TEA ARCHITECTURE

JULY 12, 2000

PLEASE SEND ALL COMMENTS TO: OPENSOURCE@DIG.COM

PREFACE

This document provides an overview of the Tea Architecture for Java developers.

There are a number of other documents complementary to this reference that cover topics associated with Tea and Tea-related tools. For additional Tea-related documentation, please visit http://opensource.go.com/.

DOCUMENT REVISION AND EVOLUTION

Document Revision Chart

Primary Author(s)	Description of Version	Date Completed
Randall Rader	Initial public release, based on interviews with Brian O'Neill.	June 30, 2000
Michael Rathjen	Minor update.	July 12, 2000

CONTENTS

1. INTRODUCTION

Tea is a strongly typed, compiled programming language, designed to work within a Java-based hosting environment. Tea is designed to provide an enforced separation between data processing and presentation, without sacrificing basic programming constructs.

Tea modules are called **templates**, and they are used for data presentation. Templates are generally used for textual output and HTML. A template cannot directly acquire or modify information, but relies instead on its host to provide data returned from called functions.

The Tea architecture is divided into two primary components: the **Compiler**, and **Runtime** Tea. Tea templates are compiled directly to Java byte code, without the use of a Java compiler. Tea does not use any special kind of interpreter. Because Tea must be integrated into a hosting environment, the implementation is very modular at both the compiler and runtime levels.

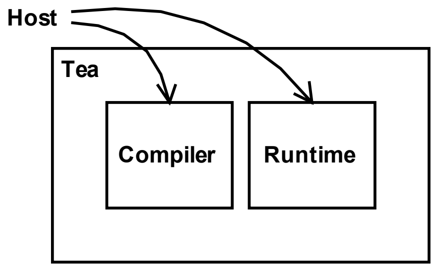

Figure 1 - Basic Tea Architecture

2. THE TEA COMPILER

2.1 Error Support

Even though types are not explicitly declared in Tea, type-checking errors can still be detected at compile time, rather than at runtime. The error generation files are defined in the **resources** directory, in the following property files:

- Scanner.properties
- Parser.properties
- TypeChecker.properties
- Compiler.properties

Error messages in these files are language-specific. Additional files may be generated to provide error support for users with alternate language requirements.

2.2 Compiler Flow

A Tea template goes through five stages when it is compiled into a Java class file, typical of most compilers:

1. A template first passes through Pre-processing;
2. The Scanner and Parser run the template through Syntax Analysis;
3. The Type Checker then performs Semantic Analysis;
4. The results are passed through a basic optimizer;
5. Finally, the Code Generator creates the Java class file.

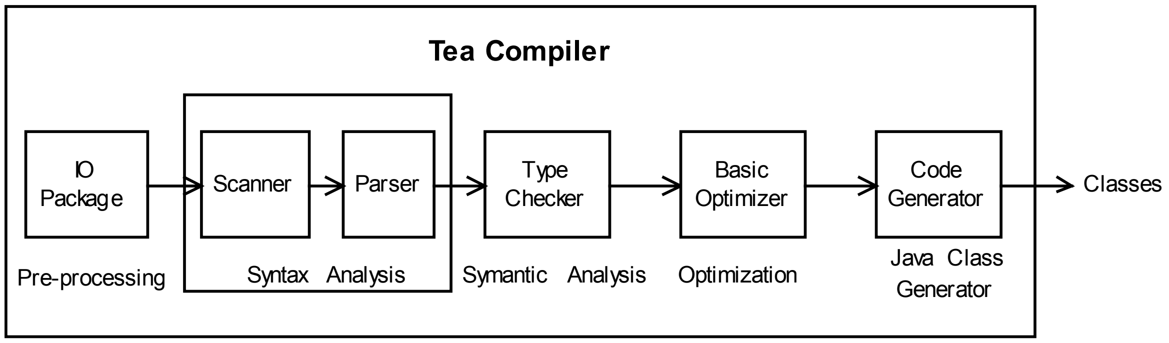

Figure 2 – Tea Compiler Flow

2.2.1 Pre-processing ("IO" package)

The "IO" Package is a basic pre-processor. It takes templates from the source reader and analyzes string states in the code and text regions. The "IO" Package strips the template and code tags (**<%** and **%>**) and reassigns string literal delimiters (' and ") to turn the templates into a "pure" code state, then passes the results to the Scanner for syntax analysis.

Here is an example of a very basic template before and after pre-processing.

Before:

```
<% template HelloWorld ( ) %>
Hello World!
```

After:

```
template HelloWorld ( )
"Hello World!"
```

2.2.2 Syntax Analysis (Scanner and Parser)

Tea Compiler Syntax Analysis is performed by the Scanner and the Parser.

2.2.2.1 Scanner

The Scanner strips comments from the template and breaks strings into **tokens**. Tea tokens consist of **reserved words**, **keywords** (which is a subset of reserved words), **operators**, **literals** (which may also be reserved words), and **identifiers**.

Example reserved words include null, true, false, foreach, if, and in. Of those, null, true, and false are not keywords because they have an associated value and are legal expressions. Thus they are literals.

Literals represent constant values and are legal expressions. `"Hello", 56.0, 10, null, true`, and `false`, are examples of literals.

Operators are special symbols, including `+, -, ..., ##`, etc.

Identifiers are everything else, typically representing variable names and object types.

Example:

```
template myPage (String x)
foreach (c in x) {
s=c & "hello"
```

The example above would be broken up as follows:

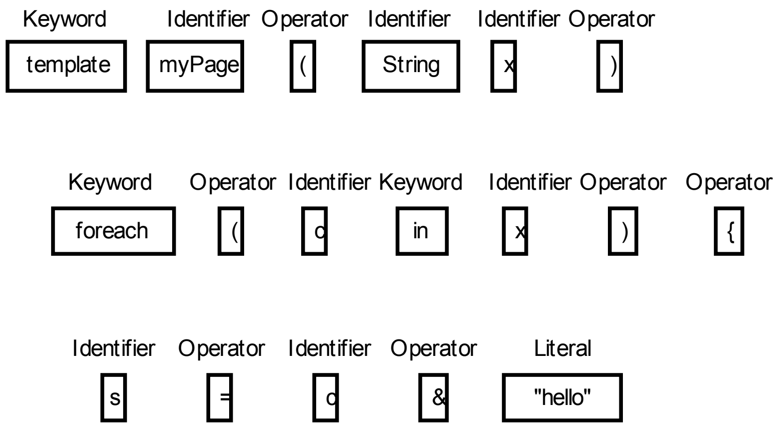

Figure 3 – Example of Tokens

2.2.2.2 Parser

The Tea Compiler uses a handwritten recursive descent parser. The Parser takes the tokens from the Scanner and creates a standard Parse tree. The example above would be parsed as follows:

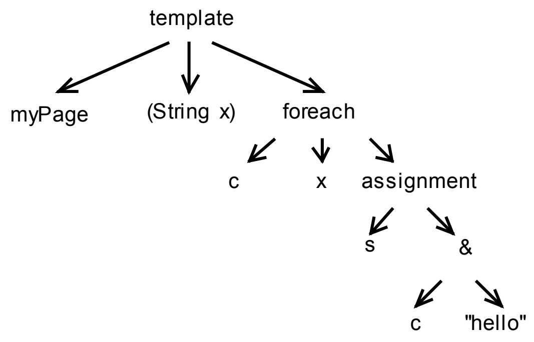

Figure 4 – Example of a Parse Tree

2.2.3 Semantic Analysis (Type Checker)

Semantic Analysis is performed by the Type Checker. The Type Checker traverses the Parse Tree using the **Visitor** design pattern to look at each node and determine its type. (See *Design Patterns*, by Gamma, et al, for additional information about the Visitor design pattern.) The Type Checker then emits the same Parse Tree with new type information. Assigning types at this stage rather than explicitly declaring them allows the compiler to verify errors as failures occur in the Parse Tree.

2.2.4 Optimization

After the template has been parsed and typed, it is run through a basic optimizer, which performs routine operations such as string concatenation, simple addition, and stripping of false branches in if statements.

2.2.5 Code Generator (Java Class Generator)

When pre-processing, syntactic and semantic analysis, and optimization are complete, the Java Class Generator creates a Java class file using a special class file API.

2.2.6 Integration

Tea compiler integration involves configuring error event handling, and enabling the reading of source files and the writing of classes.

2.2.6.1 Error events

Tea error events are handled by error listeners modeled on JDK 1.1 style event listeners. The Host takes the error event objects, which contain a rich set of information.

2.2.6.2 File compiler

The Tea Compiler includes, among others, a File Compiler, which enhances the compiler with the ability to read source files and to write classes. The File Compiler is configurable.

3. RUNTIME TEA

Integrating Runtime Tea involves selecting and configuring a Context class and an Execution class.

3.1 Context class

The Context class defines the runtime receiver interface through included functions, such as **the print (object)** function, which connects to the Host, and is required to enable outputting, and the **toString (object)** function, which allows you to customize. Other functions defined in the standard Context class include the **nullFormat**, **dateFormat**, and **numberFormat** functions. Additional functions may be added to the Context class, and multiple context classes may be used.

3.2 Compiled template skeleton implementation

See the Java Class Generator source for documentation on the compiled template skeleton implementation.