Maintenance Guide

Version 1.1 2 February 2012

Ciaran McHale

www.config4star.org

Availability and Copyright

Availability

The Config4* software and its documentation (including this manual) are available from www.config4star.org. The manuals are available in several formats:

- HTML, for online browsing.
- PDF (with hyper links) formatted for A5 paper, for on-screen reading.
- PDF (without hyper links) formatted 2-up for A4 paper, for printing.

Copyright

AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contents

Chapter 1

Introduction

1.1 The Purpose of the Manual

This manual is intended primarily for people who want to modify or maintain the source code of Config4*. For example, if you want to investigate and fix a bug, add new new functionality, or implement Config4* in another programming language, then you should consider reading this manual.

Even if you are not interested in modifying or maintaining Config4*, you still might find this manual interesting. For example, perhaps you have wondered about the motivation behind a particular aspect of the Config4* API. The information provided in this manual might satisfy such curiosity.

1.2 Structure of this Manual

The chapters in this manual are grouped into two parts.

Part I provides information on the architecture of Config4*. The chapters in Part I explain not just *how* Config4* is designed, but also *why* it was designed that way. If you spend, say, one hour, reading Part I of this manual, then that might save you several days of effort in getting up to speed with the source-code of Config4*.

Part II discusses some of the "rough edges" that exist in Config4*. If you would like to contribute to Config4*, then reading Part II may give you some inspiration on where you could make a good impact.

Part I

Architecture of Config4*

Chapter 2

History

2.1 Introduction

In this chapter, I provide a brief history of the gestation of Config4*, from its initial conception to its first public release 14 years later.

2.2 Motivation

In 1996, I did a five-month consultancy assignment in which I helped a customer design and implement some client-server applications. When we started the implementation phase, we realised that the applications would greatly benefit from having a runtime configuration file. Unfortunately, we did not have a configuration parser to hand, and a quick Internet search did not turn up any suitable ones.

I remember thinking at the time that there must be countless developers around the world in the same position: they needed to write a configuration parser for a project, but deadline pressure meant they could devote no more than, say, a day to writing it, so the resulting parser would be undocumented, lacking in useful features, possibly buggy, and unlikely to be reused in future projects. The re-invention of mediocre configuration parsers by countless developers around the world struck me as being a massive waste of time. I decided that, once the current consultancy assignment was finished, I would spend a few weeks, possibly a month, of my spare time writing a good quality C++ configuration parser.

At the time, I was employed in the professional services department[1] of a software vendor, and my employment contract contained a clause stating that whatever software I wrote—even outside office hours—belonged to my employer. At the time, my employer developed proprietary software exclusively (it would be about ten years before my employer started experimenting with developing open-source software), so I knew I would not be able to release my configuration parser as open-source software. This meant that, unfortunately, the wider world would not be able to benefit from my configuration parser. But at least my colleagues and I would be able to use it in our future consultancy assignments.

2.3 Development

I implemented the configuration parser. As far as I can recall, it provided only *name=value* statements (the value could be a string or a list), the concatenation operator (`"+"`) and scopes. It was certainly a very limited subset of what Config4* contains today. One or two years later, a software developer in another department told me that he needed a configuration parser for a new product that was being developed, and he asked if he could use my parser. I gave him a snapshot of the source code. He modified the code to remove the concatenation operator (because it was unnecessary for his needs) and added some new features (which are outside the scope of this discussion). That modified configuration parser made its way into several new products. I mention that in case any readers are familiar with the Orbix or Artix products from IONA Technologies (since acquired by Progress Software) and recognise some similarities between their configuration file syntax and that of Config4*.

As Eric Raymond famously wrote in *The Cathedral and the Bazaar* [Ray99]:

> Every good work of software starts by scratching a developer's personal itch.

Once the configuration parser was mature enough to scratch my itch, I stopped work on it and went back to other things. But every few years

[1] For any readers not familiar with the term, *professional services* basically means "consultancy and training". It is not to be confused with *personal services* (a euphemism for prostitution), although the hourly rates are similar. Disturbingly, there appear to be many other similarities between the two professions: www.thatwasfunny.com/differences-between-consulting-and-prostitution/49.

I encountered a slightly more stubborn itch that my parser could not scratch. This resulted in me occasionally adding new functionality to the parser. Sometimes, I discovered that the parser's architecture contained a significant limitation or misfeature, so I redesigned it several times over the years.

Eventually, my employer started to experiment with developing open-source software products. Soon after that, my employer agreed to transfer copyright ownership of the configuration parser to me, so that I could release it under an open-source license. However, I decided to add a few more features to the C++ implementation, write a Java version, and write comprehensive documentation before releasing it. I held off writing the Java version until I (prematurely) thought the C++ version was feature complete. This was to reduce the amount of work involved in maintaining two parallel versions.

2.4 Choosing a Name

I had struggled for many years to think of a good name for the configuration parser. However, one day a colleague mentioned the *log4j* project[2] in a conversation; I realised that the name *Config4J* would be good for a Java-based parser, and this name could be adapted to provide *Config4C*, *Config4Cpp*, *Config4Ada* and so on. Hence, the generic name: *Config4**, with the asterisk acting as a wildcard to denote the names of arbitrary programming languages.

2.5 Intertwining Development with Writing of Documentation

Years ago, I discovered a time-consuming but effective way to improve the quality of any software I was developing: write documentation for it. Writing documentation forces me to explain the various features of the software. If I find it difficult to explain a particular feature, then that makes me realise there is something wrong with the feature: perhaps it is badly designed or needlessly complex. This leads me to work in an iterative manner. I write the initial version of a piece of software. Then when I attempt to write documentation, I invariably become aware of problems in my software's architecture. Then I fix the problems in the

[2] http://logging.apache.org/log4j/

software before I continue writing the documentation. I tend to cycle several times—between writing documentation and fixing/enhancing the software—before the software becomes feature-rich, stable, and easy to use. This approach has served me well in my own personal projects that don't have deadline pressure. I don't know if such an approach would work in an environment where time-to-market is critical.

With the benefit of hindsight, I can see that at least half of the features in Config4* have come about because of my attempts to write documentation. So, if you think, "Ciaran did a great job designing Config4*", and you wonder what is my secret for good design, the answer is that I do a mediocre design initially, and then slowly improve it by trying to document it. Of course, I never realise at the time that my initial design is mediocre. I always naïvely think that my design is great. It is only when the design has matured a lot, that I can look back and think, "Wow, my initial design was flawed in *so many* ways".

Chapter 3

Architectural Overview

3.1 Introduction

In this chapter, I explain the main architectural decisions that I made in Config4*.

3.2 Hiding Implementation Details

The public API of Config4* is defined in the `Configuration` class. This is an abstract base class containing very little code. This class provides a static `create()` operation that creates an instance of a concrete subclass. In this way, the implementation details of Config4* are kept separate from its public API.

The concrete subclass is called `ConfigurationImpl`. Its most important instance variable is a hash table. When a `ConfigurationImpl` object is created, its hash table is empty initially. The hash table can then be populated by calling `insertString()`, `insertList()` and `ensureScopeExists()` directly. Alternatively (and more commonly), you can call `parse()`, which, internally, calls those update-style operations.

3.3 Use of Multiple Hash Tables

I know of two potential ways in which a configuration parser might use a hash table to store *name=value* pairs. I use the configuration file below to illustrate the two approaches:

```
foo = "a string";
bar = ["a list", "of", "strings"];
acme {
    widget = "another value";
}
```

The first approach is to use a single hash table to store all the entries. The entries in this hash table can be represented as follows:

> foo → (STRING, "a string")
> bar → (LIST, ["a list", "of", "string"])
> acme → (SCOPE, null)
> acme.widget → (STRING, "another string")

The above notation indicates that each entry in the hash table is a *name* → *tuple* mapping, in which the tuple contains two fields: a *type* (STRING, LIST or SCOPE) and a *value* (if appropriate).

The other approach is to use a separate hash table for each scope. With this second approach, the hash table for the root scope can be represented as follows:

> foo → (STRING, "a string")
> bar → (LIST, ["a list", "of", "string"])
> acme → (SCOPE, <another-hash-table>)

The hash table for the `acme` scope contains:

> widget → (STRING, "another string")

When I wrote my first configuration parser, I used the first approach, that is, a monolithic hash table. I did this for three reasons. First, it was simpler to implement. Second, it was slightly more memory-efficient. Finally, it meant that the implementation of a `lookup<Type>()` operation required a lookup on just one hash table. In contrast, the "separate hash table for each scope" approach can require multiple lookups on hash tables. For example, looking up the value of `"acme.widget"` requires two invocations of `lookup()`:

```
value = rootScopeOfHashTable.lookup("acme").lookup("widget");
```

Several years later, I added the `@copyFrom` statement to my configuration parser and, unfortunately, this introduced a severe performance problem. When using the "monolithic hash table" approach, the implementation of `@copyFrom` has to iterate over the entire contents of the hash table to find the relevant entries that should be copied. The worst-case scenario for this is when a configuration file has a scope called, say, `defaults`, and many other scopes, each of which contains the following statement:

```
@copyFrom "defaults";
```

In such a scenario, parsing the configuration file takes $O(N^2)$ time, where N is the number of entries in the configuration file. That $O(N^2)$ performance problem disappears if, instead, a separate hash table is used for each scope. For that reason, I redesigned Config4* to use a separate hash table for each scope.

The hash table used by the `ConfigurationImpl` class is implemented by the `ConfigScope` class. The (type, value) tuple used in the above discussion of hash tables is implemented by the `ConfigItem` class.

3.4 Why Creation and Parsing are Separate Steps

With Config4*, *parsing* of a configuration file is kept separate from the (initially empty) *construction* of the `Configuration` object. For example, in C++, you write:

```
cfg = Configuration::create();
cfg->parse("foo.cfg");
```

Things were not always that way. When I wrote my first configuration parser, parsing of a configuration file *was* performed in the constructor. This resulted in slightly shorter application code:

```
cfg = Configuration::create("foo.cfg");
```

Unfortunately, performing parsing in the constructor turned out to be a source of memory leaks. This is because the parser might encounter an error in the configuration file and, as a result, throw an exception. Throwing an exception from (the parser called from within) the constructor means that the object's destructor is *not* called, so heap-allocated instance variables become memory leaks. In theory, all I had to do was write the constructor as shown below:

```
ConfigurationImpl::ConfigurationImpl(const char * fileName)
{
    ... // allocate memory for instance variables
    try {
        parse(fileName);
    } catch(const ConfigurationException & ex) {
        ... // free memory of instance variables
        throw; // re-throw the exception
    }
}
```

However, on several occasions, as the project matured, memory leaks crept in due to me adding new heap-allocated instance variables but forgetting to free them in the above `catch` clause. Eventually, I grew tired of that source of recurring memory leaks, and I decided to prevent future re-occurrences by keeping parsing separate from object construction.

Several years after I made that change, I discovered two extra benefits of keeping parsing separate from construction. First, it makes it possible to preset configuration variables. Second, it makes it possible to set a security policy before parsing a configuration file.

3.5 Limitations

There are very few arbitrary limitations in the implementation of Config4*. For example:

- Aside from available RAM, there is no arbitrary limit on the size of a configuration file, or the length of lines within a configuration file.
- There is no arbitrary limit on the length of an identifier (that is, the name of a scope or variable), on the length of a string value, or on the maximum number of strings in a list.
- There is no arbitrary limit on the maximum number of nested `@include` statements. (However, operating systems typically place a limit on the number of open file descriptors within a process; that will limit the number of nested `@include` statements.)
- There is no arbitrary limit on the number of scopes or how deeply they can be nested. There is no arbitrary limit on the number of entries in a scope. The scope's hash table will resize itself when it starts to fill up.

I think you get the idea: arbitrary limitations are not common in Config4*. Having said that, Config4* *does* have some limitations, as I now discuss.

3.5.1 Number of `uid-` Entries

There can be no more than 10^9 `uid-` entries in a configuration file. That is an arbitrary limitation, albeit a large one. That limitation arises because Config4* uses a 32-bit integer to store the `uid-` counter, and the

maximum value of such an integer is $2^{31} - 1 = 2,147,483,647$. That value is a 10-digit number. I decided to round down the maximum value of the `uid-` counter to $999,999,999$ so that the expanded form of an `uid-` identifier contains nine digits instead of ten.

How likely is it that a configuration file will exceed the limit on `uid-` entries? I don't think many people will be creating big enough configuration files to have to worry about exceeding this limit within the next few years (I'm writing this statement in 2011). But the software and databases that underpin an Internet search engine, such as Google, might. If you work for such a company and wish to increase this limit, then you should do the following. Edit the `UidIdentifierProcessor` class, change the declaration of the instance variable from being a 32-bit integer to being a 64-bit one, and modify the code so that when the value of this instance variable is formatted as a string, the string contains more than nine digits.

3.5.2 Lack of File name and Line Number Information

Consider the following scenario involving two configuration files: `foo.cfg` and `bar.cfg`. The `foo.cfg` file contains the following:

```
@include "bar.cfg";
... # define some configuration variables
```

The `bar.cfg` file contains the following:

```
x = "2"    # missing semicolon
y = "tru"; # misspelling of "true"
```

Now let's consider what happens if we run a program that contains the following code:

```
cfg = Configuration.create();
try {
    cfg.parse("foo.cfg");
    boolean myBool = cfg.lookupBoolean("", "y");
} catch(ConfigurationException ex) {
    System.out.println(ex.getMessage());
}
```

When we run the program, the call to `parse()` fails because of a syntax error, and the following message is printed:

```
bar.cfg, line 2: expecting ';' or '+' near 'y'
(included from foo.cfg, line 1)
```

The error message is very informative. Not only does it correctly report the missing semicolon, it *also* specifies the location of that problem: line 2 of file `bar.cfg`, which was included from line 1 of `foo.cfg`.

Let's assume we insert the missing semicolon and run the program again. Now, `parse()` succeeds, but the call to `lookupBoolean()` fails, and the following message is printed:

```
foo.cfg: bad boolean value ('tru') specified for 'y'; should be one of:
'false', 'true'
```

That error message is less informative that the previous one. It correctly describes the problem, but it does *not* accurately specify the file name and line number of the problematic configuration variable. Instead, it just assumes (inaccurately, in this case) that the problematic variable is defined somewhere in `foo.cfg` rather than in an included file.

The lack of accurate location information in this second error message is due to that information *not* being recorded in Config4*'s internal hash tables. That information is not recorded because of a combination of my laziness and my concern for efficient memory use, as I now explain.

When the Config4* parser encounters a *name=value* statement or the opening of a scope, it enters information into the internal hash tables by calling one of the following operations: `insertString()`, `insertList()` or `ensureScopeExists()`. The following discussion applies to all those operations, so, for conciseness, I will discuss just `insertString()`.

The first configuration parser I implemented—the original ancestor of Config4*—did not have `@include` or `@copyFrom` statements. The `insertString()` operation took an extra parameter that indicated the line number at which the configuration variable was defined:

```
void insertString(String scope, String name, String value, int lineNum);
```

That line number was recorded in the hash table entry for the variable. If an operation, say, `lookupBoolean()`, could not translate a variable's value into the appropriate type, then the text message in the exception thrown could specify the line number (obtained from the entry in the hash table) and the file name (obtained by calling `cfg.fileName()`) of the problematic variable. This approach worked well, and it had minimal memory overhead: just a 4-byte integer (to store the line number) for each entry in a hash table.

Several years later, I added the `@include` statement. I realised that if error messages were to specify accurate location information, then it

would no longer be sufficient to pass a line number to `insertString()`. That operation would have to be modified to take a parameter that specified a list of *(fileName, lineNumber)* tuples, as shown in the following pseudocode:

```
void insertString(String scope, String name, String value,
                  List[(fileName, lineNum)] locationInformation);
```

That list of tuples could be stored in the hash table entry for a variable. Then an error message produced by, say, `lookupBoolean()` could indicate the file name and line number of the problematic variable, *plus* the path, if any, that traces the `@include` statements from the main configuration file to the file that contains the problematic variable. (Ideally, the path would trace not just `@include` statements, but also `@copyFrom` statements.)

Implementing that enhancement could result in a significant memory overhead. For example, let's assume there are 100 variables defined in `bar.cfg`, which is included from `foo.cfg`. Would the enhancement result in there being 100 copies of the string `"bar.cfg"` and another 100 copies of `"foo.cfg"`—separate copies for each entry in the hash table? Avoiding such redundant copies would require the implementation of a pool of unique strings, which would add complexity to the implementation of Config4*.

Would such memory overhead and/or complexity be a worthwhile investment to obtain more informative error messages? I don't know. So far, I have found it straightforward to search through a file (and included files, if any) in a text editor to find a problematic variable. But then, I have been dealing mainly with configuration files that contain only a few hundred or few thousands lines of text. Perhaps, in a few years time, somebody will be working with configuration files that contain millions of lines of text, a complex interaction of deeply nested and re-opened scopes, all compounded with `@include` and `@copyFrom` statements. In such a scenario, accurate location information in error messages might improve ergonomics significantly.

3.5.3 Information lost with round-trip `parse()` and `dump()`

If you `parse()` a configuration file and `dump()` it back out again, then you do *not* get back the full contents of the original configuration file.

As first sight, this might appear to be a limitation of the `dump()` operation. However, that view is inaccurate. To better understand the

issues involved, consider a configuration file that contains the following statement:

```
log_dir = getenv("FOO_HOME") + "/logs/" + exec("hostname");
```

The Config4* parser evaluates the expression and stores the result in the hash table for the configuration scope. For example, if `FOO_HOME` has the value `"/opt/foo"` and `hostname` returns `"host1"`, then the hash table will contain the following entry:

log_dir → (STRING, "/opt/foo/logs/host1")

The `dump()` operation simply dumps the contents of the hash table, and thus produces:

```
log_dir = "/opt/foo/logs/host1";
```

So, the limitation is not actually with the `dump()` operation, since it is faithfully reproducing the contents of the hash table. Instead, the limitation is with the parser and hash table representation, because they record a *processed* (rather than the *original*) version of what was in the input configuration file.

You might think this limitation would be easy to overcome: just have the hash table store the original expression rather than the result of evaluating the expression. However, such an approach would suffer from two significant problems.

The first problem is an increased performance overhead. This is because the overhead of evaluating the expression would not be incurred exactly once, when parsing the input file. Instead, the overhead would be incurred *every time* a `lookup<Type>()` operation is invoked (which might be multiple times in an application).

The second problem is that the internal architecture of Config4* would have to be redesigned completely to enable `dump()` to reproduce the input configuration file exactly. In particular, something more complex than a hash table would be required to store the parsed information. This is because:

- A hash table does *not* preserve the order in which entries were added to it, but such an order-preservation guarantee would be required for `dump()` to reproduce the input file accurately.
- The parser discards comments when parsing the input file. These would have to be preserved in the internal representation for `dump()` to be able to reproduce the input file accurately.

In addition, it is difficult to see how an efficient internal representation might preserve commands such as `@include`, `@copyFrom`, `@remove`, `@if-then-@else`, and conditional assignment (`"?="`) statements rather than just the *name*=*value* pairs resulting from executing those commands.

In summary, it would require a significant amount of rework to the architecture of Config4* to be able to implement a `dump()` operation that could reproduce the input configuration file accurately. In my opinion, the benefits would not justify the amount of work involved.

The preceding discussion invites a question: Why did I implement a `dump()` operation that reproduces the input configuration file so inaccurately? The answer is that my original intention in implementing `dump()` was to provide a debugging tool: the output of `dump()` helped me to check that I had implemented the hash table-based internal representation correctly. It was only later that I realised `dump()` might be useful for other purposes too, such as converting, say, an XML file into Config4* format, or storing the user preferences of a GUI-based application.

3.6 The Multi-step Build Process

Some software projects have a straightforward build system: compile all the source-code files, and then combine them to form a library, executable or jar file. Some other software projects require a multi-step build system, for example:

1. Compile a subset of the source-code files to produce a utility program, such as a code generator.

2. Run that utility program to generate additional source-code files.

3. Compile the newly generated files plus the remaining source-code files, and combine them to form a library, executable or jar file.

Config4* requires that type of multi-step build system. This is due to the default security policy, which must be embedded within the Config4* library.

The first step of the build system is to compile a few source-code files to produce a simplified version of `config2cpp` called `config2cpp-nocheck`, or a simplified version of `config2j` called `Config2JNoCheck`. In a moment, I will explain how and why these "no check" versions of the utilities are simplified. But before that, I will discuss the remaining steps of the build system.

The second step of the build system is to run the newly compiled utility on the `DefaultSecurity.cfg` file, and produce a C++ or Java class called `DefaultSecurity`.

The third step of the build system is to compile this newly generated class plus the remaining source-code files, and combine them to form a library and executable (for C++), or a jar file (for Java).

The (non-simplified) `config2cpp` and `config2j` utilities *cannot* be used in step 2 of the build system because they make use of the Config4* library, which is not built until step 3 of the build system. (In particular, it is the schema-generation functionality of those utilities that makes use of the Config4* library.)

The (simplified) `config2cpp-nocheck` and `Config2JNoCheck` utilities omit the schema-generation functionality. In doing so, they avoid any dependency on the Config4* library. These simplified utilities are used only by the build system: they are *not* copied into the `bin` directory for use by regular users of Config4*.

3.7 Features Implemented with Delegation

Two important pieces of functionality (fallback configuration and security policies) are implemented by having the user-created `Configuration` object delegate to another, but internal, `Configuration` object. In this section, I briefly explain how the delegation works.

3.7.1 Fallback Configuration

One of the instance variables in the `ConfigurationImpl` class is a C++ pointer or Java reference to another `ConfigurationImpl` object. In Config4J, this instance variable is called `fallbackCfg`, while in Config4Cpp it is called `m_fallbackConfig`. (In general, Config4Cpp uses `"m_"` as a prefix on the names of member, that is, instance, variables.) The constructor initialises this instance variable to be a C++ nil pointer or Java `null` reference. The `setFallbackConfiguration()` operation sets it to point to another `Configuration` object.

The `Configuration` class defines many type-specific lookup operations, such as `lookupList()`, `lookupString()` and `lookupBoolean()`. The implementations of those operations, either directly or indirectly, invoke a more primitive operation called `lookup()`, which looks for the desired entry in the hash tables. If `lookup()` finds the entry, then it returns a pointer/reference to the relevant hash table's `ConfigItem`; otherwise, it

continues the search by delegating to the fallback configuration object. This can be seen in the abridged pseudocode algorithm shown below:

```
ConfigItem lookup(String fullyScopedName, String localName, ...)
{
    ConfigItem    item;
    item = ...; // search for fullyScopedName in the hash tables
    if (item == null && fallbackCfg != null) {
        item = fallbackCfg.lookup(localName, localName, ...);
    }
    return item;
}
```

3.7.2 Security Policy

The enforcement of Config4*'s security policy relies on the interaction between three items: (1) a singleton object representing the default security policy; (2) two instance variables in the `ConfigurationImpl` class; and (3) an operation called `isExecAllowed()`. I will discuss each of those in turn.

In Section 3.6 on page 17, I mentioned that the `Config2JNoCheck` or `config2cpp-nocheck` utility is used to convert `DefaultSecurity.cfg` into a class called `DefaultSecurity`. In effect, that generated class provides an embedded version of the configuration file that provides the default security policy. The `DefaultSecurityConfiguration` class: (1) inherits from `ConfigurationImpl`; (2) uses its constructor to parse the embedded `DefaultSecurity` configuration file; and (3) provides a singleton object. That singleton object is the default security policy used by all `Configuration` objects.

The `ConfigurationImpl` class contains two instance variables that are used to implement the security policy:

```
// Java instance variables
Configuration    securityCfg;
String           securityCfgScope;
// C++ instance variables
Configuration *  m_securityCfg;
StringBuffer     m_securityCfgScope;
```

The `ConfigurationImpl` constructor initializes the (`m_`)`securityCfg` variable to point to the `DefaultSecurityConfiguration` singleton object, and initialises (`m_`)`securityCfgScope` to be an empty string (denoting the root

scope). A programmer can update those instance variables by calling the `setSecurityConfiguration()` operation.

Recall that there are three ways Config4* can execute an external command:

```
cfg.parse("exec#command");
@include "exec#command";
name = exec("command");
```

Whenever Config4* is asked to execute an external command, it calls `isExecAllowed()` to determine if the security policy in effect allows the specified command to be executed. That operation makes its decision by comparing details of the specified command to the `allow_patterns`, `deny_patterns` and `trusted_directories` variables that appear in the (`m_`)`securityCfgScope` scope of the (`m_`)`securityCfg` configuration object.

3.8 Thread safety

Implementations of Config4* are *not* thread safe. The lack of thread safety was a deliberate design decision, and was based on two considerations.

First, some programming languages do not provide portable synchronisation facilities. Thus, avoiding reliance on such facilities helps to keep the architecture of Config4* portable across programming languages.

Second, all the operations in the API of Config4* fall into one of two categories: either they are *query* operations such as `lookup<Type>()`, or they are *update* operations such as `parse()`, `ensureScopeExists()`, `insert<Type>()`, `remove()` and `empty()`. I imagine that most multi-threaded, Config4*-based applications will use a single thread to call one or more update operations to initialise a `Configuration` object. Once initialisation is complete, the `Configuration` object can then be made available to other threads within the application, but those threads will invoke only query operations on it. It *is* safe for multiple threads to invoke query operations concurrently.

Chapter 4

Portability

4.1 Introduction

Many people consider a piece of software to be portable if it can be built with different brand names of compiler on different operating systems. I think that is an overly narrow view of portability. In my opinion, the portability of software is increased if the software refrains from using the most recently introduced features of a programming language, because that means the software is portable to both new and older versions of a programming language.

4.2 Compatibility with Old Compilers

During the 15 years I spent working in the professional services department of a software vendor, I visited many companies who were slow to upgrade software that they were using. An example of how slow some companies are to upgrade is provided by Microsoft. Microsoft released version 6.0 of the Visual Studio C++ compiler in 1998. In 2010, some developers are *still* using that version of the compiler, despite the fact that Microsoft have brought out five major newer releases in the intervening 12 years.

When implementing Config4Cpp and Config4J, I decided to avoid using relatively new language features whenever possible. By "relatively new", I mean less than 5 or 10 years old. My intention is that Config4Cpp and Config4J can be used not just with the latest versions of compilers,

but with older compilers too.

For Config4J, this means that annotations, generics and enumerations (all introduced in Java 5) have been avoided. I also decided to avoid using the `assert` keyword (introduced in Java 1.4); instead I wrote the `Util.assertion()` operation to provide similar functionality.

For Config4Cpp, I have been happy to use exceptions, namespaces, and single inheritance, but I avoided `static_cast<>`, multiple inheritance, template types and the standard C++ library (instead I rely only on the standard C library). Further discussion of this is given in the *Config4* C++ API Guide*.

4.3 Platform-specific Issues

In general, portability of Java code is better than portability of C++ code. This has resulted in Config4J containing very little platform-dependent code. In fact, the only non-trivial platform-dependent code is in the implementation of `getenv()` in the `ConfigurationImpl` class. You can find a discussion of this in comments in the code.

In Config4Cpp, I have encapsulated platform-specific code in the files `platform.h` and `platform.cpp`. Interested readers should look at the comments in those files to see the approach taken to dealing with platform-specific issues.

Chapter 5

Coding Conventions

5.1 Introduction

This chapter discusses conventions used in the source code of Config4J and Config4Cpp.

5.2 Naming Conventions

Most identifiers in Config4J and Config4Cpp are spelled using mixed capitalisation without any underscores—what is sometimes called "camel case" convention. Class names begin with an upper-case letter (for example, `LexToken`), while the names of operations and variables begin with a lower-case letter (for example, `lookupString()`). Named constants are spelled in all upper-case with underscores (for example, `CFG_SCOPE`, which is defined in the `Configuration` class).

The naming conventions discussed above should be familiar to most Java programmers. Those naming conventions are less widely used among C++ programmers, where some people prefer identifier names to consist of lower-case letters and underscores instead of camel case. I decided to use the Java naming convention in Config4Cpp to provide consistency in the public API and, to a lesser extent, implementation code.

Some minor differences in the naming conventions arise with regard to what C++ programmers call an *instance* (or *member*) variable, and what Java programmers call a *field*. In Config4Cpp, I use `"m_"` as a prefix on

the names of such variables because that convention is deeply ingrained in my memory muscles. I do not use any such prefix in Config4J. If such a variable has a public accessor operation, then this is given a `"get"` prefix in Config4J, but not in Config4Cpp. For example, a C++ instance variable called `m_foo` might have an accessor operation called `foo()`, while the Java counterparts are called `foo` and `getFoo()`.

Java requires that the name of a source-code file match the name of the class contained in it. For example, the file `Foo.java` contains a class called `Foo`. C++ does not have this same requirement, but the source-code of Config4Cpp uses that naming convention.

5.3 Use of a Single Namespace/Package

All the source code of Config4Cpp is in a single `namespace`. The default name of this is `config4cpp`, but you can change that by editing the `<config4cpp/namespace.h>` file. Likewise, all the source code of Config4J is in a single `package`. The name of this is `org.config4j`, but you should be able to change the package name easily by doing a global search-and-replace on the source-code files. Alternatively, an integrated development environment (IDE) might provide a "refactoring" menu option to change the package name.

Putting all the source code into a single `namespace`/`package` was done deliberately to help companies avoid a potential versioning link problem, as I now explain.

Let's assume your company makes and sells a software library called Foo. Internally, Foo uses version p.q of Config4Cpp. One of your customers is trying to build an application that links with both the Foo library and also the Bar library (which is sold by another company). The Bar library also uses Config4Cpp internally, but, unfortunately, it uses the newer version x.y. If versions p.q and x.y of Config4Cpp are not binary compatible, then you are likely to receive technical support calls from your customer, asking you to urgently upgrade to the x.y version of Config4Cpp, so the customer can build their application without link errors.

You can avoid the need to deal with such technical support requests if you take two steps when implementing the Foo library. First, change the `namespace` of Config4Cpp when compiling it for use inside the Foo library. Second, if the Foo library needs to expose a configuration API to users of Foo, then do *not* expose the Config4Cpp API directly. Instead,

look at the source code of some of the demo programs (discussed in the *Config4* Getting Started Guide*) to see how you can put a Foo "wrapper" API around the Config4Cpp API. If you take those two simple steps, then your customers will not encounter the linking problems discussed above.

There is, unfortunately, a price to be paid for the above advice: code bloat. In particular, your customer's application will end up being linked with several copies of Config4Cpp (each compiled in a different `namespace`). At the time of writing, each copy of the Config4Cpp library will add a few hundred KB to an application. However, I think such code bloat is an acceptable price to pay when when modern computers have several GB of RAM.

5.4 Indentation and Spacing

Indentation in source-code files is with TAB characters. Please configure your text editor or integrated development environment (IDE) so that a TAB character displays as 4 spaces.[1] Lines of source code should never be more than 80 columns wide.

Please do not put spaces around `"("` or `")"` characters when invoking a function. Also, the opening `"{"` in an if-then-else statement, while-loop or for-loop should not be on a separate line (unless you need to avoid line wrap).

```
obj.someOp ( parameter ); // bad
obj.someOP(parameter);    // good

if (someCondition)        // bad
{
    ...
}
if (someCondition) {      // good
    ...
}
```

[1] If you use the `vim` text editor, then the following information may be useful. By default, `vim` treats a TAB character as 8 spaces. You can override this for C++ and Java files by placing the following line into the `.vimrc` file in the directory specified by the `HOME` environment variable:

```
autocmd FileType cpp,java setlocal tabstop=4 shiftwidth=4
```

Chapter 6

Parsers and Lexical Analysers

6.1 Introduction

In this chapter, I discuss the approach taken to implement the parsers and lexical analysers that are used in Config4*.

6.2 Avoidance of Parser Generators

There are many tools available that can generate a lexical analyser or parser. However, I decided to write those parts of Config4* by hand. I did this for two reasons.

First, when I was an undergraduate student at university, I learned how to use the UNIX tools `lex` (for generating a lexical analyser) and `yacc` (for generating a parser). But I also learned how to write a lexical analyser and recursive-descent parser by hand. I prefer the hand-written approach, at least for small grammars.

Second, one of my hopes is that people will volunteer to implement Config4* in other programming languages. It should be straightforward to port a hand-written lexical analyser and parser to another (object-oriented or procedural) programming language. This is because those components are implemented with just "normal code", so the port from C++ to, say, Ada would be just a port of "normal code".

In contrast, if I had used, say, `lex` and `yacc` in the C++ implementation, and you wanted to implement an Ada version, then you would have had to do the following.

1. Find `lex`- and `yacc`-like tools for Ada.

2. Translate the `lex` and `yacc` files into the syntax required for their counterparts in Ada. Doing this would probably require you to learn how to use `lex` and `yacc` and their Ada counterparts.

3. If there is not a one-to-one match of features in `lex` and `yacc`, and their Ada counterparts, then you would have to to figure out how to emulate some of the `lex` and `yacc` functionality in their Ada counterparts. If you did this incorrectly, then you might introduce subtle bugs in the lexical analyser or parser.

The above steps would introduce complexity that does not exist with a hand-written lexical analyser and parser.

6.3 Lack of Error Recovery

The parsers in many compilers implement error recovery. This means that when the parser encounters an error, it reports the error and then tries to recover by skipping input until the parser encounters, say, the next semicolon (indicating the end of a statement) or close brace (indicating the end of a scope). Having recovered, the parser can examine the rest of the input file to check if it contains any additional errors. Error recovery enables a single run of a compiler to report several errors. This can speed up software development, especially if the compiler is slow.

To keep the Config4* parser simple, it does *not* make any attempt to do error recovery. Instead, when the parser encounters an error, it reports the error and stops immediately. The lack of error recovery simplifies the design of the Config4* parser. And because the parser is extremely fast, I do not feel it is particularly burdensome for a user to fix one error at a time and then attempt to re-parse a configuration file.

6.4 A Hierarchy of Lexical Analysers

The Config4* library provides parsers—and, hence, lexical analysers—for two distinct languages: (1) the configuration language; and (2) the

schema language. There is a lot of overlap between the lexical analysers used to implement the two parsers. For example, the lexical analysers recognise string literals, identifiers and punctuation symbols (such as `"="` and `","`) in the same way. However, each lexical analyser must recognise distinct sets of keywords and function names.

To avoid code bloat and simplify maintenance, the lexical analysers are implemented as a class hierarchy that consists of a base class plus two sub-classes. The base class, `LexBase`, implements almost all the functionality required of the two lexical analysers. However, this base class is *not* hard-coded with details of keywords or function names. Instead, two of its instance variables are pointers/references to arrays that provide information about keywords and function names. The constructors of the `ConfigLex` and `SchemaLex` subclasses simply initialise those arrays with information about the keywords and function names specific to a particular language.

6.5 Parsing `@if`-then-`@else` statements

Config4* contains a bug in how `@if`-then-`@else` statements are parsed. In this section, I start by explaining a subtle problem that underpins the bug. Afterwards, I explain the buggy approach I took to addressing the problem.

6.5.1 A Subtle Problem

Consider the following configuration file:

```
@if (osType() == "unix") {
    # then part
    install_dir = getenv("FOO_HOME");
    log_dir = install_dir + "/logs/" + exec("hostname");
} @else {
    # else part omitted for brevity
    ...
}
```

Let's assume the above file is parsed on a Windows-based computer, so the condition evaluates to false. Because of this, the parser must *ignore* all the statements in the "then" part of the `@if`-then-`@else` statement, and instead process all the statements in the "else" part.

But what is meant by *ignore*? Obviously, the parser must not update the `Configuration` object with *name=value* pairs for `install_dir` or `log_dir`.

Actually, *ignore* implies more than just "do not update". It also implies "do not query" the `Configuration` object. To understand why, consider the second assignment statement in the above configuration file. When parsing that statement, the parser must not try to query the value of the `install_dir` variable because the "do not update" behaviour means that that variable does not exist.

Ignore also implies that function calls should not be evaluated. There are two reasons for this. The first reason is optimisation—for example, it would be a waste of CPU cycles to execute the `hostname` command and capture its output if that output will then be silently ignored. The other reason is that the parameter passed to a function might be a variable that (because of the "do not update" behaviour) might have not been assigned a value.

Ideally, when the parser is ignoring the "then" part of the `@if`-then-`@else` statement, the *only* thing it should be doing is making sure there are no syntax errors in the statements being ignored. Unfortunately, the grammar used by the parser makes this difficult to do in an efficient and easy-to-code manner. To understand why, consider the following abridged assignment statements:

```
foo = "hello" + ... ;
foo = getenv( ... ) + ... ;
foo = [ ... ;
foo = split( ... ) + ... ;
foo = bar + ... ;
```

The parser processes each of the above statements as follows.

First, the parser obtains the `foo` identifier token from the lexical analyser. At this point, the parser does not know if the statement is an assignment statement or the opening of a scope. To find out, the parser obtains the next token from the lexical analyser. If that next token is an open brace (`"{"`), then the parser knows it is parsing the opening of a scope. Conversely, if the token is an assignment operator (`"="` or `"?="`), then the parser knows it is parsing an assignment statement.

Now that the parser knows it is dealing with an assignment statement, it needs to determine if the assignment operator is followed by a string expression or a list expression. To do this, the parser obtains the next token from the lexical analyser. If that next token is a string literal (such as `"hello"`) or the name of a function that returns a string

(such as `"getenv("`), then the parser knows it must parse a string expression. Conversely, if the token is `"["` or the name of a function that returns a list (such as `"split("`), then the parser knows it must parse a list expression.

So far, all that I have explained above is straightforward. However, a problem arises if the token after the assignment statement is an identifier (such as `bar`). In this case, the parser must invoke `type()` on the `Configuration` object to determine the type of the variable specified by the identifier. Unfortunately, when the assignment statement is nested inside the non-executed branch of an `@if`-then-`@else` statement, the "do not query" behaviour *prevents* the parser from being able to invoke `type()`.

6.5.2 An Imperfect Approach to Tackling the Problem

The approach taken by the Config4* parser to ignore the statements in a non-executed part of an `@if`-then-`@else` statement is as follows.

The parser goes into a loop, in which it consumes tokens from the lexical analyser until it finds the closing brace (`"}"`) of that part of the `@if`-then-`@else` statement. The `skipToClosingBrace()` operation of the `ConfigParser` class implements that logic.

That approach offers the benefit of being trivial to implement. Unfortunately, it has a significant flaw: it allows syntax errors to go undetected. An example of this is shown in the configuration file below.

```
@if (osType() == "unix") {
    greeting == "Hello, UNIX user"; # syntax error
} @else {
    greeting = "Hello, Windows user";
}
```

That file contains a syntax error: the first assignment statement mistakenly uses `"=="` (the equality testing operator) instead of `"="` (an assignment operator). The Config4* parser will *not* report that error if the configuration file is used on Windows. It is only when a person later uses the configuration file on a UNIX machine that the syntax error will be reported.

Could Config4* be modified to use a more intelligent way of ignoring the non-executed part of an `@if`-then-`@else` statement? Probably, but I suspect it would involve significant changes to the parser, and those changes would introduce a lot of complexity.

Part II

Areas for Improvement

Chapter 7

Missing Components

7.1 Introduction

A mature software project includes more than just code. It also contains other components. In this chapter, I list some of those "other components" that currently are missing from, or inadequate in, Config4*. Perhaps some readers will be willing to contribute to Config4* by rectifying these deficiencies.

7.2 Cross-platform Build System

The C++ build system uses Gnu `make` and has been tested on just two platforms: Linux and Cygwin, both using the Gnu C++ compiler. In addition, the build system produces only statically-linked libraries. It would be nice to produce shared libraries too.

The Java build system uses Apache `ant`, which provides better portability than Makefiles. However, there is probably room for improvement in the `build.xml` file.

7.3 Javadoc and Doxygen Documentation

The *Config4* C++ API Guide* and *Config4* Java API Guide* document the API (that is, *application programming interface*) of Config4*. However, that documentation is arranged in logical groupings, and sometimes

it is more convenient to have API documentation that is arranged as an alphabetical list of class names and operation names.

For this reason, it would be useful for Javadoc- or Doxygen-style comments to be added into the source code. It is likely that the law of diminishing returns will apply: it is most important to provide this documentation for the publicly-accessible classes and operations, and less important to provide it for the internal classes and operations.

7.4 Installation Packages

The usability of Config4* would be enhanced if it could be distributed in a range of popular installation package formats, such as `".rpm"`, `".deb"` and InstallAnywhere.

7.5 Regression Test Suite

Why does Config4Cpp not have a comprehensive regression test suite? This is because I started writing my first configuration parser (which would eventually mature to be Config4Cpp) before Kent Beck popularised the development of unit testing frameworks.[1] Thus, it was easy for me to get into the bad habit of not having unit tests that I could use for regression testing and, unfortunately, that bad habit has survived.

Unit testing frameworks were widely available by the time I wrote Config4J, so why does Config4J not have a comprehensive regression test suite? This is because I wrote the initial version of Config4J by making a copy of Config4Cpp's source code and then spending two repetitive strain injury-inducing weeks converting C++ syntax into Java syntax. It seemed easier to do just that conversion than to do the conversion and also write unit tests.

Currently, Config4Cpp and Config4J *do* have a regression test suite, but it is only for schema types. This is in the `tests/schema-types` directory of an installation.

[1] http://en.wikipedia.org/wiki/XUnit

Chapter 8

Rethinking the Architecture

8.1 Introduction

It would be great if the complete list of features in Config4* had formed in my head during, say, a weekend, and then I had spent several months designing and implementing them. But, as I explained in Chapter 2, Config4* started life with minimal functionality (just assignment statements and scopes), and slowly acquired extra features over the course of almost 15 years before its first public release. Occasionally, this resulted in me discovering that the existing architecture of Config4* was not flexible enough to elegantly support a new feature that I decided to add. In some cases, I redesigned Config4* to better accommodate the new feature. In some other cases, I added the new feature with an inelegant hack.

In this chapter, I discuss some of the Config4* features that might benefit from a better design.

8.2 Parsing `@if`-then-`@else` statements

Section 6.5 on page 29 discusses a bug in the parsing of `@if`-then-`@else` statements. In summary, the parser uses a simplistic algorithm to ignore a non-executed "then" or "else" branch: it discards lexical tokens until it finds the closing `"}"` of the branch. The problem with this algorithm is

that syntax errors in a non-executed branch can go undetected, and this violates the *fail fast* principle [Sho04].

8.3 Location Information in Error Messages

Section 3.5.2 on page 13 explains why some error messages from Config4* do not specify the line number and file name of the source of the error. It would be good to find a memory-efficient and reasonably simple way to overcome that limitation.

8.4 Uid- entries

I introduced the `"uid-"` prefix fairly late in the development of Config4*. For many years before I introduced this feature, I had occasionally been frustrated at the need to define unique names for similar identifiers, for example, `employee_1`, `employee_2`, `employee_3`, and the need to write code that could identify and process such entries in a well-defined order.

Then one day, I started to write the *Comparison with other Technologies* chapter of the *Config4* Getting Started Guide*. In that chapter, I planned to explain how Config4* was clearly superior to other configuration technologies, including Java properties, the Windows Registry and XML. However, as I was writing the section on XML, I realised that Config4* was *not* clearly superior. It was better than XML in several ways, but XML had one important feature that was lacking in Config4*: the ability for a file to contain several distinct entries that have the same name. This focused my attention enough for me to brainstorm on a way to add similar functionality to Config4*. The result was that I added the `"uid-"` prefix to the syntax of Config4*, and I enhanced the API with some new operations to support that new feature.

The `"uid-"` prefix provides a solution to a particular problem. The problem is certainly an important one, but I am not sure that the `"uid-"` prefix is the *best* solution to that problem. I say that for two reasons.

- I have had only limited opportunities to use the `"uid-"` prefix in anger, so I do not yet have sufficient experience to feel confident that it will stand the test of time.

- Second, I designed the `"uid-"` prefix *partly* to address a frustration I had been having, and *partly* as a "marketing tick-box feature" so that Config4* could hold its own in comparisons with XML. As

many technically-minded people know, technical decisions made for marketing reasons are often suboptimal.

Despite the above concerns, I have been reasonably happy with the `"uid-"` prefix so far. However, perhaps there is a still-to-be-discovered mechanism that is better and more elegant than the `"uid-"` prefix.

8.5 Alternative Schema Validators

There are several reasons why I like the Config4* schema validator. It provides a lot of useful functionality in a relatively small amount of code. The schema language is intuitive. And the API of the `SchemaValidator` class is trivial to use.

Having said all that, the schema language has limitations. Perhaps somebody will be able to enhance the schema language with new features. Or perhaps somebody will develop a competing schema language for use with Config4*. If you wish to take the latter approach, then you might find the following information useful.

For the most part, the schema validator interacts with Config4* using just its public interface. The only exception to this is that the schema validator's lexical analyser (the `SchemaLex` class) inherits from the `LexBase` class, which currently is *not* part of Config4*'s documented API. There is no compelling reason for `LexBase` to remain an undocumented implementation detail. It is that way due to laziness on my part. So, if you want to experiment with writing your own schema validator and you would like to inherit from `LexBase`, then you have two options.

One option is to implement your class in the same namespace/package as Config4*. I think that might be good for prototyping purposes, but it runs the risk of the namespace/package growing to an unmanageable size, especially if several people each implement their own competing schema validators.

The other option is to *first* refactor code and write documentation to make `LexBase` a first-class citizen of the documented API of Config4*, and *then* write a schema validator (in a new namespace/package) that makes use of it.

8.6 Drawback of an Abstract Base Class

Consider the following scenario. Fred is a software developer working for the Acme company. He plans to use Config4J in several company-

internal applications. These applications require the ability to retrieve email addresses and dates from configuration files. Fred decides to write a class called, say, `AcmeConfiguration`, that inherits from `Configuration` and adds the required lookup-style operations. Once this has been done, Fred can then use the `AcmeConfiguration` class in the applications he wants to implement.

Unfortunately, Fred *cannot* implement the `AcmeConfiguration` class simply by inheriting from the `Configuration` class and adding a few `lookup<Type>()` operations. This is because the `Configuration` class contains approximately 90 `abstract` operations. The `AcmeConfiguration` subclass will have to provide an implementation for each of those. The implementation of those inherited `abstract` operations can be achieved through delegation, as illustrated in Figure 8.1.

This approach can work, but it involves a lot of tedious delegation code, which Fred would prefer to not have to write. I can think of three options to remove that burden from Fred.

The first option is for the Config4* library to contain a class called, say, `InheritableConfiguration` that implements the required delegation-based infrastructure. Then Fred could implement his `AcmeConfiguration` class by inheriting from `InheritableConfiguration` and adding *just* his new functionality.

The second option is to recombine the hidden implementation details of the `ConfigurationImpl` class with the publicly visible API of the `Configuration` class.

Those two options seem obvious, but I decided to not implement either one; at least not yet. This is because I suspect that: (1) I am not the first person to have encountered this issue; and (2) the options may have some subtle ramifications I have not thought of. I will wait until I gain a bit more experience in this area (or other people share their experience with me) before deciding which option to implement.

The third option, which I currently recommend, is discussed near the end of the *The `SchemaValidator` and `SchemaType` Classes* chapter in the *Config4* Java API Guide*. In brief, if Fred wants to provide lookup operations for email addresses and dates, then he should also consider providing schema support for those data types. He could then define the lookup operations on the `SchemaType<Type>` classes.

Figure 8.1: Pseudocode of a `Configuration` subclass

```
package com.acme.common;
import org.config4j.Configuration;
import java.util.Date;

public class AcmeConfiguration extends Configuration
{
    private Configuration    cfg; // delegation object

    public AcmeConfiguration()
    { cfg = Configuration.create(); }

    //--------
    // Implement all the inherited operations through delegation
    //--------
    public String lookupString(String scope, String name)
    { return cfg.lookupString(scope, name); }

    public String lookupString(String scope, String name,
                               String defaultValue)
    { return cfg.lookupString(scope, name, defaultValue); }

    ... // likewise for all the other inherited operations

    //--------
    // Now add new operations
    //--------
    public Date lookupDate(String scope, String name)
    { ... }
    public Date lookupDate(String scope, String name,
                           String defaultValue)
    { ... }
    public String lookupEmailAddress(String scope, String name)
    { ... }
    public String lookupEmailAddress(String scope, String name,
                                     String defaultValue)
    { ... }

}
```

Chapter 9

Other Programming Languages

9.1 Introduction

I hope that, over time, people will volunteer to implement Config4* for numerous programming languages. If that happens, then there will be several important benefits.

First, no matter what programming language a person decides to use for an application, there is a good chance that he or she can use Config4* rather than having to use an inferior configuration technology.

Second, it is common for different parts of a large project to be implemented in different languages. For example, in a client-server system, server applications might be implemented in C++, client applications in Java, and administration utilities in Perl. Since those applications are all part of the same overall project, there is a good chance that they will all need the ability to parse a common configuration file.

Third, the `"uid-"` prefix promotes Config4* from being "merely" a configuration language to being a file format suitable for data exchange. Having implementations of Config4* in many programming languages will make it possible for a program implemented in one language to exchange data with programs implemented in other languages.

This chapter summarises some of the issues that are likely to arise in implementing Config4* in a wide variety of programming languages.

9.2 Scripting Languages

Let's assume an implementation of Config4* requires about 10,000 lines of code (excluding comments and blank lines). What's a good way to implement Config4* for popular scripting languages, such as Lua, Perl, PHP, Python, Ruby and Tcl?

One way is to write 10,000 lines of Lua code to implement Config4Lua, then write another 10,000 lines of code to implement Config4Perl, and so on. However, this approach suffers from two problems. First, many scripting languages are interpreted, so Config4* implemented directly in a scripting language will be relatively slow. Second, 10,000 lines of code in each of several scripting languages quickly adds up to be a *lot* of code to write and maintain.

A better way is based on the fact that many popular scripting languages (including those mentioned at the start of this section) are implemented in C and can be extended with new functionality written in C. Thus, a good first step would be to write 10,000 lines of C code to implement Config4C. Then, a few hundred lines of extra code could be written to provide a Lua extension "wrapper" around the Config4C library, another few hundred lines of code could be written to provide a Perl extension "wrapper" around the Config4C library, and so on. This approach would provide a compiled—and hence fast—implementation of Config4* for scripting languages. It would also significantly reduce the amount of code, and hence effort, required to add Config4* to scripting languages.

9.3 Advice for Implementers

If you want to implement Config4* for another programming language, then I suggest you take the following steps.

Start by reading all the Config4* documentation. In particular, make sure you understand the information in this manual (the *Config4* Maintenance Guide*).

Then make a copy of the source code of Config4Cpp or Config4J, and start hacking at the copy to translate its syntax into that of your target programming language. Doing this translation will be tedious—I know this, because I used this approach to develop Config4J from Config4Cpp—but it offers two benefits. First, it will help to familiarise you with the architecture and source code of Config4*. Second, this

translation approach is much faster than developing a new implementation of Config4* from scratch.

Chapter 10

Internationalisation

10.1 Introduction

One aspect of internationalisation is the ability of an application to accept input in multiple human languages, such as English, Greek and Japanese. Nowadays, that ability is usually achieved by providing support for Unicode in the application. Having read some books on Unicode, I have come to two conclusions. First, the Unicode standard has some rough edges that can be irritating. Second, and more frustratingly, Unicode is not implemented widely enough in programming languages. Both of these issues affect Config4*, as I explain in this chapter. However, I expect that some readers may have a poor understanding of Unicode, so I will start by giving an overview of its concepts and terminology.

10.2 Unicode Concepts and Terminology

Unicode 1.0 was defined as a 16-bit character set. This meant it could represent a maximum of $2^{16} = 65,536$ characters. In Unicode terminology, a *code point* is a number that denotes a character within a character set. For example, in the ASCII character set, code point 65 denotes the character 'A'. Thus, Unicode 1.0 had $2^{16} = 65,536$ code points.

The designers of Unicode 1.0 estimated that supporting all the living languages in the world would require about 16,000 code points, so the 16-bit limit of Unicode 1.0 seemed to be more than sufficient. However, within a few years, they realised that their estimate was too low. That,

combined with an emerging desire for Unicode to be able to support ancient languages such as Egyptian Hieroglyphs, meant Unicode had to extend beyond 16-bits.

To accommodate the additional code points (and allow room for future ones), Unicode 2.0 was defined to be a 21-bit character set. Of course, since a 21-bit word size is uncommon in computers, Unicode is normally thought of as being a 32-bit character set (the high-end 11 bits are unused). You might think this means that Unicode can now support a maximum of $2^{21} = 2,097,152$ code points. However, some technical details in the Unicode specification mean that parts of the number range are unusable, so Unicode is able to support (only) 1,114,112 code points. Currently (as of Unicode 5.2), 107,361 of these code points have been assigned, so there is still significant room for future expansion.

10.2.1 Planes and Surrogate Pairs

If you want to store a collection of all the 1,114,112 code points in Unicode 2.0, then you could use a single-dimensional array of that size. However, another possibility is to use a two-dimensional array, because $17 \times 65,536 = 1,114,112$. When the Unicode Consortium were extending Unicode beyond 16 bits, it decided to use such a two-dimensional representation. In Unicode terminology, the range of code points is spread across 17 *planes*, where each plane consists of $2^{16} = 65,536$ code points.

The 17 planes are numbered 0..16. Plane 0 contains the 2^{16} code points from Unicode 1.0. To enable Unicode 2.0 to expand beyond the 16 bits of Unicode 1.0, 16 code points within plane 0 were reserved for use as escape codes. This makes it possible to represent a code point in plane N with two 16-bit words: the first word specifies the escape code for plane N, and the second word specifies an index into that plane. In Unicode terminology, such a two-word pair is called a *surrogate pair*; the escape code is called the *high surrogate*, and the following word is called the *low surrogate*.

10.2.2 UCS-2, UTF-8, UTF-16 and UTF-32

A surrogate pair is one way to *encode* a 21-bit Unicode code point, and that encoding format is known as UTF-16.

UCS-2 refers to the "16-bits fixed size" encoding used in Unicode 1.0. Many people mistakenly think that UCS-2 and UTF-16 are the same. The difference between them is subtle: UTF-16 supports surrogate pairs

(thus making it possible to support 21-bit code points), while UCS-2 does *not* support surrogate pairs.

Another Unicode encoding format is UTF-32, which, as its name suggests, encodes a 21-bit code point as a 32-bit integer (the highest 11 bits are unused). Obviously, UTF-32 is a trivial encoding format.

Yet another Unicode encoding format is UTF-8. This uses one byte to encode code points from 0..127, and uses multi-byte escape sequences to encode higher code points. The details of this encoding format are outside the scope of this discussion. The main point to note is that UTF-8 uses between 1 and 4 bytes to encode a code point; the higher the code point, the more bytes are required to encode it.

10.2.3 Merits of Different Encodings

There is no "obviously best" encoding for Unicode. Instead, each encoding has benefits and drawbacks.

UTF-32. The main benefit of this encoding is the convenience for programmers of knowing that a codepoint is always represented in a fixed-size amount of memory (a 4-byte integer). Because of this, programmers do not have to worry about correctly handling surrogate pairs (in UTF-16) or multi-byte escape sequences (in UTF-8).

The main drawback is the amount of RAM or disk space required to store UTF-32 strings. Some people hold the viewpoint that RAM and disk space are getting exponentially cheaper, so concerns about space inefficiency will gradually reduce over time. I partially agree with that sentiment. However, although the *capacity* of disk drives increases rapidly from year to year, the *bandwidth* available for transferring files to/from a disk (or across a network) rises more slowly. Because of this, it is beneficial to use a compact encoding when storing Unicode text in files or transferring them across a network.

UTF-16. All the code points required to support the majority of the world's living languages are contained in plane 0. Notable exceptions include Chinese, Japanese and Korean (often abbreviated to CJK). Plane 0 does not encode all the ideographs used in CJK, but it encodes *most* of the commonly used ones. Because of this, surrogate pairs tend to be used infrequently in most UTF-16 strings. Thus, one significant benefit of UTF-16 is that strings encoded in

it usually require about half as much space as strings encoded in UTF-32.

Another benefit of UTF-16 is that writing code to deal with the possibility of surrogate pairs is easier than writing code to deal with the possibility of multi-byte escape sequences (in UTF-8).

Because of the above two benefits, UTF-16 is commonly perceived as providing a better "size versus complexity" trade-off than either UTF-8 or UTF-32.

UTF-8. A string encoded in UTF-8 is guaranteed to be no longer than (and is typically much shorter than) the same string encoded in UTF-32. The same is not true when comparing UTF-8 to UTF-16.

Whether a string encoded in UTF-8 consumes *less memory* or *more memory* than the same string encoded in UTF-16 depends on the language used in the string. For example, UTF-8 usually requires just one or two bytes to encode a character used in a Western language, but two or three bytes to encode a character used in an Eastern language.

Despite this uncertainty of the space efficiency of UTF-8 versus UTF-16, UTF-8 is commonly *perceived* as being the most space-efficient encoding of Unicode.

Another benefit of UTF-8 is that it works well with byte-oriented networking protocols.

A drawback of UTF-8 is the complexity involved in writing code that correctly deals with multi-byte escape sequences.

UCS-2 It is best to avoid UCS-2 when writing new applications. Some programmers working with UTF-16 write code that does not handle surrogate pairs. In effect, this means that their applications can handle only UCS-2 rather than UTF-16. Sometimes, such programmers will claim that this limitation is acceptable because (they mistakenly believe that) plane 0 encodes *all* the characters of *all* the world's living languages, and they do not feel it is important for their applications to support, ancient languages, such as Egyptian Hieroglyphs. However, that assumption about plane 0 is incorrect: some living languages contain characters that are encoded outside plane 0.

10.2.4 Transcoding

If a programming language supports Unicode, then it is likely that the language provides native support for *one of* UTF-8, UTF-16 or UTF-32.[1] It is common for the programming language to provide utility functions for converting between its native Unicode encoding and other character set encodings.

The term *transcoding* is commonly used to refer to the conversion of a string from one character-set encoding to another. In some programming languages with Unicode support, transcoding takes place automatically during file input/output.

For example, when an application reads a text file, the file contents are transcoded from the character set specified by the computer's locale into the programming language's internal Unicode format. The application then processes the text in the Unicode encoding. Finally, when the application writes the text back out to file, the text is automatically transcoded from the programming language's Unicode format back into the character-set encoding specified by the computer's locale.

Programming languages that automatically transcode during file input/output try to achieve the best of both worlds: they provide a programmer-friendly encoding (typically UTF-16 or UTF-32) to manipulate strings in RAM, and a space-efficient encoding (perhaps UTF-8) when transferring to/from disk or across a network.

10.3 Unicode Support in Java

Java has always supported Unicode through its 16-bit `char` type. The first version of Java was released in January 1996, which was during the final months of Unicode 1.x. Because of this, Java supported UCS-2 initially.

Version 2.0 of Unicode, which defined UTF-16 and UTF-32, was released in July 1996. However, Java continued to support just UCS-2 for another eight years. Java 5.0, released in September 2004, finally upgraded Java's Unicode support from UCS-2 to UTF-16. To provide support for UTF-16, the `Character` and `String` classes were extended with new operations to identify surrogate pairs, to convert a surrogate

[1] An exception is the D programming language, which provides distinct data-types for *each of* UTF-8, UTF-16 and UTF-32 (http://en.wikipedia.org/wiki/D_%28programming_language%29#String_handling).

pair into a 32-bit code point, and to manipulate code points. For example, the `Character.isLetter()` operation is now overloaded to take either a 16-bit value (a Java `char`) or a 32-bit code point.

The main place in Config4J's source code where Unicode support arises is the lexical analyser. In particular, the lexical analyser calls `Character.isLetter()` to help it determine if a character is part of an identifier. There are two obvious ways to handle this in the lexical analyser.

Approach 1. The lexical analyser could ignore the possibility of surrogate pairs. Doing this would mean that Config4J could be compiled with relatively new compilers (Java 5.0 and later) and also with older compilers. However, by failing to correctly handle surrogate pairs, Config4J would be restricted to working with UCS-2 rather than UTF-16.

Approach 2. The lexical analyser could make direct use of operations (introduced in Java 5.0) that support surrogate pairs and 32-bit code points. Doing this would enable Config4J to support UTF-16, but would make it impossible for people to compile Config4J with older (pre-Java 5.0) compilers.

There is another, but non-obvious, way for the lexical analyser to handle Unicode issues.

Approach 3. The lexical analyser could use reflection to determine if the surrogate pair- and code point-related operations of Java 5.0 are available. If those operation are available, then the lexical analyser would use reflection to invoke them, and thus Config4J would support UTF-16. Conversely, if those operations are not available, then the lexical analyser would *not* attempt to invoke them, and hence Config4J would gracefully degrade to supporting UCS-2. This approach would offer the best of the two previous approaches: Config4J could be compiled with both old and new compilers, and it would support UTF-16 if the Java runtime environment does. There are two minor drawbacks to this approach.

First, invoking operations via reflection is more complex than invoking them directly. Thus, the lexical analyser would be harder to write and maintain. However, the use of reflection would be *very* localised, so the complexity introduced would be minimal.

Second, invoking operations via reflection is slower than invoking them directly. However, most of the Java 5.0-specific operations re-

quired are trivial enough to be reverse engineered and implemented inside Config4J, so they could be invoked without the need to use reflection. Doing that would ensure that the performance overhead of using reflection would be incurred *only when* a surrogate pair was encountered, which is likely to be very infrequently.

Currently, Config4J uses approach 1. I would like to see Config4J enhanced to support approach 3. I have not implemented approach 3 yet due to a combination of reasons. First, I wanted to release a "good enough" initial version of Config4J and defer improvements for a later release (rather than defer an initial release until Config4J was perfect). Second, I prefer to not write code for working with surrogate pairs unless I can test that code properly and, unfortunately, at the moment I do not have a good way to create, say, CJK-based configuration files that can be used for testing.

10.4 Unicode Support in C and C++

Ideally, I would like Config4Cpp to have the following properties.

- Not be limited to working with 8-bit characters, such as those in, say, English, but rather support the use of characters in arbitrary languages. Since much of the world is converging on Unicode for such support, Config4Cpp should support Unicode.
- Be portable across different C++ compilers and different operating systems.
- Rely on only the standard C library.

In this section, I explain the challenges that make achieving all the above very difficult, if not impossible.

10.4.1 Limitations in the Standard C Library

UTF-8 is an example of a *multi-byte* character encoding: it uses one or more bytes to encode each character.

UCS-2 and UTF-32 are examples of *wide* character encodings: they use fixed-size integers (16 or 32 bits) to represent each character.

UTF-16 is a bit unusual. Its support for surrogate pairs means that it is not a wide (that is, fixed-size) character encoding. Likewise, it is

not a multi-byte encoding since its basic unit is a 16-bit word rather than an 8-bit byte.

The C and C++ programming languages define the `char` type, which is usually associated with single-byte character encodings, such as ASCII or the ISO-Latin-*N* family of encodings. However, the `char` type can also be used with multi-byte encodings, such as UTF-8.

C and C++ also define the `wchar_t` type, which is for use with wide character encodings.[2] The C and C++ language specifications do *not* define the size of `wchar_t`; the specifications merely state that `wchar_t` is wide enough to hold all code points in the wide character encodings supported by the compiler and its runtime libraries.

- The width of `wchar_t` might be as little as 8 bits. That statement might seem like an oxymoron, but it makes sense if you consider a compiler that is developed for use with an embedded system. Such systems typically have a limited amount of RAM and may not have a requirement to support internationalisation. In such a scenario, the `wchar_t` type and its supporting functions might be implemented as placebo wrappers around the `char` type and its supporting functions.

- If the width of `wchar_t` is 16 bits, then this will be sufficient for UCS-2 and some non-Unicode encodings.

- If the width of `wchar_t` is 32 bits, then this will be sufficient for UTF-32 and some non-Unicode encodings.

An important point is that UTF-16 *cannot* be supported by a 16-bit wide `wchar_t`; at least, not without resorting to third-party (and probably proprietary) functions for dealing with surrogate pairs. This is important because Microsoft Windows (mis)uses a 16-bit wide `wchar_t` type with UTF-16. Thus, if you are writing Unicode-aware applications on Windows, then you are forced to use functions outside of the standard C library to deal with surrogate pairs. This can make it difficult to write Unicode-aware applications that are portable between Windows and UNIX-based operating systems, most of which use a 32-bit wide `wchar_t` type. One way to maintain cross-platform portability is to not attempt to deal with surrogate pairs; this will result in your application supporting UTF-32 on UNIX but only UCS-2 on Windows.

[2] In C, `wchar_t` is a `typedef` name (defined in `<stddef.h>`) for an integral type, while in C++ `wchar_t` is a keyword.

Another portability problem is that a 32-bit wide `wchar_t` type and its supporting functions might use UTF-32, *or* they might use a non-Unicode encoding. For example, whether or not UTF-32 is used by `wchar_t` on Solaris depends on a user-specified locale setting, and incorrectly assuming that UTF-32 is always used can result in application bugs.[3]

10.4.2 Use of Third-party Unicode Libraries

Having an operating system or programming language provide built-in support for Unicode is desirable, but it is not strictly necessary. This is because there are third-party libraries (both open- and closed-source) that provide Unicode support. However, these libraries tend to be many MB in size. For example, the C/C++ version of the ICU[4] library occupies about 22MB when built for Linux. Libraries that implement Unicode are large because they provide a lot of functionality that is driven by large tables. For example:

- The library must provide a set of properties for each Unicode code point. The properties include the official human-readable name for the code point and its category (an upper-case letter, a lower-case letter, a digit, a punctuation character, and so on). If it is an upper-case letter, then the code point of the corresponding lower-case letter is stored (and vice versa); that information is required to implement utility functions such as `toUpper()` and `toLower()`.

- The library might also provide tables to support transcoding (see Section 10.2.4 on page 51) between Unicode and other character set encodings. For example, ICU provides over 300 transcoding tables.

Config4Cpp could be modified so that it uses ICU (or some other Unicode library). Doing that would provide Config4Cpp with portable Unicode support. However, Currently, the Config4Cpp library occupies a few hundred KB. Modifying Config4Cpp to use ICU would add an extra 22MB to its memory footprint. That increase in required memory may be acceptable on many desktop and server machines, but it would make Config4Cpp too heavyweight for use in embedded systems.

[3] http://defect.opensolaris.org/bz/show_bug.cgi?id=11076#c14

[4] ICU (*International Components for Unicode*) is an open-source library, available in C/C++ and Java flavours. It is hosted at http://site.icu-project.org/.

10.4.3 UTF-8, UTF-16 or UTF-32?

As a thought experiment, let's assume we decide to modify Config4Cpp to support UTF-32. Obviously, the public API of Config4Cpp will have to change. For example, the signatures of the `insert<type>()` and `lookup<type>()` operations will change to accept UTF-32 strings rather than 8-bit strings (and those UTF-32 strings would then be stored in the internal hash tables).

The UTF-32 version of Config4Cpp will be convenient for programmers who are developing applications that use UTF-32. However, it will not be convenient for programmers who work with UTF-16 or UTF-8. Such programmers will have to frequently transcode (that is, convert) between UTF-32 strings (used by Config4Cpp) and UTF-8 or UTF-16 strings (used by other parts of their applications). The details of how to transcode between UTF-8, UTF-16 and UTF-32 can be found easily with an Internet search, so implementing such transcoding functions will not be difficult. However, there are two problems. First, littering application code with calls to those transcoding functions will decrease code readability and maintainability. Second, repeated transcoding will impose a performance overhead.

Those two problems (decreased readability and a performance overhead) are not unique to using UTF-32. Those same problems will arise whenever different parts of an application use different character encodings. So, it doesn't really matter if Config4Cpp uses UTF-8, UTF-16, UTF-32 or a locale-specified encoding: the encoding used by Config4Cpp will be convenient for *some* application developers, and inconvenient for others.

The lack of an "obviously right" Unicode encoding choice affects C/C++ applications because those language specifications do not define built-in support for Unicode. This issue does not arise in, say, Java, because the designers of Java chose to standardise on a specific Unicode encoding.

10.4.4 Approach Currently Used in Config4Cpp

I do not have sufficient Unicode experience to be able to make an informed decision about which character encoding would provide the "lesser of all evils" for use in Config4Cpp. Because of this, I decided to use only the features available in the standard C library. This works as follows.

- All the operations in Config4Cpp's public API work with C-style

strings, that is, `char*`. Config4Cpp assumes those strings are encoded according to the locale in effect, which might be a single-byte encoding (for example, ASCII or ISO-Latin-*N*) or a multi-byte encoding (for example, UTF-8).

- The standard C library provides functions, such as `isalpha()` and `isdigit()` to determine the category of a `char`. Those functions work reliably for a single-byte encoding. However, to deal with the possibility of a multi-byte encoding, it is necessary to use `mbstowcs()` to transcode the stream of `char` into a stream of `wchar_t` and then use `iswalpha()` and `iswdigit()` to check the category of a character. That approach is used by Config4Cpp's lexical analyser so that it can correctly identify the characters permitted in identifiers.

- Once the lexical analyser has identified a (potentially multi-byte) character's category, it discards the `wchar_t`. The *name=value* entries in Config4Cpp's hash tables are stored as C-style strings, that is, as null-terminated arrays of `char`. The encoding used in those strings is the encoding specified by the locale in effect.

The approach described above is convenient for application developers who work with C-style strings. It is also convenient for developers who *know* that the locale uses the UTF-8 encoding. Developers who prefer, say, UTF-16 or UTF-32 will have to transcode between the locale's encoding and their preferred encoding.

I do *not* claim that this approach is ideal. Rather, I view it as a temporary measure until a better approach can be determined. In particular, I hope that the open-sourcing of Config4* will result in internationalisation experts within the open source community offering advice on how to improve this aspect of Config4Cpp.

Chapter 11

Localisation

11.1 Introduction

Currently, Config4* does *not* support localisation. In particular, error messages embedded in exceptions are hard-coded in English. Config4* would be more user-friendly for non-English speakers if such error messages could be provided in the language specified by the locale in effect.

11.2 One Possible Approach for Localisation

I do not have any experience of localising software applications. From the little I have read on the subject, I have formed the following (possibly incorrect) assumptions about how localisation might be introduced to Config4Cpp.

First, we need to (somehow) add Unicode support to Config4*. In particular, we need: (1) the ability to store a collection of strings in a Unicode encoding, that is, one of UTF-8, UTF-16 or UTF-32; and (2) the ability to transcode those strings into the character encoding of the current locale.

Second, we need a data store that contains parameterised error messages (written as Unicode strings) in multiple languages. If we provide Unicode support in Config4*, then it might be possible to use an embedded configuration file as the data store.[1] As an example, an error

[1] Utilities such as `config2cpp-nocheck` or `Config2JNoCheck` (Section 3.6 on page 17) could be used to embed the configuration file in the Config4* library.

messages configuration file might be in the format shown in Figure 11.1.

Figure 11.1: Hypothetical error messages localisation file

```
#--------
# Parameterised error messages in English
#--------
en {
  1 = "%%s1, line %%i1: expecting ';' near '%%s2'";
  2 = "%%s1, line %%i1: expecting '=', '?=' or '{' near '%%s2'";
  ...
}

#--------
# Parameterised error messages in French
#--------
fr {
  1 = "%%s1, ligne %%i1: j'attendais ';' près de '%%s2'";
  2 = "%%s1, ligne %%i1: j'attendais '=', '?=' ou '{' près de '%%s2'";
  ...
}
... # Scopes for other languages
```

The parameterised text for, say, error 2 in the current locale would be obtained as follows:

```
language = ...; # extract the language code from the locale
errMsg = cfg.lookupString(language, "2");
```

Then all the string place holders (such as `"%s1"` and `"%s2"`) and all the integer place holders (such as `"%i1"`) in the parameterised error message would be replaced with values from an array of strings and an array of integers. The resulting string could then be used as a localised error message when throwing an exception.

An interesting aspect of the above proposal is that Config4* would store localised messages in Config4* format. Unless care was taken, this might introduce a bootstrapping problem in building Config4*. Obviously, that is a drawback of the proposal. A benefit of the proposal is that Config4* would not rely on a third-party localisation library. Avoiding reliance on third-party libraries can keep down the memory requirements of Config4*.

Bibliography

[Ray99] Eric S. Raymond. *The Cathedral & the Bazaar*. O'Reilly, 1999. www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

[Sho04] Jim Shore. Fail fast. *IEEE Software*, pages 21–25, Sept/Oct 2004. www.martinfowler.com/ieeeSoftware/failFast.pdf.