CMPE 152: Compiler Design

August 29 Lab

Department of Computer Engineering
San Jose State University

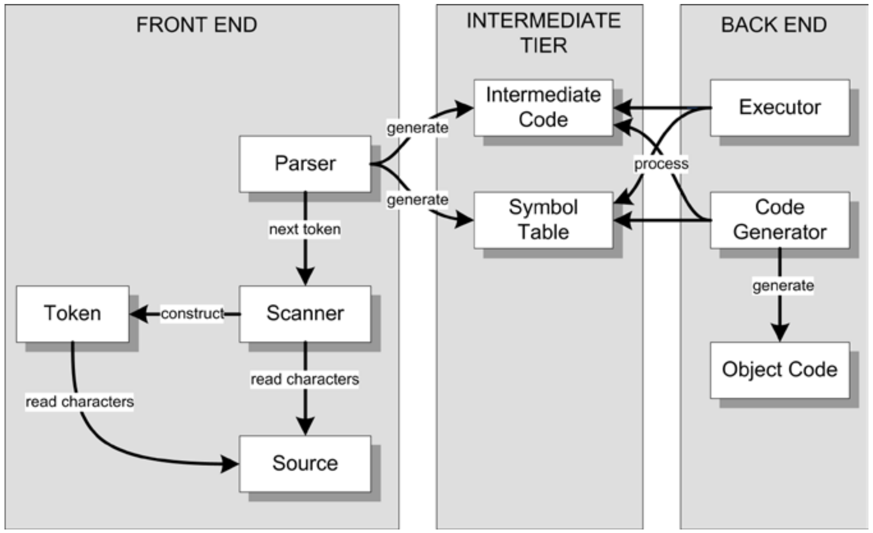
Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

SILICON VALLEY'S
FIRST CHOICE
FOR NEW
ENGINEERING HIRES
— Silicon Valley Business Journal, 2013

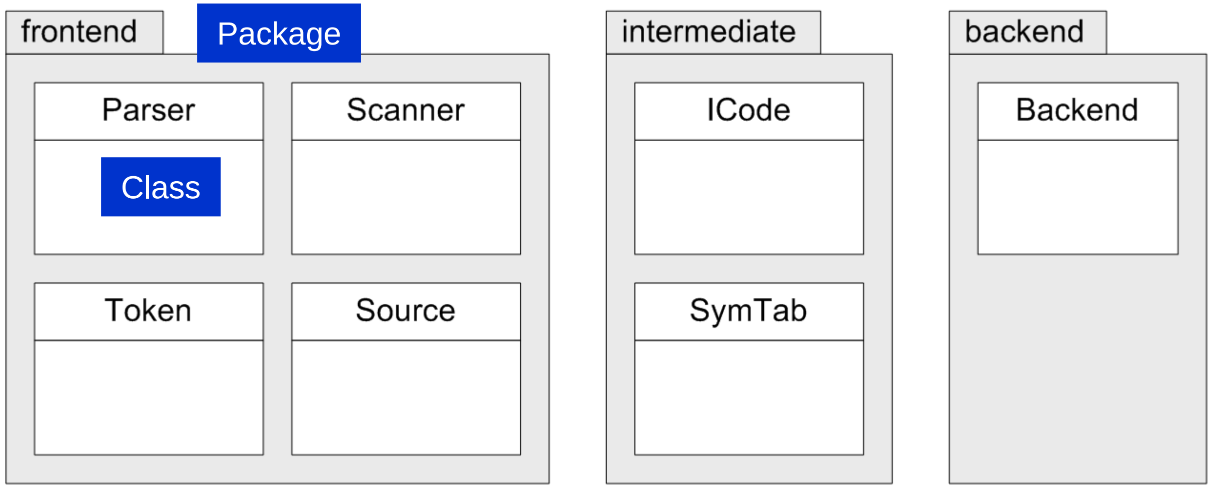
San José State
UNIVERSITY

Three C++ Namespaces

FROM:

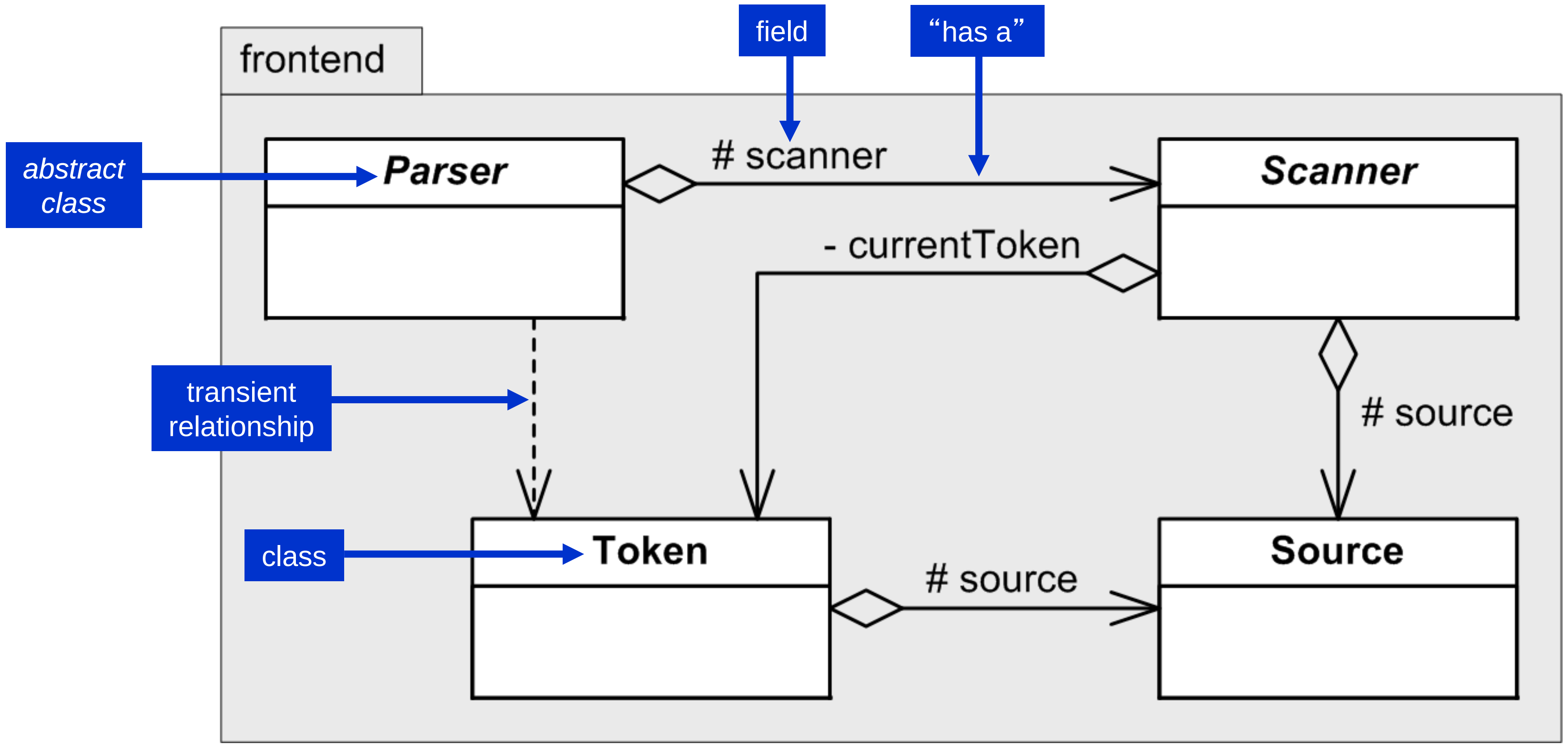


TO:



UML package and class diagrams.

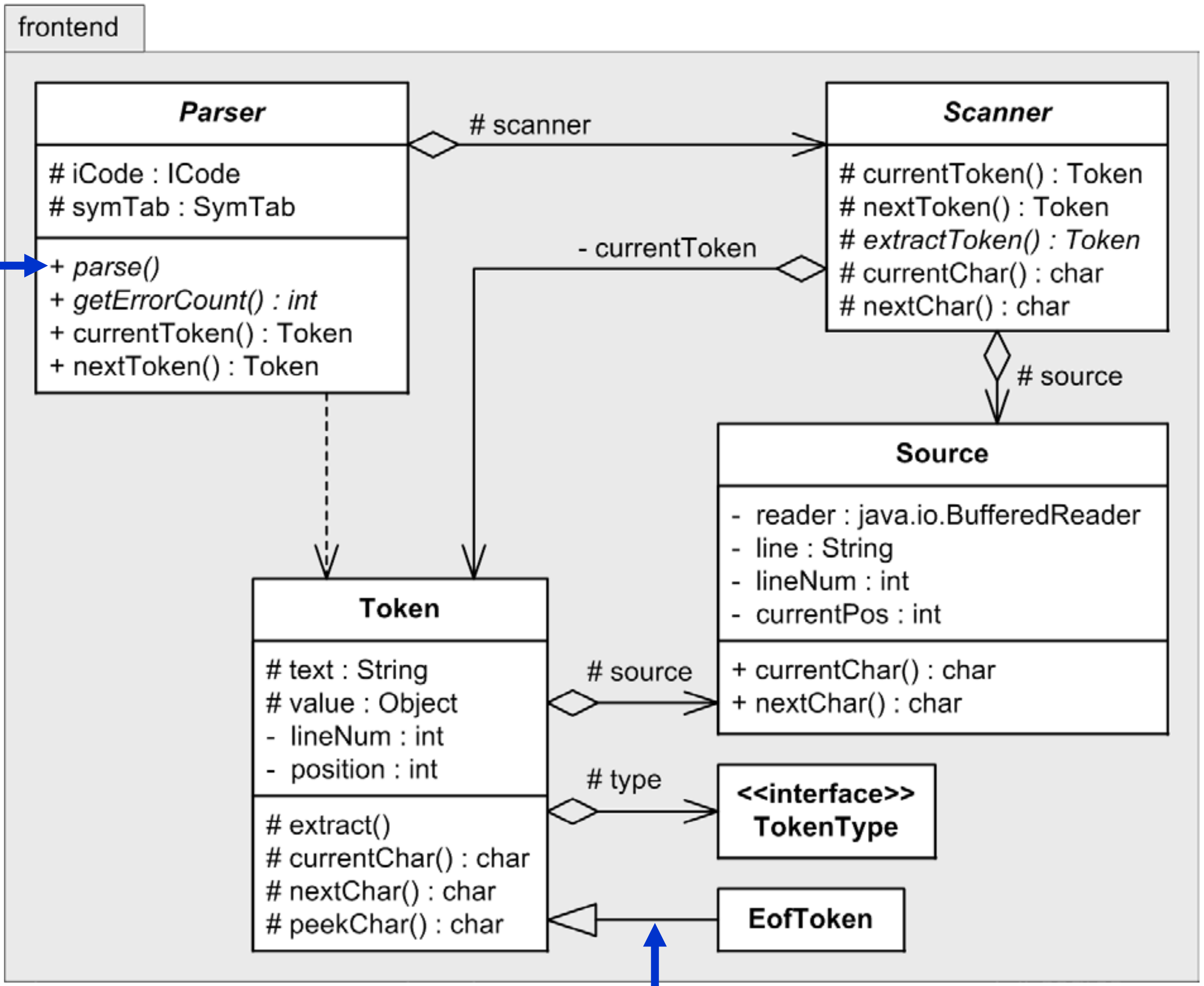
Front End Class Relationships



These four **framework classes** should be **source language-independent.**

+	public
-	private
#	protected
~	package

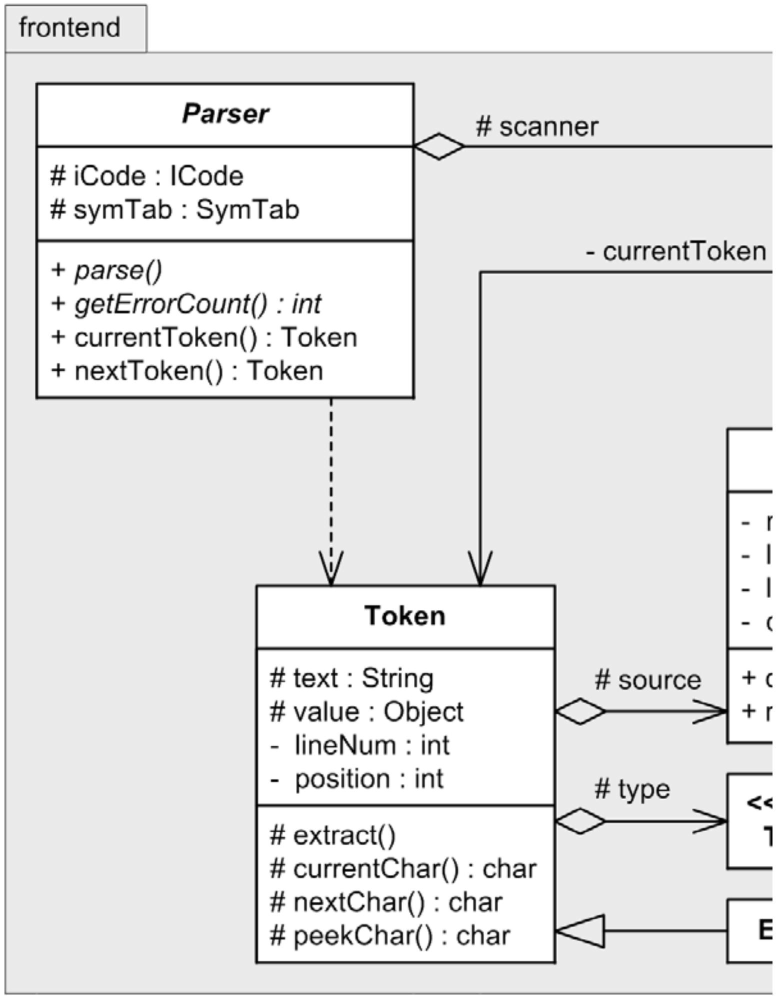
Front End Fields and Methods



abstract method

"subclass of" or "is a"

The Abstract Parser Class



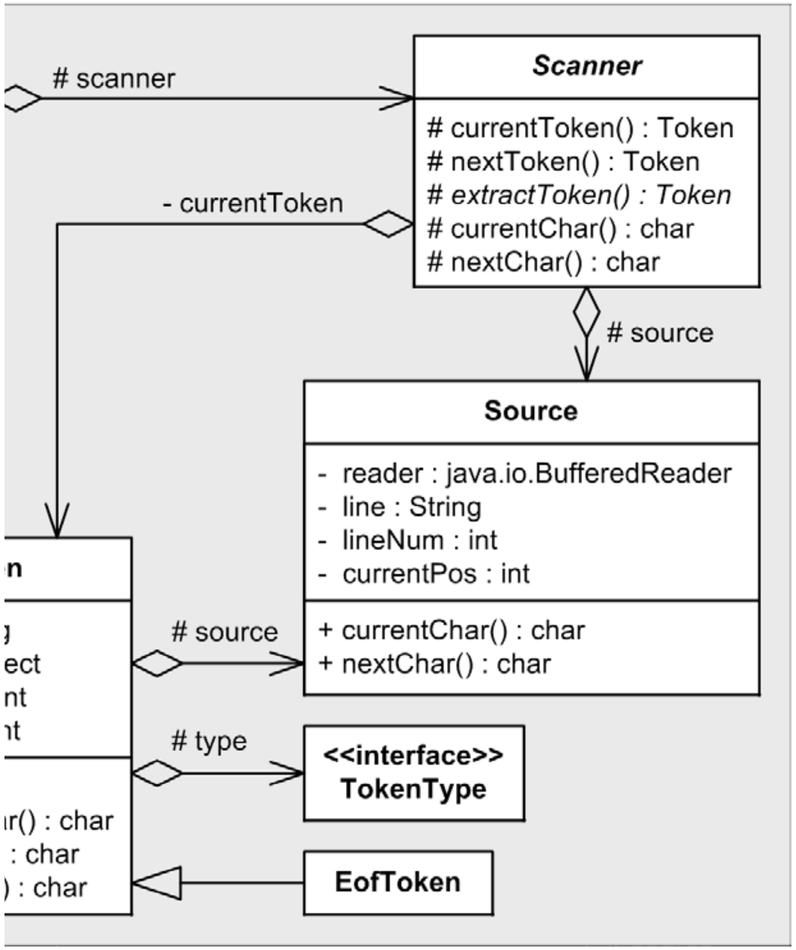
- Fields **icode** and **symtab** refer to the intermediate code and the symbol table.
- Field **scanner** refers to the scanner.
- Abstract **parse()** and **get_error_count()** methods.
  - To be implemented by language-specific parser subclasses.
- "Convenience methods" **current_token()** and **next_token()** simply call the **current_token()** and **next_token()** methods of **Scanner**.

The Abstract Scanner Class

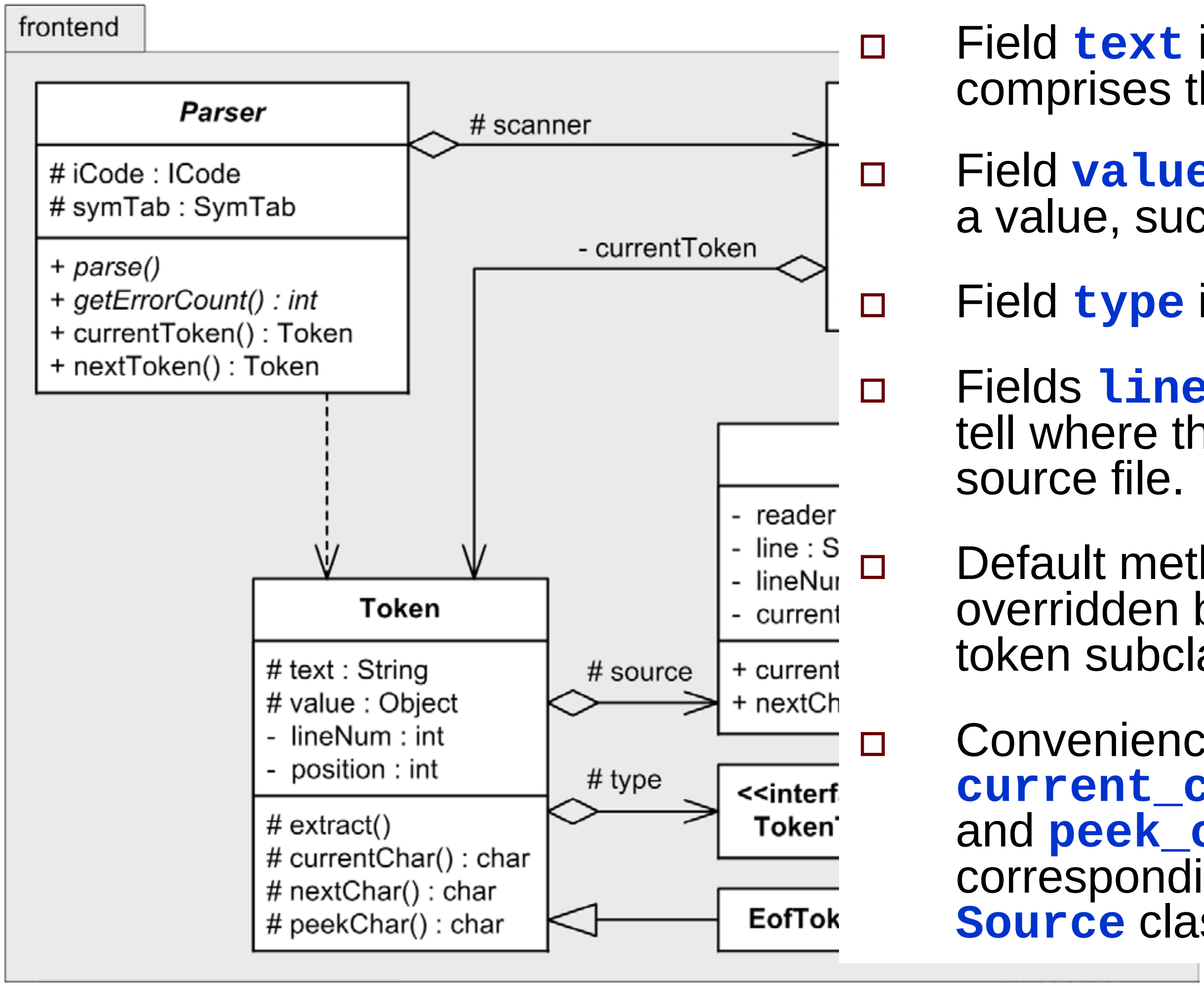
- Private field `current_token` refers to the current token, which protected method `current_token()` returns.

- Method `next_token()` calls abstract method `extract_token()`.
  - To be implemented by language-specific scanner subclasses.

- Convenience methods `current_char()` and `next_char()` call the corresponding methods of `Source`.

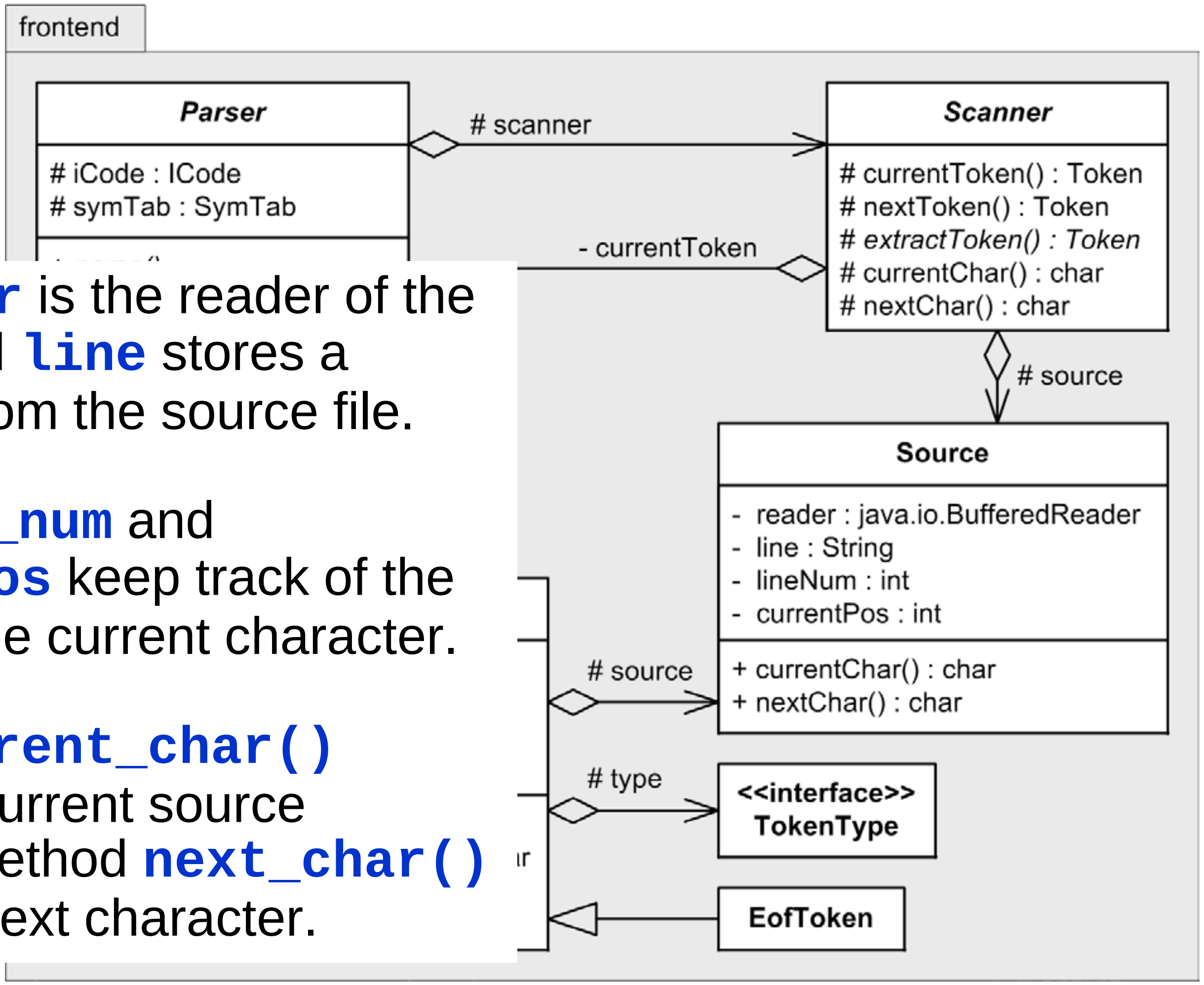


The Token Class



- Field **text** is the string that comprises the token.
- Field **value** is for tokens that have a value, such as a number.
- Field **type** is the token type.
- Fields **line_num** and **position** tell where the token is in the source file.
- Default method **extract()** will be overridden by language-specific token subclasses.
- Convenience methods **current_char()**, **next_char()**, and **peek_char()** call the corresponding methods of the **Source** class.

The Source Class



- Field **reader** is the reader of the source. Field **line** stores a single line from the source file.
- Fields **line_num** and **current_pos** keep track of the position of the current character.
- Method **current_char()** returns the current source character. Method **next_char()** returns the next character.

Current Character vs. Next Character

- Suppose the source line contains **ABCDE** and we've already read the first character.

`current_char()`	**A**
`next_char()`	**B**
`next_char()`	**C**
`next_char()`	**D**
`current_char()`	**D**
`current_char()`	**D**
`next_char()`	**E**
`next_char()`	***eol***

Messages from the Front End

- The **`Parser`** generates messages.
  - Syntax error messages
  - Parser summary
    - number of source lines parsed
    - number of syntax errors
    - total parsing time
- The **`Source`** generates messages.
  - For each source line:
    - line number
    - contents of the line

Front End Messages, *cont'd*

- We want the message producers (`Parser` and `Source`) to be loosely-coupled from the message listeners.

- The producers shouldn't care who listens to their messages.
- The producers shouldn't care what the listeners do with the messages.

- The listeners should have the flexibility to do whatever they want with the messages.

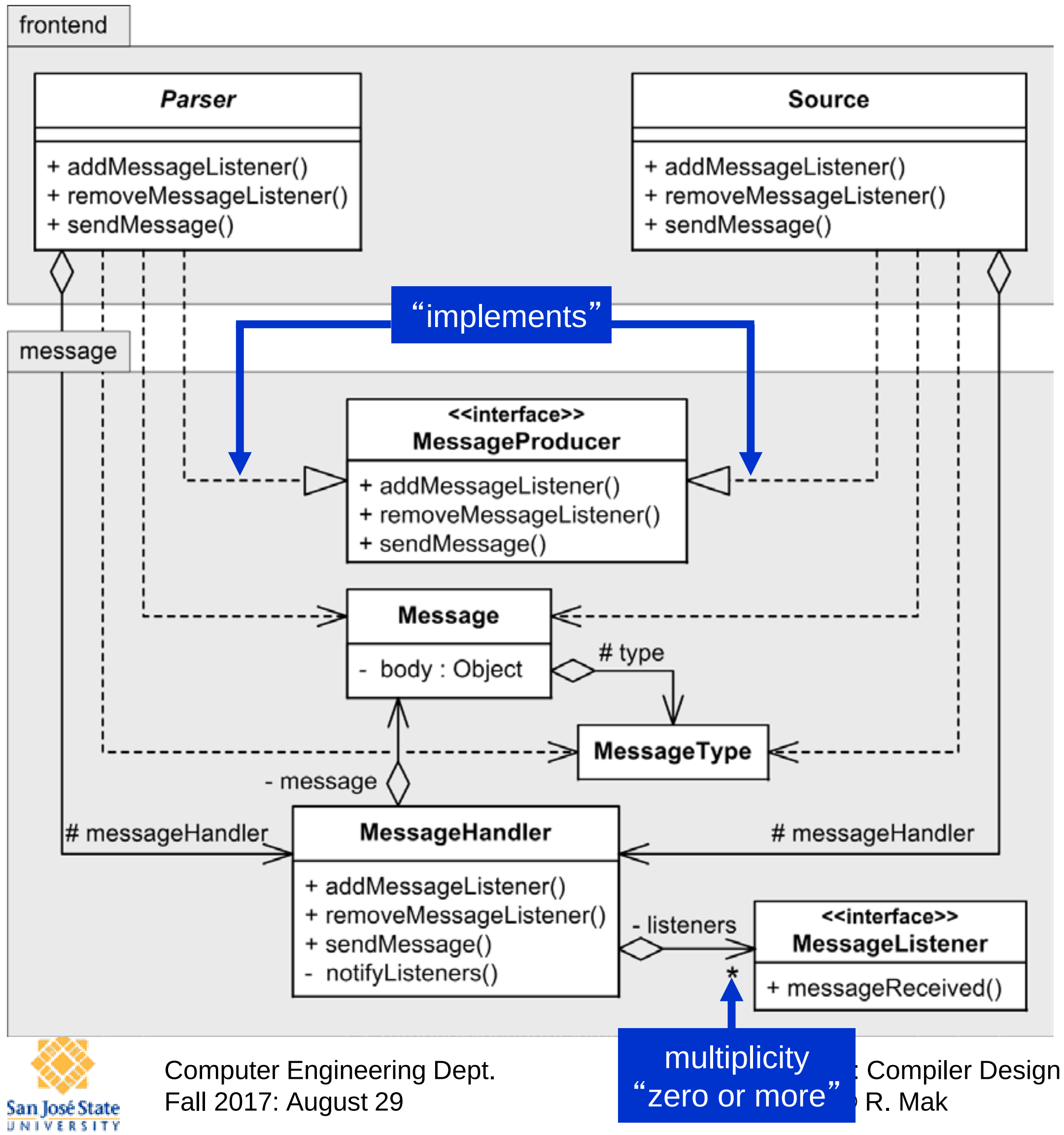
Front End Messages, *cont'd*

- Producers implement the `MessageProducer` interface.
- Listeners implement the `MessageListener` interface.
- A listener registers its interest in the messages from a producer.
- Whenever a producer generates a message, it "sends" the message to all of its registered listeners.

Front End Messages, *cont'd*

- A message producer can delegate message handling to a `MessageHandler`.
- This is the Observer Design Pattern.

Message Implementation



- Message producers implement the **`MessageProducer`** interface.
- Message listeners implement the **`MessageListener`** interface.
- A message producer can delegate message handling to a **`MessageHandler`**.
- Each **`Message`** has a message **`type`** and a **`body`**.

This appears to be a lot of extra work, but it will be easy to use and it will pay back large dividends.

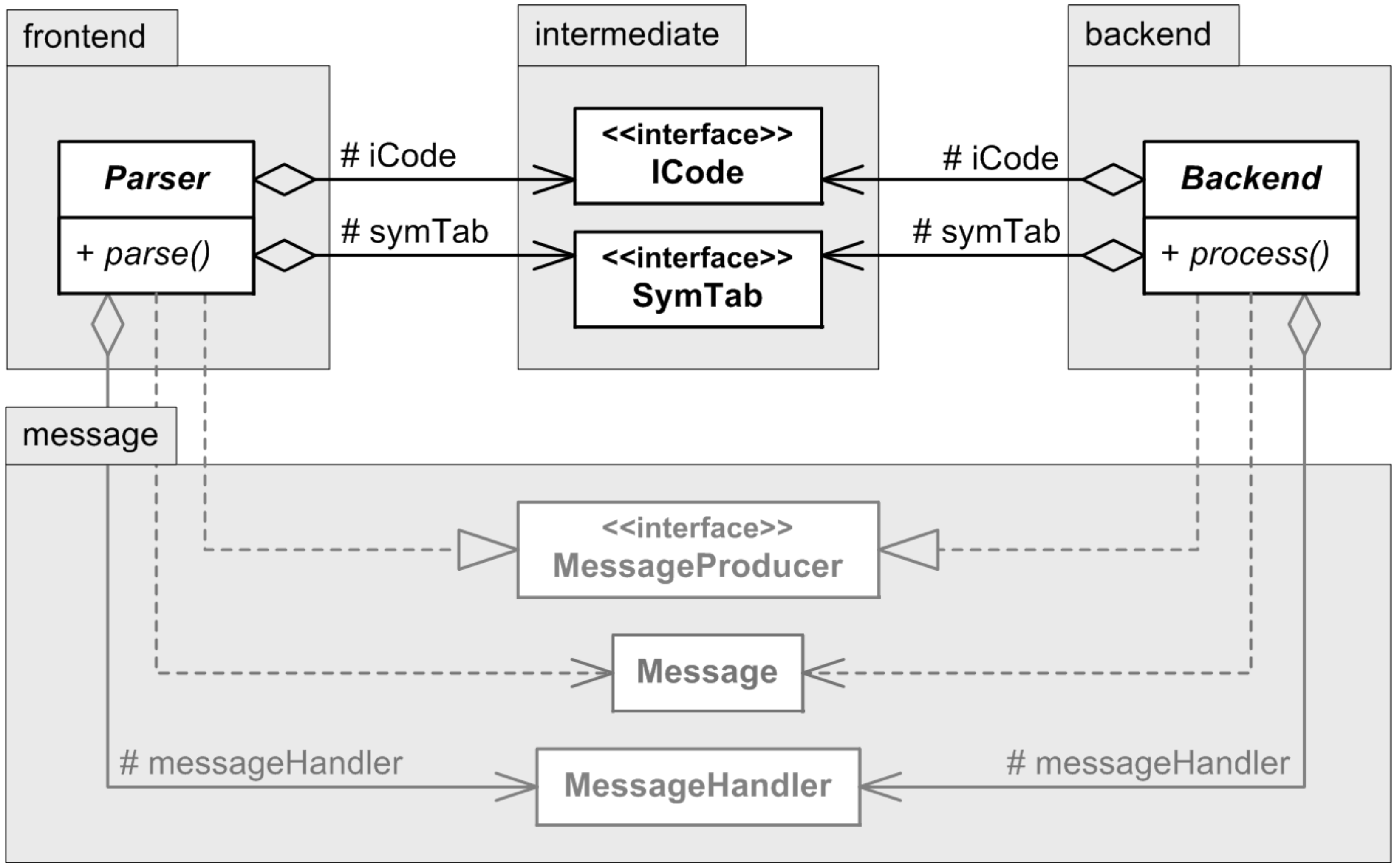
Two Message Types

- `SOURCE_LINE` message
  - the source line number
  - text of the source line

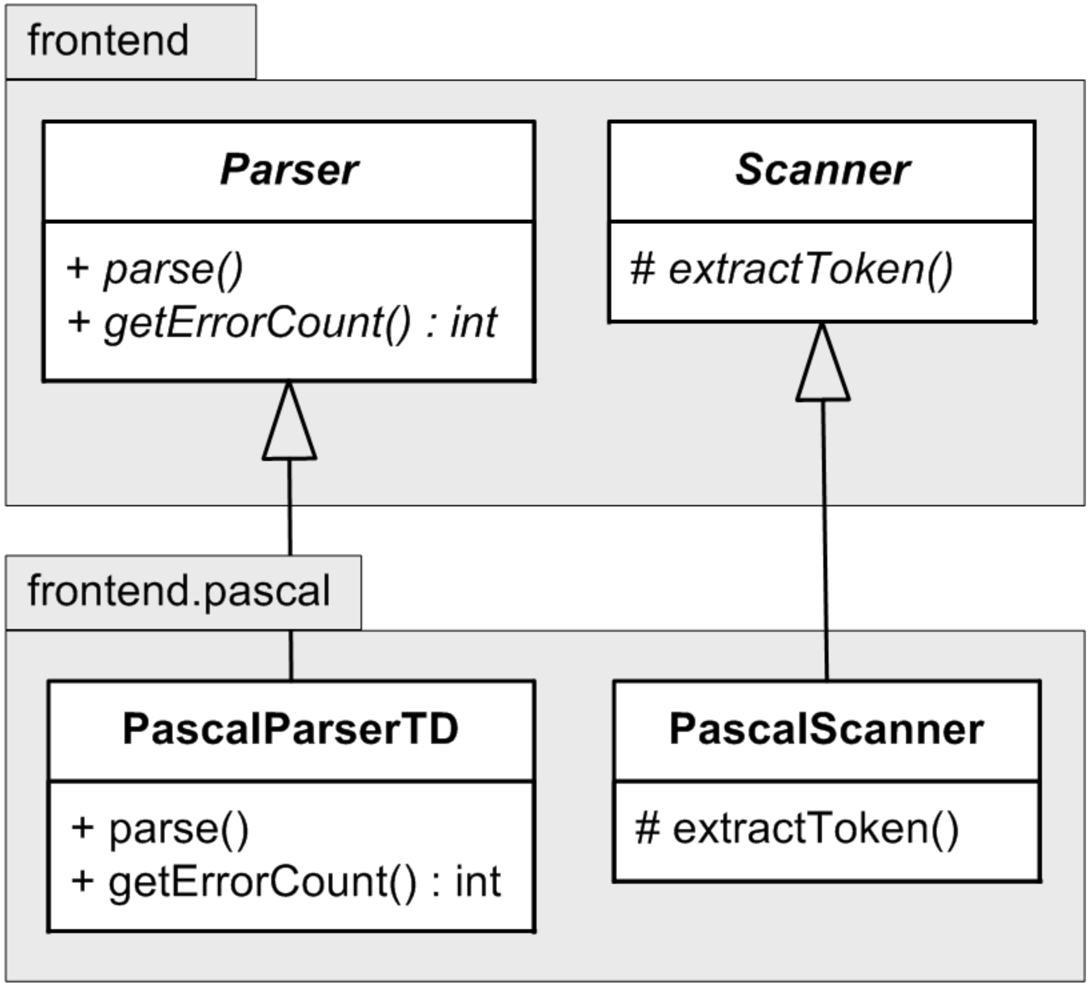
- `PARSER_SUMMARY` message
  - number of source lines read
  - number of syntax errors
  - total parsing time

By convention, the message producers and the message listeners agree on the format and content of the messages.

Good Framework Symmetry



Pascal-Specific Front End Classes



- **`PascalParserTD`** is a subclass of **`Parser`** and implements the **`parse()`** and **`get_error_count()`** methods for Pascal.
  - TD for "top down"
- **`PascalScanner`** is a subclass of **`Scanner`** and implements the **`extract_token()`** method for Pascal.

Strategy
Design Pattern

The Pascal Parser Class

- The initial version of method `parse()` does hardly anything, but it forces the scanner into action and serves our purpose of doing end-to-end testing.

The Pascal Parser Class, *cont'd*

```
void PascalParserTD::parse() throw (string)
{
    steady_clock::time_point start_time = steady_clock::now();

    int last_line_number;
    Token *token = nullptr;

    // Loop over each token until the end of file.
    while ((token = next_token(token)) != nullptr)
    {
        last_line_number = token->get_line_number();
    }

    // Send the parser summary message.
    steady_clock::time_point end_time = steady_clock::now();
    double elapsed_time =
            duration_cast<duration<double>>(end_time - start_time).count();
    Message message(PARSER_SUMMARY,
                    LINE_COUNT, to_string(last_line_number),
                    ERROR_COUNT, to_string(get_error_count()),
                    ELAPSED_TIME, to_string(elapsed_time));
    send_message(message);
}
```

What does this `while` loop do?

The Pascal Scanner Class

- The initial version of method **extractToken()** doesn't do much either, other than create and return either a default token or the EOF token.

```
Token *PascalScanner::extract_token() throw (string)
{
    Token *token;
    char current_ch = current_char();

    // Construct the next token.  The current character determines the
    // token type.
    if (current_ch == Source::END_OF_FILE)
    {
        token = nullptr;
    }
    else
    {
        token = new Token(source);
    }

    return token;
}
```

Remember that the **Scanner** method **next_token()** calls the abstract method **extract_token()**.

Here, the **Scanner** subclass **PascalScanner** implements method **extract_token()**.

The Token Class

- The **`Token`** class's default **`extract()`** method extracts just one character from the source.
  - This method will be overridden by the various token subclasses.
  - It serves our purpose of doing end-to-end testing.

```
void Token::extract() throw (string)
{
    text = to_string(current_char());
    next_char();  // consume current character
}
```

The Token Class, *cont'd*

- A character (or a token) is "consumed" after it has been read and processed, and the next one is about to be read.
- If you forget to consume, you will loop forever on the same character or token.

A Front End Factory Class

- A language-specific parser goes together with a scanner for the same language.
- But we don't want the framework classes to be tied to a specific language. Framework classes should be language-independent.
- We use a factory class to create a matching parser-scanner pair.

Factory Method Design Pattern

A Front End Factory Class, *cont'd*

- **Good:**

“Coding to the interface.”

```
Parser parser =
    FrontendFactory::create_parser( … );
```

  - Arguments to the `create_parser()` method enable it to create and return a parser bound to an appropriate scanner.
  - Variable `parser` doesn't have to know what kind of parser subclass the factory created.
  - Once again, the idea is to maintain loose coupling.

A Front End Factory Class, *cont'd*

- **Good:**

```
Parser *parser =
    FrontendFactory::create_parser( … );
```

- **Bad:**

```
PascalParserTD *parser =
    new PascalParserTD( … )
```

  - Why is this bad?
  - Now variable `parser` is tied to a specific language.

A Front End Factory Class, *cont'd*

```
Parser *FrontendFactory::create_parser(string language, string type,
                                       Source *source)
    throw (string)
{
    if ((language == "Pascal") && (type == "top-down"))
    {
        Scanner *scanner = new PascalScanner(source);
        return new PascalParserTD(scanner);
    }
    else if (language != "Pascal") {
        throw new string("Parser factory: Invalid language '" +
                         language + "'");
    }
    else {
        throw new string("Parser factory: Invalid type '" +
                         type + "'");
    }
}
```

Pascal Programming Workshop

- Install Free Pascal: https://www.freepascal.org