;
end;
```

The *State* pattern is very flexible and can be used for parsing other file formats like XML as well or in source code compilers. The *Design Patterns* book gives a totally different example where they talk about designing a TCPConnection component and the different states of the connections (Established, Listen, Closed, etc.) is being represented as state subclasses.

I hope you can find many more uses for the *State* pattern now that you have yet another design pattern in your programming toolbox.

2 The Iterator Pattern - Toolbox 1'2009. Page 11.