CS 153: Concepts of Compiler Design

August 29 Class Meeting

Department of Computer Science
San Jose State University

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

Basic Info

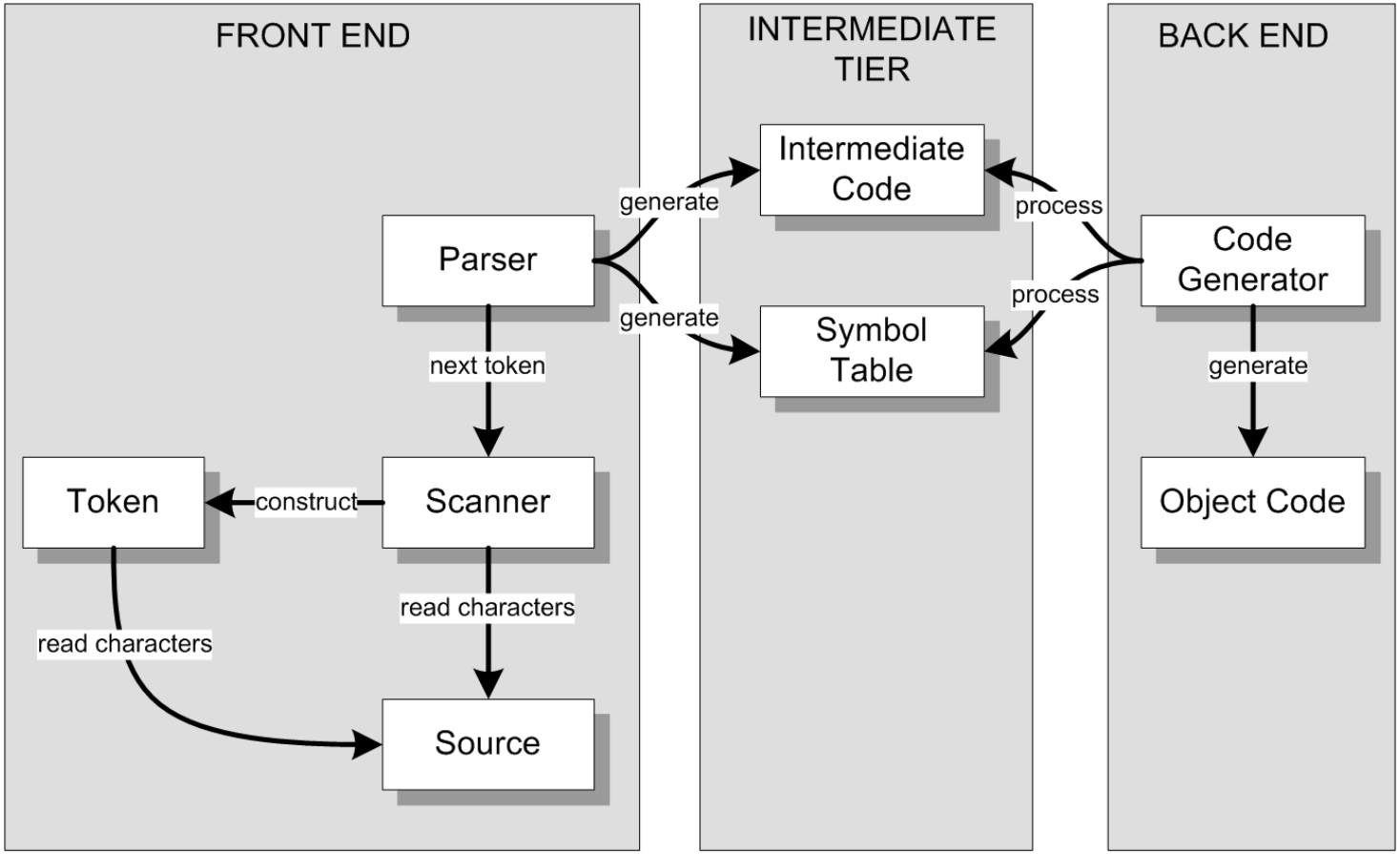
- Office hours
  - TuTh 3:00 – 4:00 PM
  - ENG 250
- Website
  - Faculty webpage: http://www.cs.sjsu.edu/~mak/
  - Class webpage: http://www.cs.sjsu.edu/~mak/CS153/index.html
  - Syllabus
  - Assignments
  - Lecture notes

Permission Codes?

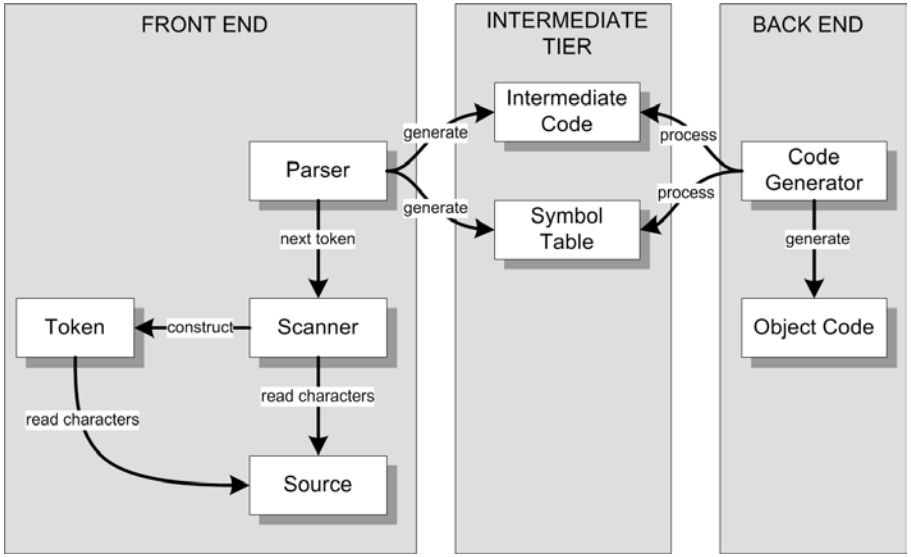
- If you need a permission code to enroll in this class, please fill out and hand in a filled-out and signed “Add Code Information” form.
  - Be sure to list prerequisite courses that you’ve successfully completed.
  - Prerequisites: CS 47 or CMPE 102, CS 146, and CS 154 (with a grade of "C-" or better in each); Computer Science, Applied and Computational Math, or Software Engineering majors only.
- Priority will be given to graduating seniors.
  - You must show your graduating senior card.

Conceptual Design (Version 2)

- We can architect a compiler with three major parts:



Major Parts of a Compiler



- Front end
  - Parser, Scanner, Source, Token
- Intermediate tier
  - Intermediate code (icode)
    - "Predigested" form of the source code that the back end can process efficiently.
    - Example: parse trees
    - AKA intermediate representation (IR)
  - Symbol table (symtab)
    - Stores information about the symbols (such as the identifiers) contained in the source program.
- Back end
  - Code generator
    - Processes the icode and the symtab in order to generate the object code.

Only the front end needs to be source language-specific.

The intermediate tier and the back end can be language-independent!

What Else Can Compilers Do?

- Compilers allow you to program in a high-level language and think about your algorithms, not about machine architecture.
- Compilers provide language portability.
  - You can run your C++ and Java programs on different machines because their compilers enforce language standards.

What Else Can Compilers Do? *cont'd*

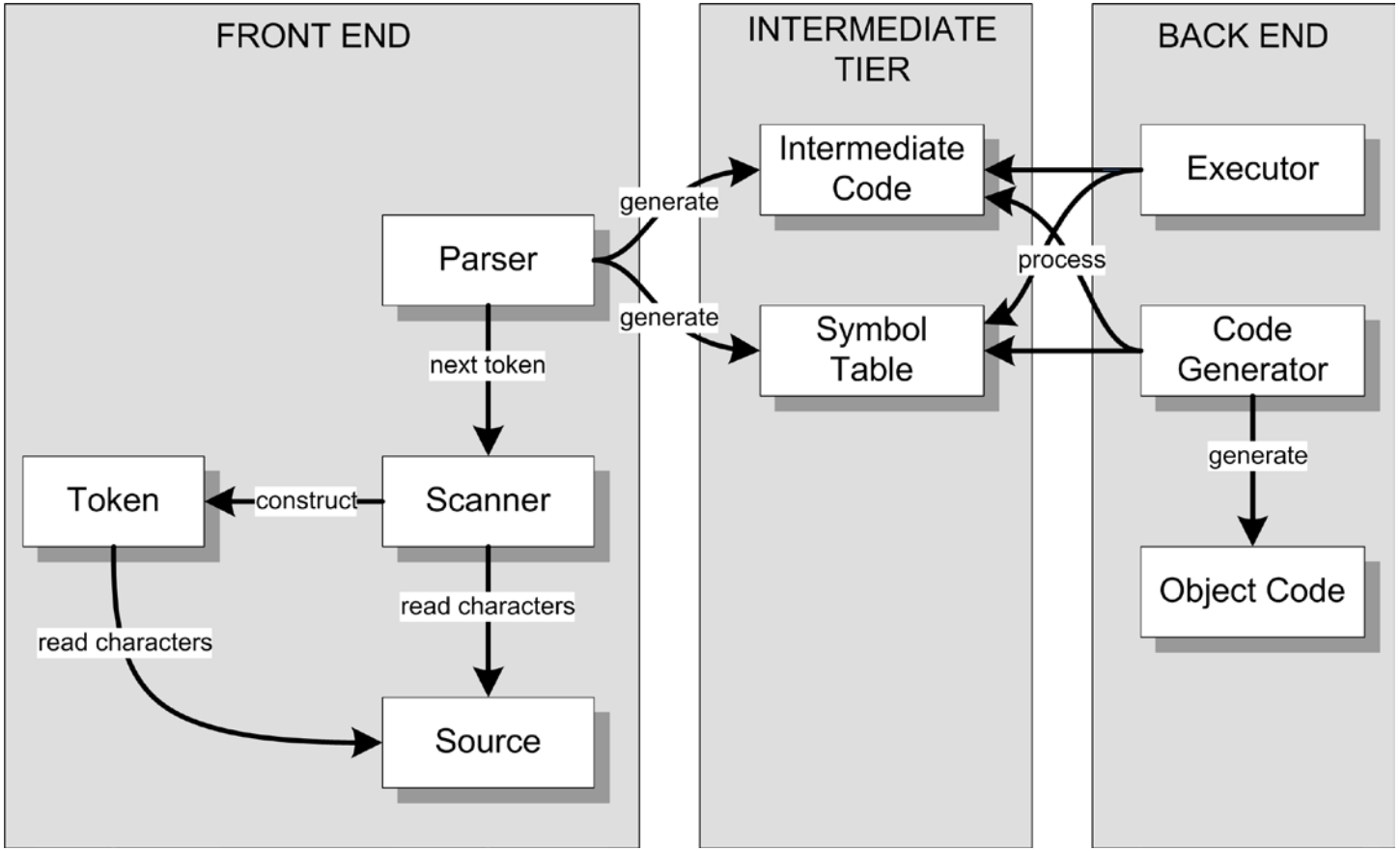
- Compilers can optimize and improve the execution of your programs.
  - Optimize the object code for speed.
  - Optimize the object code for size.
  - Optimize the object code for power consumption.

What about Interpreters?

- An interpreter executes a source program instead of generating object code.
- It executes a source program using the intermediate code and the symbol table.

Demo

Conceptual Design (Version 3)



- A compiler and an interpreter can both use the same front end and intermediate tier.

Comparing Compilers and Interpreters

- A compiler generates object code, but an interpreter does not.
- Executing the source program from object code can be several orders of magnitude faster than executing the program by interpreting the intermediate code and the symbol table.
- But an interpreter requires less effort to get a source program to execute
  - ➔ faster turnaround time

Demo

Comparing Compilers and Interpreters, *cont'd*

- An interpreter maintains control of the source program's execution.
- Interpreters often come with interactive source-level debuggers that allow you to refer to source program elements, such as variable names.
  - AKA symbolic debugger

Comparing Compilers and Interpreters, *cont'd*

- Therefore ...
  - Interpreters are useful during program development.
  - Compilers are useful to run released programs in a production environment.
- In this course, you will ...
  - Modify an interpreter for the Pascal language.
  - Develop a compiler for a language of your choice.
  - You can invent your own programming language!

Take roll!

Key Steps for Success

- Whenever you develop a complex program such as a compiler or an interpreter, key first steps for success are:

  1. Design and implement a proper framework.
  2. Develop initial components that are well-integrated with the framework and with each other.
  3. Test the framework and the component integration by running simple end-to-end tests.

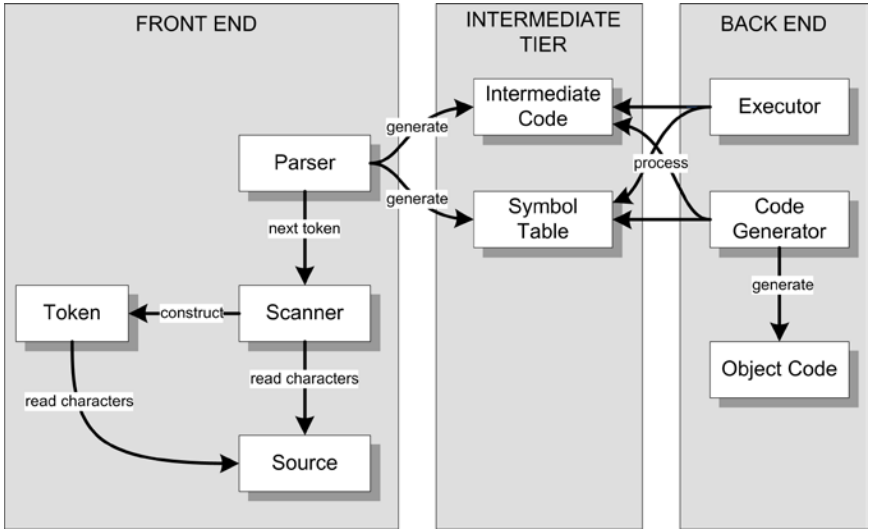
- Early component integration is critical, even if the initial components are greatly simplified and don't do very much.

Key Steps for Success, *cont'd*

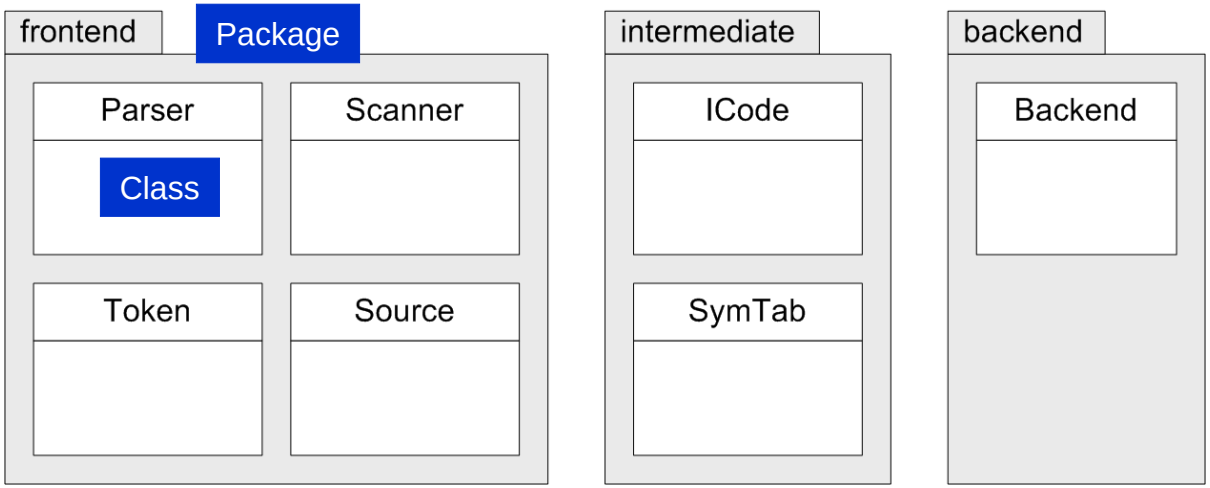
- Test your framework and components and get them working together as early as possible.
- The framework and the initial components then form the basis upon which you can do further development.
- You should always be building on code that already works.

Three Java Packages

FROM:

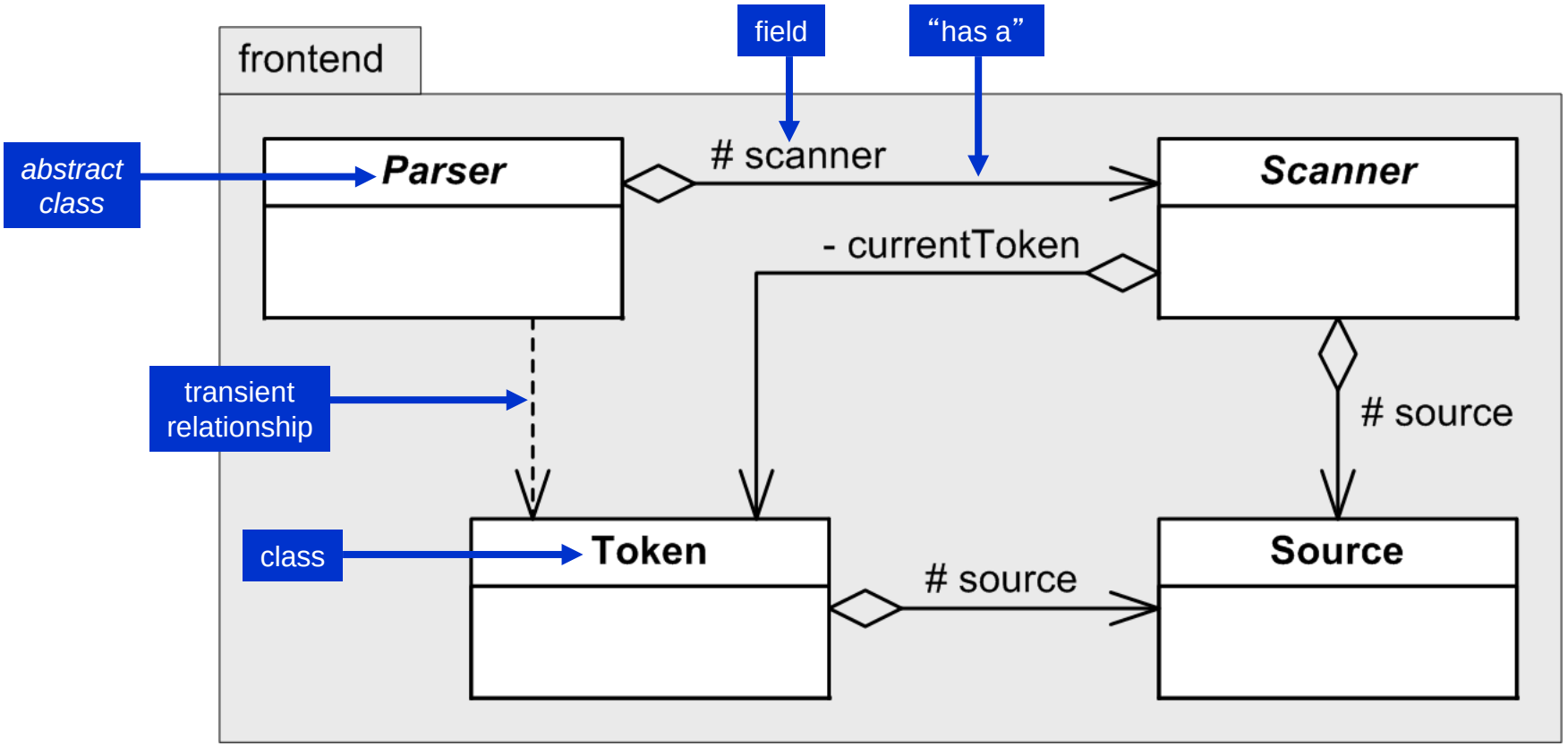


TO:



UML package and class diagrams.

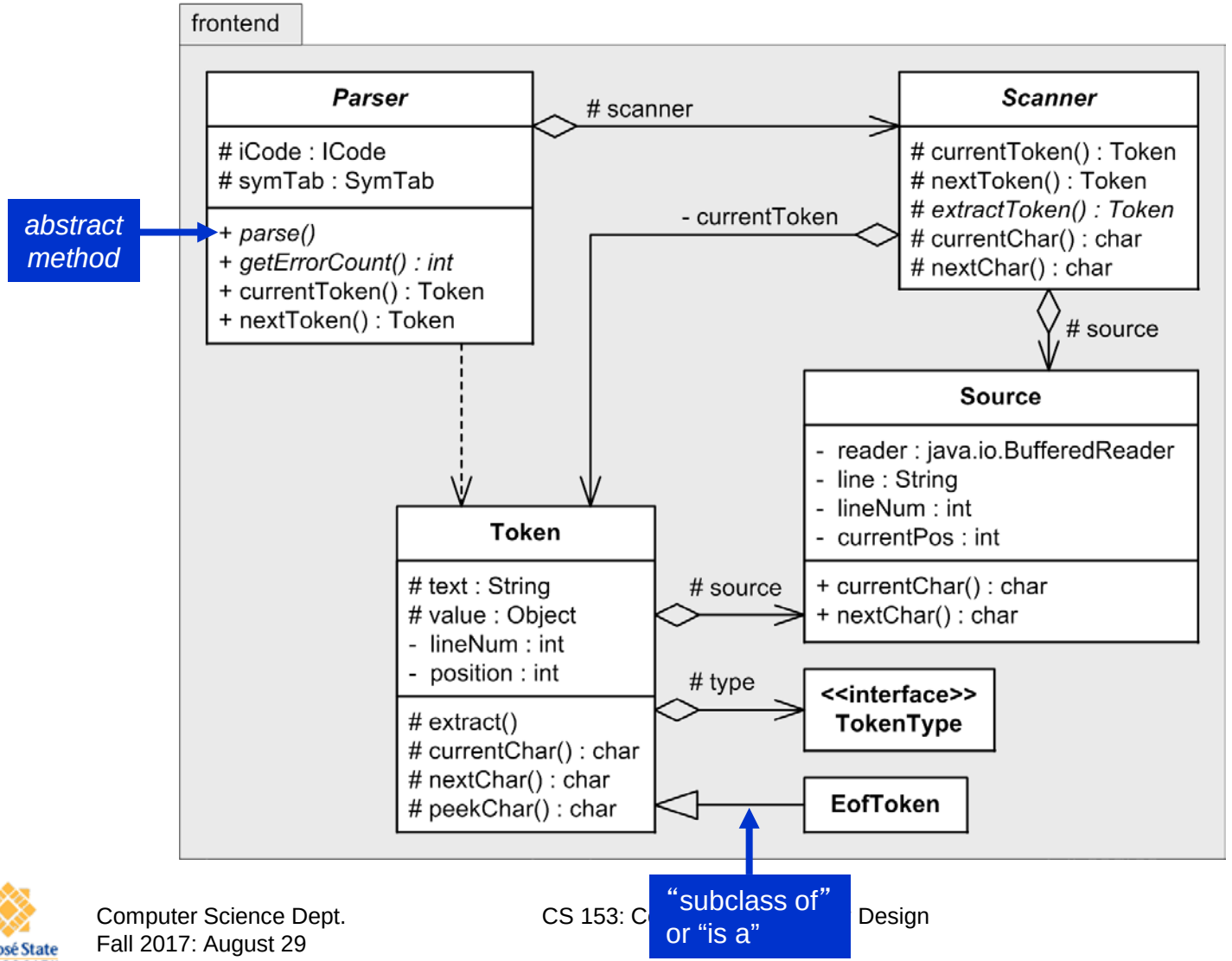
Front End Class Relationships



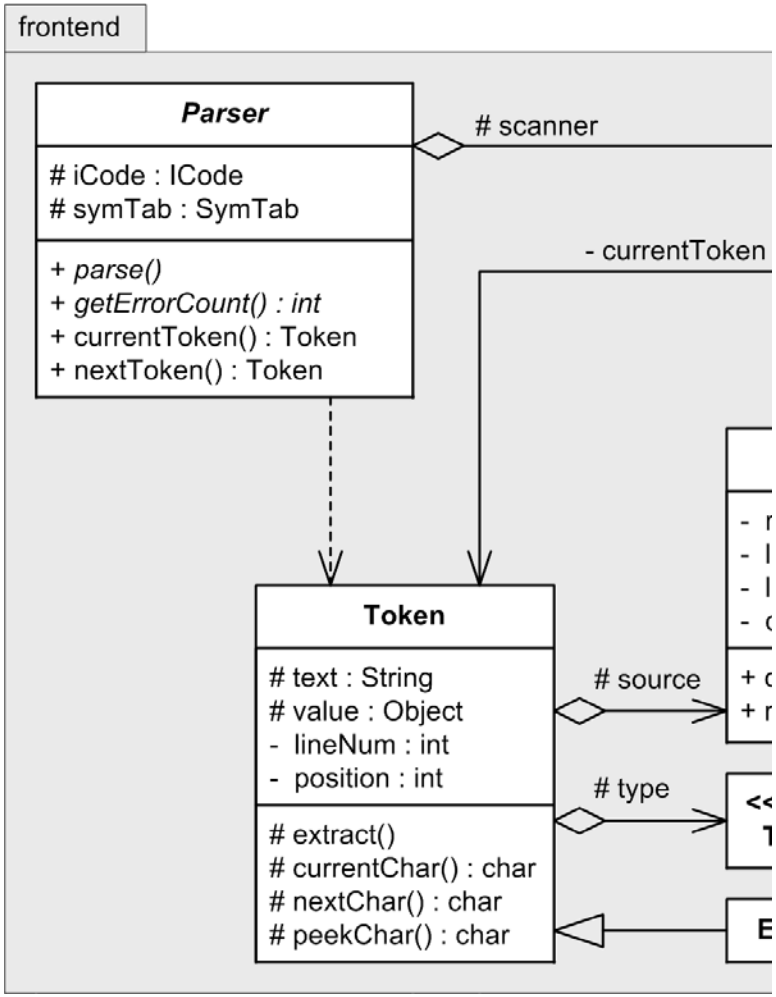
These four **framework classes** should be **source language-independent**.

+	public
-	private
#	protected
~	package

Front End Fields and Methods



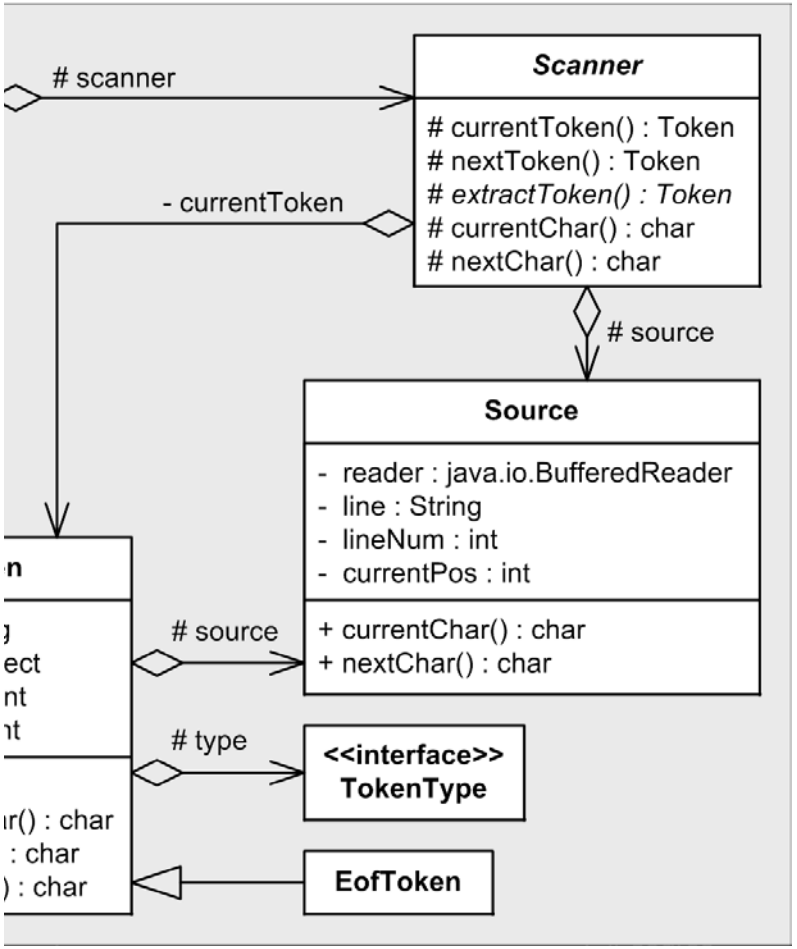
The Abstract Parser Class



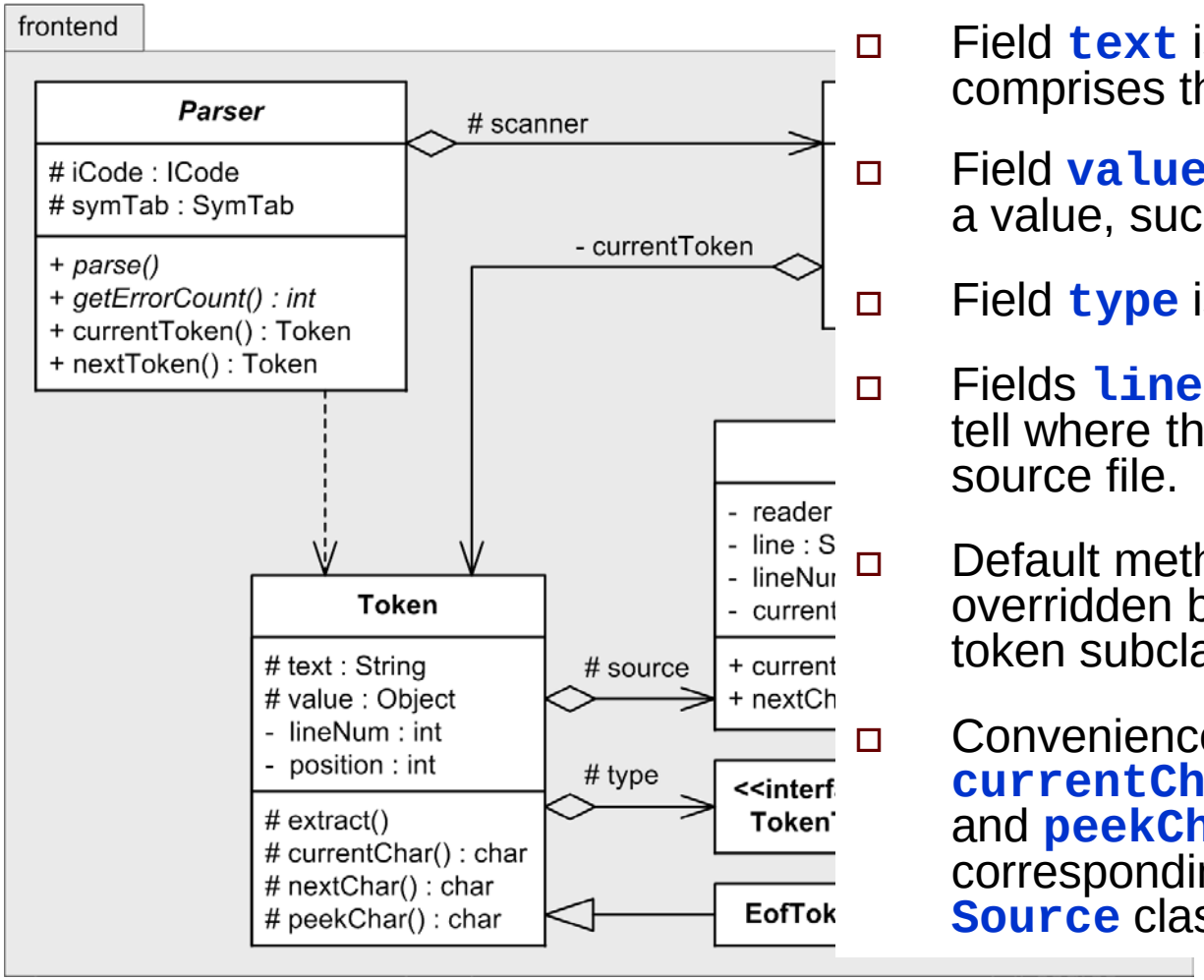
- Fields **`iCode`** and **`symTab`** refer to the intermediate code and the symbol table.
- Field **`scanner`** refers to the scanner.
- Abstract **`parse()`** and **`getErrorCount()`** methods.
  - To be implemented by language-specific parser subclasses.
- "Convenience methods" **`currentToken()`** and **`nextToken()`** simply call the **`currentToken()`** and **`nextToken()`** methods of **`Scanner`**.

The Abstract Scanner Class

- Private field **`currentToken`** refers to the current token, which protected method **`currentToken()`** returns.
- Method **`nextToken()`** calls abstract method **`extractToken()`**.
  - To be implemented by language-specific scanner subclasses.
- Convenience methods **`currentChar()`** and **`nextChar()`** call the corresponding methods of **`Source`**.

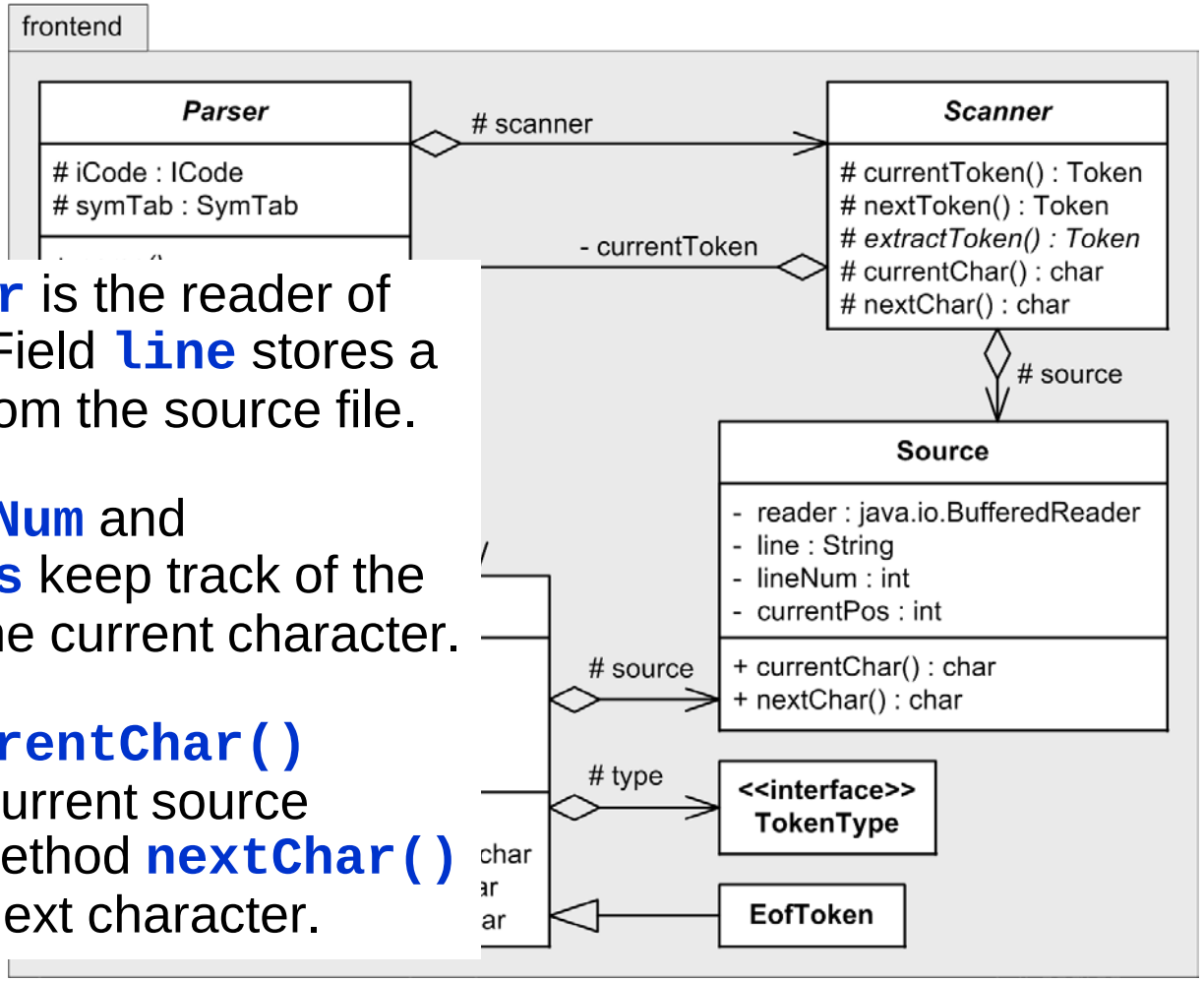


The Token Class



- 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 **`lineNum`** and **`position`** tell where the token is in the source file.
- Default method **`extract()`** will be overridden by language-specific token subclasses.
- Convenience methods **`currentChar()`**, **`nextChar()`**, and **`peekChar()`** 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 **`lineNum`** and **`currentPos`** keep track of the position of the current character.
- Method **`currentChar()`** returns the current source character. Method **`nextChar()`** returns the next character.

Current Character vs. Next Character

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

`currentChar()`	**A**
`nextChar()`	**B**
`nextChar()`	**C**
`nextChar()`	**D**
`currentChar()`	**D**
`currentChar()`	**D**
`nextChar()`	**E**
`nextChar()`	***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.

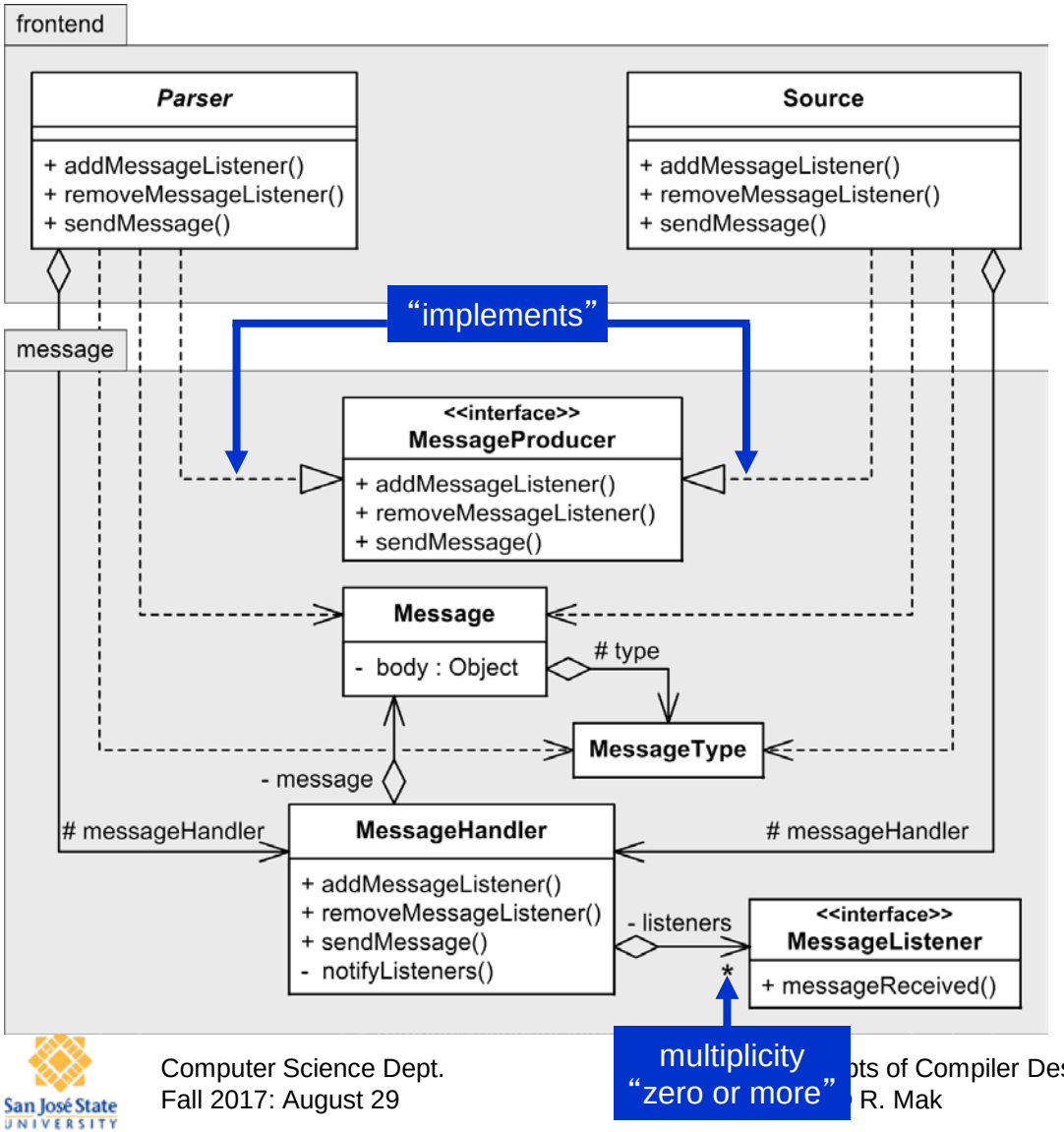
Front End Messages, *cont'd*

- 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