Practical Usage Guide

Version 1.2 30 September 2021

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

Contents

Chapter 1

Introduction

1.1 The Purpose of the Manual

Sometimes, when a new programming/programmable technology is released, it can take developers several years to figure out how they can exploit the technology to best effect.

For example, let's assume that Sony or Nintendo release a new video game console that is significantly more powerful than the current generation of game consoles. The games released soon after the launch of this new games console may well be very enjoyable, but they are unlikely to push the console to the limits of its capabilities. It is likely to be the games that are released a few years after the console's launch that will fully exploit its capabilities.

By default, one might expect that developers will initially use Config4* in simple ways—for example, to process a handful of variables in a runtime configuration file—and only after several months or years will developers figure out how to use Config4* in more adventurous ways that exploit its full capabilities.

This manual provides some shortcuts on that road to enlightenment. As I explain in the *History* chapter of the *Config4* Maintenance Guide*, the maturing of Config4* from its inception to its first public release took place over almost 15 years. During those years, I used prototype versions of Config4* in personal projects. This extensive use of Config4* has given me insights into how Config4* can be used in non-trivial ways. In this manual, I share many of those insights, so readers can learn to properly exploit Config4* sooner rather than later.

1.2 Structure of this Manual

The chapters in this manual are grouped into three parts.

The chapters in Part I provide examples of relatively straightforward ways to use Config4* for a wide variety of purposes.

Part II discusses issues associated with using a configuration file to specify details for the creation and initialisation of objects.

Part III provides a case study of how Config4* is used in Config4JMS, which is a library that simplifies use of the Java Message Service (JMS).

Part I

Straightforward Uses of Config4*

Introduction to Part I

The chapters in Part I provide examples of relatively straightforward ways to use Config4*. If there is any logic to the sequencing of chapters here, it is that I have arranged them in approximate order of increasing complexity. But, in general, each chapter in Part I is self-contained. This makes it feasible to read the chapters out-of-sequence, or to read a single chapter that matches your interests and ignore the rest.

Chapter 2

Migrating from Another File Format

2.1 Introduction: Description of Problem

Consider the following scenario. Your company has been selling an application for the past five years and, up until now, that application has used, say, an XML file to store its configuration information. You are designing an updated version of the application, and you would like to switch from using XML to Config4* for its configuration file. There is just one problem: you need to find a way for users to be able to automatically convert their existing XML-based configuration data into Config4* format. How can you achieve this goal?

2.2 Solution

An easy way to help users migrate from, say, an XML file format to Config4* is to write a program that does the following:

1. Call `Configuration.create()` to create an empty configuration object.
2. Parse an input XML file and store the result in a DOM tree.
3. Traverse the DOM tree and, for each *name-value* pair you encounter, call `cfg.insertString()` or `cfg.insertList()` to add the *name-value* pair into the configuration object.

4. When you have finished traversing the DOM tree, call `cfg.dump()` to get a textual representation of all the configuration information, and write this to a configuration file.

That's all that is required.

Chapter 3

Preferences for a GUI Application

3.1 Introduction: GUI Preferences

Many applications expose their configuration files to users. If a user wishes to reconfigure such an application, then he or she uses a text editor to modify its configuration file. That approach is common for applications that do *not* have a graphical user interface (GUI). However, in GUI-based applications, it is common for configuration changes to be made through the GUI itself rather than through an external text editor.

For example, when using a GUI application on Microsoft Windows, you can typically use the *Tools* → *Options...* menu item to open a tabbed dialogue box that enables you to modify the application's configuration. The equivalent menu item in GUI applications running on UNIX-based operating systems is often *Edit* → *Preferences*.

3.2 Persisting Preferences

When you make changes to the "preferences" or "options" of a GUI-based application, the application saves those changes to a (hidden) configuration file, so that if you quit the application and restart it, then the application can reload the most recent set of configuration values. A GUI application on Microsoft Windows typically uses the Windows Registry as its "hidden configuration file". A GUI application running on UNIX

might use a Java properties file or an XML file for the same purpose.

3.3 Using Config4* to Persist Preferences

You may be wondering if, when developing a GUI application, you could use Config4* to store the application's preferences/options. Unfortunately, I do not know the answer to that. This is because I do not have much experience with developing GUI applications, and hence cannot offer knowledgeable advice.

I suspect that most GUI applications are built with the aid of a framework library that simplifies the development of such applications. For all I know, such framework libraries might automate the saving and loading of preferences/options data. If that is the case, then those framework libraries are probably programmed to use, say, the Windows Registry or an XML file. If this assumption is correct, then it will probably be easier for you to use that provided functionality rather than try to modify the framework library (or work around it) to use a Config4* file instead.

However, perhaps some readers will be building a GUI application *without* the aid of such a framework library. Or perhaps some readers want to implement a new framework library that simplifies the development of GUI applications. If such readers want to consider using Config4*, then I offer the following advice:

- Read a Config4* configuration file in the usual way, that is, by creating an empty `Configuration` object and then calling `parse()`. However, also use fallback configuration so that a GUI application can work "out of the box".

- When the user modifies some of the preferences/options via the GUI, call `cfg.insertString()` or `cfg.insertList()` to insert (or update) the corresponding entries in the configuration object. Then call `cfg.dump()` to get a textual representation of all the configuration information, and write this to the application's hidden configuration file.

- You might want to read the discussion of the `dump` command in the *The `config4cpp` and `config4j` Utilities* chapter of the *Config4* Getting Started Guide*. That discussion explains why reading a Config4* file and then dumping it back again does *not* preserve adaptable configuration that had been present in the original file.

Keep that limitation in mind when designing your framework library or GUI application.

Chapter 4

Code Generation

4.1 Introduction

There is a wide variety of domain-specific code generation tools that can generate repetitive code via a user-written template or script file. Often, such tools could be made more flexible by equipping them with a configuration-file parser. In this chapter, I illustrate this by describing a code generation tool that I developed.

Before describing the code generation tool, I first need to provide some background information.

4.2 Overview of CORBA IDL

There are many competing technologies that can be used to build client-server applications. One such technology, CORBA, is a standard for an object-oriented version of remote procedural calls.

One feature central to CORBA is the interface definition language (IDL). An IDL file serves a purpose similar to a Java `interface` or a C++ header file: it defines a public interface. More specifically, an IDL file defines the public interface(s) of a server application.

IDL provides primitive types such as `boolean`, `short`, `long`, `float`, `char` and `string`. IDL also provides constructed types, including `struct` (similar to a C `struct`), a `union` (similar to a variant record in Pascal), and `sequence` (roughly similar to a `std::vector` in C++ or an `ArrayList` in Java). All these types can be used as parameters to operations defined

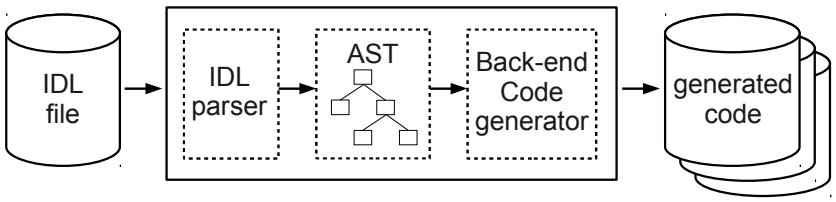
in an `interface`.

Each CORBA product provides an IDL compiler that translates the types defined in an IDL file into corresponding types in a programming language (most commonly C++ or Java, but some CORBA products support other languages, such as C, Ada, SmallTalk, Cobol, PL/I, LISP, or Python). Thus, for example, a C++ programmer can manipulate an IDL `struct` through its C++ representation, while a Java programmer can manipulate it through its Java representation. An IDL `interface` is translated into two types in a programming language: a client-side *proxy* class and a server-side *skeleton* class. When a client wants to invoke an operation on a remote object in a server process, the client invokes the operation on a local proxy object, which marshals the invocation request into a binary buffer, sends that buffer across the network to the server application, and waits for the reply. A server-side skeleton object unmarshals the incoming request, dispatches it to the target object, then marshals the reply and transmits it across the network to the client.

4.3 Architecture of an IDL Compiler

The architecture of a typical IDL compiler is shown in Figure 4.1. A parser analyses an input IDL file, performs semantic checks, and builds an in-memory representation, called an abstract syntax tree (AST), of what it has parsed.

Figure 4.1: Architecture of an IDL compiler



When parsing is finished, control is then passed to the back-end code generator. The code generator traverses the AST (perhaps several times) and uses print statements to generate code. By the way, that high-level architecture is not unique to IDL compilers; compilers for a great many languages are likely to share a broadly similar architecture.

4.4 Repetitive Application-level Code

In 1995, I started working in the consultancy and training department of a CORBA vendor (IONA Technologies), When working on consultancy assignments, I noticed it was common for CORBA server applications to contain significant amounts of repetitive code. For example, let's suppose you want to put a CORBA server "wrapper" around a legacy system. If the legacy system has, say, 50 public operations, then you might define one or more CORBA IDL interfaces that, between them, contain a similar number of IDL operations. When you start to implement the CORBA server, you will quickly notice that each operation is implemented in a similar manner:

- Perform data-type translation to convert each input parameter from its IDL type to the corresponding legacy type.
- Then call the legacy operation that corresponds to the IDL operation.
- Finally, perform data-type translation to convert each output parameter from its legacy type to the corresponding IDL type.

It is not uncommon for there to be many thousands (or even tens of thousands) of lines of repetitive code in such server applications.

Writing thousands of lines of repetitive code by hand is error-prone and can quickly become a maintenance nightmare. I decided to design a code-generation tool that could automate the generation of such repetitive code.

4.5 Architecture of `idlgen`

The code-generation tool I developed became known as `idlgen`, which was a contraction of *IDL code generator*. The high-level architecture for `idlgen` is shown in Figure 4.2.

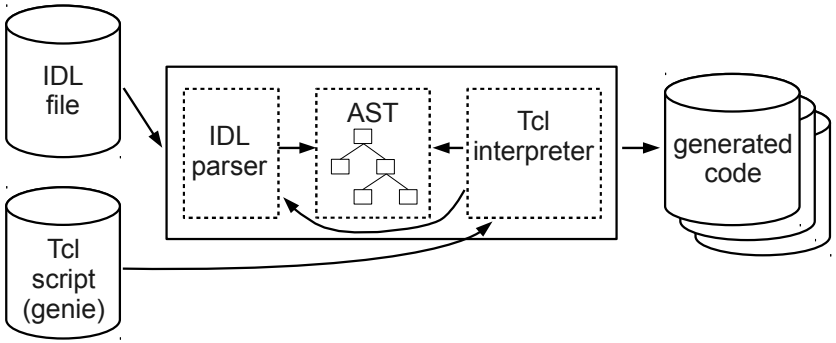
If you compare Figures 4.1 and 4.2, you will notice that `idlgen` has an architecture very similar to that of an IDL compiler. However, there are a few differences, as I now discuss.

The biggest difference is that `idlgen` replaces the fixed back-end code generator with an interpreter for a scripting language called Tcl.[1] Doing

[1] Here are some details for interested readers. The Tcl interpreter is packaged as a library of C functions, and the library contains a hash table that provides a mapping from the name of a command to a C function that implements the command. You can extend the Tcl interpreter with a new command called, say, `foo`, by writing a C function that implements the desired functionality, and then registering that C function with the name `foo` in the Tcl interpreter's hash table. Once you have done that, the (now extended) Tcl interpreter can execute scripts that contain the `foo` command. When implementing `idlgen`, I used this technique to put a Tcl wrapper around the parser, and Tcl wrappers around each node in the AST produced by the parser.

I chose Tcl because, at the time, it was one of the few scripting languages that had been designed to be extensible. Since then, scripting languages that are designed (or retrofitted) to be extensible are more common. If I were designing `idlgen` today, then I might be tempted to use Python or Lua instead because they are arguably better scripting languages and have a less unusual syntax.

Figure 4.2: Architecture of `idlgen`



this makes it possible to write a back-end code generator as a Tcl script. The term *genie* is used to refer to a "code **gen**$_{\mathbf{i}}$**e**ration script".

Another, but more minor, difference is that a genie is not "just" a back-end code generator to which control is passed after an input file has been successfully parsed. Instead, a genie (also) acts as the mainline of the code generator. It is the genie that calls the parser to parse an input IDL file. When parsing is complete, control returns to the genie, which can then traverse the AST and generate code. Allowing the genie to control the mainline provides some flexibility. For example, each genie can decide what command-line options it will support.

4.6 Benefits of Code Generation

I found that, when writing back-end code generators in Tcl, I was up to 100 times more productive than writing back-end code generators in

C++. Obviously, a Tcl-based code generator was not as fast as one implemented in C++, but it was usually "fast enough". For example, a genie would typically generate C++ code about five or ten times faster than a C++ compiler could compile that generated code. Thus, the relative slowness of a code generator implemented in an interpreted scripting language was never a bottleneck in application development.

In many of my consultancy assignments, I realised that the customer's project would require significant amounts of repetitive code. In such cases, I might spend two or three weeks writing a project-specific genie that contained, say, 3000 lines of Tcl, and that genie would then generate the tens (or even hundreds) of thousands of lines of repetitive code required for the project.

My aim is not to engage in self-praise on the merits of `idlgen`. Rather, I simply offer it as an example of how, when a code generator is the right tool for the job, then it can significantly reduce the effort involved. I assume that many other domain-specific code generators provide similarly significant increases in productivity.

4.7 Using Configuration in Code Generation

You may be wondering, "What has `idlgen` got to do with Config4*?" The answer is that `idlgen` has *two* built-in parsers: one for IDL, and another for a predecessor of Config4*.[2] The presence of this configuration parser greatly enhances the flexibility, and hence power, of `idlgen`. I explain why through the following example.

In Section 4.4, I explained how a project that puts a CORBA server wrapper around a legacy system might require a significant amount of repetitive code. Let's assume you are working on such a project, and the IDL file you write for the CORBA server is similar to that shown in Figure 4.3.

When the server application starts, it initially creates one `Factory` object and one `Administration` object. An administration client connects to the `Administration` object and invokes an operation to get information about the server's status, or to ask the server to gracefully shutdown. Other client applications connect to the `Factory` object and invoke `create_foo()` or `create_bar()`, which results in the server creating

[2]This predecessor supports a very limited subset of Config4*'s syntax: only *name=value* (where the *value* could be a string or a list of strings), scopes and an include command.

Figure 4.3: Example IDL file

```
interface Foo {
    void op1(...);
    void op2(...);
    ...
    void op20(...);
    void destroy();
};
interface Bar {
    void op21(...);
    void op22(...);
    ...
    void op50(...);
    void destroy();
};
interface Factory {
    Foo create_foo(...);
    Bar create_bar(...);
};
interface Administration {
    string get_server_status();
    void shutdown_server();
};
```

a new `Foo` or `Bar` object on behalf of the client. (The "separate object for each client" approach might be for, say, auditing or security purposes). Then the client can invoke some of the operations on the newly created object. Between them, the `Foo` and `Bar` interfaces contain 50 operations (denoted as `op1..op50` in the IDL file) that wrap correspondingly named operations in the legacy system. When the client is finished, it invokes `destroy()` to destroy the `Foo` or `Bar` object that the server had previously created for it.

When writing a genie to generate repetitive code for the project, you can code the genie so it parses not just an IDL file, but also a configuration file, such as that shown in in Figure 4.4.

Syntactically, the `interface_type` variable is a list, but its contents are arranged as a two-column table that maps the name of an IDL interface into a "type": either *singleton* (meaning the server process contains only only instance of the specified interface) or *dynamic* (meaning that a create-style operation is used to create instances of this interface dynamically). Knowing this "type" of each IDL interface makes it possible for

Figure 4.4: Configuration file for a genie

```
interface_type = [
  # name               singleton/dynamic
  #-------------------------------------
   "Factory",         "singleton",
   "Administration",  "singleton",
   "Foo",             "dynamic",
   "Bar",             "dynamic",
];
operation_type = [
  # wildcarded name      type
  #--------------------------------------
   "Factory::create_*",  "create",
   "*::destroy",         "destroy",
   "Administration::*",  "hand-written",
   "Foo::op1",           "query",
   "Foo::op3",           "query",
   "Foo::op8",           "query",
   "Foo::op10",          "query",
   "Foo::*",             "update",
   "Bar::op27",          "query",
   "Bar::op32",          "query",
   "Bar::op33",          "hand-written",
   "Bar::op30",          "query",
   "Bar::*",             "update",
];
code_segment_files = [
    "some-code-segments.txt",
    "more-code-segments.txt",
];
```

the genie to generate a `main()` function that creates one instance of each singleton interface (and does *not* create instances of dynamic interfaces).

The `operation_type` table specifies a "type" for each operation in all the IDL interfaces. To keep this table short, the operation name can contain `"*"`, which is a wildcard that matches zero or more characters. For example, `"*::destroy"` matches both `Foo::destroy` and `Bar::destroy` (IDL uses `"::"` as the scoping operator). This enables the genie to use a cascading if-then-else statement to decide what kind of code to generate for the implementation of each IDL operation, as shown in the following pseudocode:[3]

[3]Some scripting languages make it possible to use polymorphism (instead of a cascading if-then-else statement) to call one of several procedures. This technique can result in shorter, easier-to-maintain code. For example, the Tcl syntax is: `generate_${opType}_operation $op`. However, I have shown a cascading if-then-else statement for simplicity of discussion.


```
foreach op in anInterface.listOperations() {
    opType = op.getOperationType();
    if (opType == "create") {
        generate_create_operation(op);
    } else if (opType == "destroy") {
        generate_destroy_operation(op);
    } else if (opType == "query") {
        generate_query_operation(op);
    } else if (opType == "update") {
        generate_update_operation(op);
    } else if (opType == "hand-written") {
        opName = op.getFullyScopedName();
        codeSegmentName = "implementation of " + opName + "()";
        print(getCodeSegment(codeSegmentName));
    } else {
        error("unknown operation type: " + opType);
    }
}
```

The assumption in the above pseudocode is that most create-style operations can be implemented using one kind of repetitive code, most destroy-style operations can be implemented using a second kind of repetitive code, and so on. There may be some operations that require hand-written code. If a code generation tool uses a general-purpose scripting language, then it should be possible to write a parser for what I call "code segment" files. Such a file contains a collection of named code segments. The syntax used in such a file is not important for the discussion at hand, but Figure 4.5 shows one possible format.

The `code_segments_files` variable in the configuration file specifies a list of code segment files. The genie can parse those files so that the cascading if-then-else statement shown earlier can copy-and-paste a code segment into the generated code for each operation that must be hand-written.

If a code generation tool does *not* enable scripts to parse a configuration file to access information such as that shown in Figure 4.4, then code generation scripts tend to be limited to generating the same type of repetitive code for every interface and every operation. But when, as is the case with `idlgen`, the code generator provides a parser for configura-

Figure 4.5: Example of a code segments file

```
START: implementation of Bar::op33()
... // code that implements Bar::op33()
END: implementation of Bar::op33()

START: implementation of Administration::get_server_status()
... // code that implements Administration::get_server_status()
END: implementation of Administration::get_server_status()

START: implementation of Administration::shutdown_server()
... // code that implements Administration::shutdown_server()
END: implementation of Administration::shutdown_server()
```

tion files, code-generation scripts can become significantly more flexible. In my experience, scripts can become capable of generating *all* the repetitive code required for a project, rather than just a small subset of the code or "starting point" code that a programmer must then modify.

4.8 Comparison with Annotations in Java 5

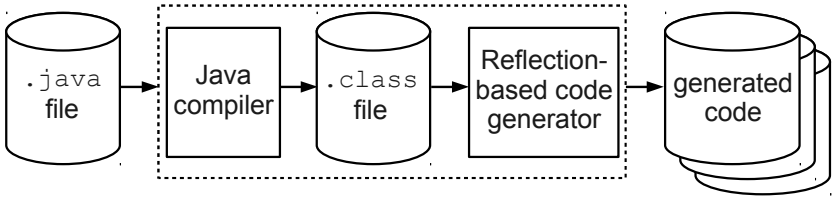
I have explained how it can be useful for a code generation tool to have access to two types of information about data-types: (1) information provided by, say, an AST; and (2) extra information provided in a configuration file, such as that shown in Figure 4.4 on page 19. Java 5 provides broadly similar functionality, as I now discuss.

A Java compiler converts a `".java"` file into a `".class"` file that contains bytecode *plus* type information. The type information in a `".class"` file is conceptually similar to the information available in the AST of a compiler. Java's reflection API makes it possible to navigate the type information stored in `".class"` files. This makes it possible to use the reflection API to implement a code-generation tool. You can see this easily by comparing Figure 4.1 on page 14 with Figure 4.6.

Version 5 of Java added several new features to the language, one of which is support for *annotations*. I will briefly describe the syntax and use of annotations, and then explain their relevance to code generation.

A commonly-encountered difficulty in adding a new feature to an existing programming language is that the new feature might require a new keyword, but introducing a new keyword would break existing programs that already use its spelling in identifiers. A (potentially ugly) solution to this problem is to overload the semantics of an *existing* keyword so it

Figure 4.6: Reflection-based code generation in Java



can be used for the new language feature. That was the approach taken by the designers of Java when adding annotations to the language. Java has always used the keyword `interface` to define an interface. Now, with Java 5, you can use `@interface` (the `"@"` symbol in front of the keyword is not a typo) to define an annotation. As an example, here is the definition of an annotation called `@CodeGenType`:

```
@interface CodeGenType {
    String type();
}
```

The above defines an annotation, called `@CodeGenType`, that takes a `String` parameter called `type`.

Once an annotation type has been defined, you can instantiate the annotation at the start of the declaration of, say, an interface, class or operation. You can see examples of this in Figure 4.7.

Figure 4.7: Example use of Java annotations

```
@CodeGenType(type = "dynamic") interface Foo
{
    @CodeGenType(type = "query") void op1(...);
    @CodeGenType(type = "update") void op2(...);
    ...
    @CodeGenType(type = "update") void op20(...);
    @CodeGenType(type = "destroy") void destroy();
};
```

Instantiating an annotation at the start of a declaration associates the instantiated annotation with the item being declared. The Java compiler can write details of instantiated annotations into the generated `".class"` file. Then, a code generation tool like that shown previously in Figure 4.6 can use Java reflection to access the instantiated annotations associated with declarations.

Look again at the code-generation configuration file in Figure 4.4 on page 19. In particular, notice that the `interface_type` table specifies that `Foo` is a `"dynamic"` interface. That configuration information is reproduced by the Java annotation on the `Foo` interface in Figure 4.7. Likewise, the type information specified for operations of `Foo` in the `operation_type` table in Figure 4.4 is reproduced by Java annotations in Figure 4.7. Thus, we see that metadata for code generation can come from either a configuration file or from annotations embedded in an input file. Both approaches seem to offer similar functionality.

Is it a good idea for a programmer to put metadata about a program in source-code files (as is the case with Java annotations)? Or should the programmer put the metadata in a separate file (such as a configuration or XML file)?

I have written many `idlgen` genies that parsed configuration files to obtain metadata about IDL interfaces. In those genies, the metadata indicated important characteristics about the type of code that should be generated. In other words, the metadata specified high-level implementation details. Implementation details do *not* belong in the specification of an interface. Therefore, it was entirely proper for the metadata to be written somewhere other than in IDL files.

Just as (metadata about) implementation details do not belong in the definition of a CORBA IDL `interface`, I feel that such metadata do not belong in an Java `interface` either. However, Java annotations are often used in a `class` rather than in an `interface`. And since a `class` *does* contain implementation details, it seems reasonable for annotations to appear there.

I do not hold a strong view about whether it is best to store metadata in source-code files or in separate configuration files. Perhaps the decision should be made on a case-by-case basis. If so, then it would be useful for future language designers to equip their languages with: (1) something akin to Java annotations that can be embedded in source-code files; and (2) a standardised way to store metadata in, say, a configuration file. In this way, programmers could mix-and-match the two approaches in whatever way best suits their needs.

Chapter 5

Server Applications

5.1 Introduction

It seems intrinsic to the nature of developing client-server applications that tedious, repetitive code has to be written—especially in server applications. For example, server applications are often required to execute not just "business logic" code, but also "infrastructure logic" code for every incoming request: to perform security checks, validate input parameters, log input and output parameters, and so on.

There are many competing technologies for developing client-server applications. Some of those technologies provide ways to automate commonly required, server-side "infrastructure logic", while other technologies require programmers to manually write such code.

In this chapter, I explain how Config4* can be used to reduce the burden of writing some types of "infrastructure logic" code.

5.2 Validation Checks for Parameters

Consider a client-server application in which the client presents a form for the user to fill in, and then sends details from the filled-in form to the server for processing. The server should validate the input data before it tries to process it. Doing this can involve a lot of tedious, repetitive code, as you can see in Figure 5.1.

The pseudocode shown for the `placeOrder()` operation makes two validation checks on the `customerName` parameter, in both cases throwing a descriptive exception if the check fails. If all the parameters to `placeOrder()` require a similar level of validation checking, then the programmer will have to write and maintain several dozen lines of validation code for just a single operation. And if the server's public interface has many operations, then it is easy to imagine the server containing many hundreds or even several thousands of lines of validation code.

Figure 5.1: Manual validation of parameters can be tedious

```
void placeOrder(
    String    customerName,
    String[]  shippingAddress,
    Float     cost,
    String    creditCardNumber
    String    discountCode) throws ValidationException
{
    if (customerName == null) {
        throw new ValidationException("You must specify a "
                                + "value for customerName");
    }
    if (customerName.length() > 40) {
        throw new ValidationException("The value of "
                                + "customerName is too long");
    }
    ... // validation checks for the other parameters

    ... // business logic code
}
```

A better approach is to use Config4* to define a simple schema language for describing the validation checks to be performed on parameters. A hypothetical example of this is shown in Figure 5.2.

The configuration file contains a scope for each operation in the public interface of the server. Within such a scope, there are variables corresponding to each parameter of the operation. The value of a variable is a list of *name=value* strings that specify the validation checks to be performed on the parameter when the operation is invoked. Figure 5.3 shows the outline of a `Validator` class that can perform the validation checks described in the configuration file shown in Figure 5.2.

The `Validator` class provides a `validate()` operation that is overloaded for parameters of different types. It might require, say, 500–1000 lines of code to implement this class, but that class needs to be implemented just once and then it can be reused to perform parameter

Figure 5.2: Configuration for validation rules

```
placeOrder {
  customerName = ["mandatory=true", "maxLength=40"];
  shippingAddress = ["mandatory=true", "minSize=3", "maxSize=5"];
  shippingAddress-item = ["mandatory=true", "maxLength=60"];
  cost = ["mandatory=true", "min=10"];
  discountCode = ["mandatory=false", "maxLength=10"];
  creditCardNumber = ["mandatory=true", "fixedSize=16",
                      "pattern=[0-9]*"];
}

updateOrder {
  #--------
  # Most parameters are similar to those in placeOrder(), so...
  #--------
  @copyFrom "placeOrder";
  ... # now add/modify validation rules as required
}

cancelOrder {
  ...
}
```

validation for many different operations in a single server. Perhaps the class could be reused across several related projects. Thus, the effort required to implement the `Validator` class can be repaid easily if use of the class significantly reduces the amount of parameter-validation code required in server operations. Figure 5.4 shows the intended use of the `Validator` class.

During initialisation, the server parses a configuration file containing the parameter-validation rules (like those shown in Figure 5.2). As the pseudocode in Figure 5.4 shows, this configuration file could be embedded in the application via use of the `config2cpp` or `config2j` utility. Then, the body of each operation in the server can validate all its parameters in a concise way: it creates a `Validator` object and calls `validate()` once for each parameter. In this way, the amount of validation code required in an operation with N parameters can be reduced from, say, $8N$ lines of code (as illustrated by the validation checks for `customerName` in Figure 5.1 on page 26) to just $N + 1$ lines of code (as illustrated in Figure 5.4).

If the public API of the server is defined in, say, CORBA IDL, and

Figure 5.3: A `Validator` class

```
public class Validator {
    private Configuration  cfg;
    private String         opName;

    public Validator(Configuration cfg, String opName)
    {
        this.cfg = cfg;
        this.opName = opName;
    }

    public void validate(String value, String paramName)
                                        throws ValidationException
    {
        String[] constraints = cfg.lookupList(opName, paramName);
        //--------
        // Iterate over "constraints" and throw an exception if
        // "value" violates any of them.
        //--------
        ...
    }

    public void validate(String[] value, String paramName)
                                        throws ValidationException
    { ... }

    public void validate(Float value, String paramName)
                                        throws ValidationException
    { ... }
}
```

you have access to a code generation tool, for example, `idlgen`, then it is possible to reduce the amount of hand-written validation code even further. You can do this by writing a genie that generates a `Util` class containing utility methods that encapsulate the $N+1$ lines of validation code for each public operation of the server. By doing this, the code of `placeOrder()` can be reduced to that shown in Figure 5.5: parameter validation is achieved by delegating to the (generated) utility operation `Util.validatePlaceOrder()`.

To briefly summarise, in this section I have shown a two-step approach to significantly reduce the amount of parameter validation code

Figure 5.4: Example use of the `Validator` class

```
//--------
// The following code is executed during server initialisation.
//--------
validationCfg = Configuration.create();
validationCfg.parse(Configuration.INPUT_STRING,
                    embeddedValidationConfig.getString());

//--------
// This is an example of how to perform parameter-validation
// in an operation.
//--------
void placeOrder(
    String   customerName,
    String[] shippingAddress,
    Float    cost,
    String   creditCardNumber
    String   discountCode) throws ValidationException
{
    Validator v = new Validator(validationCfg, "placeOrder");
    v.validate(customerName, "customerName");
    v.validate(shippingAddress, "shippingAddress");
    v.validate(cost, "cost");
    v.validate(creditCardNumber, "creditCardNumber");
    v.validate(discountCode, "discountCode");
    ... // business logic code
}
```

Figure 5.5: Encapsulating calls to `validate()` in utility functions

```
void placeOrder(
    String   customerName,
    String[] shippingAddress,
    Float    cost,
    String   creditCardNumber
    String   discountCode) throws ValidationException
{
    Util.validatePlaceOrder(customerName, shippingAddress, cost,
                            creditCardNumber, discountCode);
    ... // business logic code
}
```


that needs to be embedded in server-side operations.

The first step is to write a `Validator` class (Figure 5.3 on page 28) that can perform parameter validation checks based on information in a configuration file (Figure 5.2). By doing this, the amount of validation code required in an operation with N parameters can be reduced from, say, $8N$ lines of code (as illustrated by the validation checks for `customerName` in Figure 5.1 on page 26) to just $N + 1$ lines of code (as illustrated in Figure 5.4).

The second step is to write a genie that generates a `Util` class containing utility methods that encapsulate the $N + 1$ lines of validation code for each public operation of the server. By doing this, the parameter validation code embedded in each public operation can be reduced to a single line of code that delegates to a generated utility operation (Figure 5.5).

Ideally, those two steps would *not* need to be performed by an application developer. Instead, the `Validator` class and the genie would be provided by a vendor who sells tools for building client-server applications. If this were done, then an application developer might not even need to explicitly invoke `Util.validate<OperationName>()` from the body of a public operation. Instead, that invocation could be made from the dispatch logic generated by the vendor's tools for building client-server applications.

5.3 Dispatch Rules

Consider the following scenario. You are in charge of a team that is developing a client-server application. You decide to split your team into two sub-teams: one to develop the client application, and the other to develop the server application. In this way, you hope to get some development work done in parallel. However, it turns out that there is a lot more work required to develop the server application than to develop the client application. The client development team reach their first milestone fairly quickly; then they start to complain that they need a server to test their client against, but the server team have not yet finished their work. Can anything be done to help the client development team make progress?

The obvious solution is for the client development team to implement a test version of the server so they can test their client against it. Even if this test server offers simplistic functionality, it will at least permit

the client development team to test basic connectivity. A drawback of this approach is that the test server will have a short lifespan—it will be discarded when the real server application is mature enough to test against—so any work put into writing the test server will appear to be wasted effort. In this section, I describe an alternative approach; one in which Config4* plays a small but important role.

Let's assume the server application exposes an interface called `Foo` that defines 10 operations. The server team intend to write a class, called `FooImpl`, that implements that interfaces. The implementation of the operations in that class will contain the "business logic" code required in the server.

My suggestion is that, along with implementing the `FooImpl` class, the server should contain two other classes, as I now discuss.

The `FooTest` class implements the operations of the `Foo` interface, but with "test logic" rather than with "business logic". The test logic in an operation might be as simple as printing a message to say the operation was called and then return a dummy result. Or perhaps the test logic might be more complex. The choice is up to the client development team, since they will be writing this class and using it to test their client application.

The other class, `FooDispatch`, also implements the operations of the `Foo` interface. Each incoming request is executed by the `FooDispatch` class, and that class uses a `simulation_rules` configuration variable like that shown in Figure 5.6 to decide if it should delegate the request to the corresponding operation on the `FooImpl` or `FooTest` class.

Figure 5.6: Simulation rules

```
simulation_rules = [
    # wildcarded operation name   simulate?
    #-------------------------------------
     "Foo.op1",                  "true",
     "Foo.op3",                  "true",
     "*",                        "false",
];
```

The `simulation_rules` table maps a wildcarded string of the form *interface.operation* to a boolean value. Code to implement that mapping is provided by the `shouldSimulate()` operation of the `FooDispatch` class, which is shown in Figure 5.7.

The constructor of the `FooDispatch` class creates instances of both

Figure 5.7: Pseudo-code of the `FooDispatch` class

```
class FooDispatch implements Foo {
    private FooImpl  businessObj;
    private FooTest  testObj;
    private boolean  simulateOp1;
    private boolean  simulateOp2;
    ...
    private boolean  simulateOp10;

    public FooDispatch(String[] simulationRules)
    {
        businessObj = new FooImpl();
        testObj = new FooTest();
        simulateOp1 = shouldSimulate(simulationRules, "Foo.op1");
        simulateOp2 = shouldSimulate(simulationRules, "Foo.op2");
        ...
        simulateOp10 = shouldSimulate(simulationRules, "Foo.op10");
    }

    private boolean shouldSimulate(String[] rules, String opName)
    {
        for (int i = 0; i < rules.length; i += 2) {
            String pattern = rules[i + 0];
            String boolStr = rules[i + 1];
            if (Configuration.patternMatch(opName, pattern)) {
                return boolStr.equals("true");
            }
        }
        return false;
    }

    public long op1(...)
    {
        Foo targetObj = businessObj;
        if (simulateOp1) {
            targetObj = testObj;
        }
        return targetObj.op1(...);
    }
    ... // likewise for the other operations
};
```

`FooImpl` and `FooTest`, and stores those as instance variables. The constructor then calls `shouldSimulate()` to decide whether each operation should delegate to the corresponding operation on the `FooImpl` or `FooTest` object. For efficiency, these decisions are cached in boolean instance variables, rather than being recalculated for each invocation. The implementation of an operation uses a simple if-then-else statement to delegate the request to the "test" or "business" object.

Some readers may assume the approach outlined above is burdensome because it requires programmers to write three classes—`FooImpl`, `FooTest` and `FooDispatch`—for the server instead of just one (`FooImpl`). Doesn't doing this triple the effort required to write the server? Actually, no. The server development team have to write `FooImpl`, so that does not count as an extra burden. Likewise, if the client development team want a "test server" to test their client against while waiting for the "real" server to be mature enough to test against, then they would have to implement (something similar to) `FooTest`. As such, the need to write `FooTest` does not count as an extra burden either. The *only* extra burden is in writing `FooDispatch`, and that class is trivial enough to not be much of a burden.[1]

Using the above approach, project development can proceed as follows.

- The server development team start by writing the server mainline, `FooDispatch` and stubbed versions of both `FooTest` and `FooImpl`. These stubbed versions will do something very basic, such as printing a `"This operation was called"` diagnostic message. The important thing is to get this skeletal version of the server implemented as quickly as possible.

- The client development team start writing their application. When they first need to test against the work-in-progress server, they edit the `simulation_rules` table in a configuration file so that all operations delegate to the test logic. In testing, they will be able to see the `"This operation was called"` diagnostic messages.

- If the client development team need more complex test logic in the server, then they modify the `FooTest` class however they wish; then recompile and retest. With the client team modifying `FooTest` and the server team modifying `FooImpl`, there should be few, if any, conflicts, because the two teams are modifying different source-code files. Hence, development of the client and its "test" server can proceed in parallel with development of the "real" server.

[1] If you are building your client-server application using the Orbix implementation of CORBA, then you could write an `idlgen` genie to generate `FooDispatch` and an initial implementation of `FooTest`.

- Whenever the server team finishes implementing an operation, they can inform the client team. The client team can then modify the `simulation_rules` table in their configuration file so they can now test against the "business logic" implementation of that operation. If they discover a bug in that operation, they can inform the server team and modify the `simulation_rules` table to revert back to using the "test logic" version of that operation until the bug is fixed.

- When all the operations in the server have been implemented, the `simulation_rules` table can be modified so that all operations delegate to the business logic implementations.

- Before you put the server into production, you might decide to modify the `shouldSimulate()` operation in the `FooDispatch` class (see Figure 5.7) so that it is hard-coded to return *false*. In this way, you can guard against the possibility of misconfiguration resulting in the "test logic" implementations of operations being used in a production environment.

  Alternatively, if you are not concerned about the possibility of such misconfiguration, then you could deploy the server with the ability to execute either "business logic" or "test logic" implementations of operations. Obviously, the "business logic" implementations would be used in day-to-day operations, but being able to temporarily revert to using "test logic" implementations might be a useful troubleshooting aid for whenever something goes wrong.

There are many competing technologies available for building client-server applications. It is common for these technologies to provide a class that delegates an incoming request to the target object. Such classes always provide some "added value" when performing the delegation. For example, the class might unmarshal an incoming request before dispatching (that is, delegating) it. Or the class might perform auditing, security checks, or manage transaction boundaries when dispatching an incoming request. The "should I dispatch this request to the 'business logic' or 'test logic' implementation of an operation?" functionality *could*

be designed into the dispatch classes of future technologies for building client-server applications. If that ever happens, then it will remove the (albeit small) burden from application developers of writing code such as the `FooDispatch` class shown in Figure 5.7.

5.4 Summary

It is common for server applications to contain not just "business logic" but also "infrastructure" code. Some technologies for building client-server applications can reduce the burden of implementing some types of infrastructure task. For example, a client-server technology might automate security checks or transaction boundaries.

In this chapter, I have discussed two other types of infrastructure tasks that are unlikely to be automated by current client-server technologies: the validation of input parameters, and deciding whether an incoming request should be dispatched to the real "business logic" implementation of an operation or to a "test" implementation. I have explained how Config4* can simplify the implementation of both those tasks.

Chapter 6

Test Suites

6.1 Introduction

If you are writing software to automate the running of a test suite, then, as I discuss in this chapter, you may find some Config4* features useful.

6.2 Regression Test Suite

Over time, a software project is likely to acquire an ever-growing collection of tests to check that specific pieces of functionality work correctly. It can be useful for a project team to rerun its entire test suite each night, to check if newly added or modified code has broken existing code. In addition, when a developer modifies code in a particular subsystem of the project, it can be useful for him to be able to immediately run the subset of tests that are related to the modified subsystem, rather than wait for the nightly run of the entire test suite.

There are some programming language-specific framework libraries that simplify the task of writing and running tests,[1] but many projects develop their own bespoke testing frameworks.

A feature common to many testing frameworks, whether bespoke or not, is that each test has a unique name, and the testing framework *knows* the names of all the tests. For example:

- If each test is implemented by a separate function, then the name of the function acts as the name of the test, and the test framework contains a map that is (somehow) populated with *name-of-test* → *pointer-to-function* entries.
- If each test is implemented as a collection of files in a subdirectory, then the name of a subdirectory acts as the name of the test it contains, and the testing framework can obtain a list of the names of those subdirectories.

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

Several years ago, when I wrote a bespoke testing framework, I found it useful for the testing framework to obtain two list variables, `include_tests` and `exclude_tests`, from a configuration file. The testing framework iterated over the entire list of test names, and it executed a test only if: (1) its name matched a wildcarded pattern in the `include_tests` list; and (2) its name did *not* match any wildcarded patterns in the `exclude_tests` list. Pseudocode to check those conditions is provided in Figure 6.1.

Figure 6.1: Pseudocode of `shouldExecuteTest()`

```
boolean shouldExecuteTest(
    String[] includeTests,
    String[] excludeTests,
    String   testName)
{
    for (int i = 0; i < excludeTests.length; i++) {
        String pattern = excludeTests[i];
        if (Configuration.patternMatch(testName, pattern)) {
            return false;
        }
    }
    for (int i = 0; i < includeTests.length; i++) {
        String pattern = includeTests[i];
        if (Configuration.patternMatch(testName, pattern)) {
            return true;
        }
    }
    return false;
}
```

I found this approach to be simple and effective. For example, you can execute all tests with the following configuration:

```
include_tests = ["*"];
exclude_tests = [];
```

Perhaps you use the name of a software component as a prefix on the names of tests related to that component. If so, then you can execute all tests *except* those related to the *foo* and *bar* components by using the following configuration:

```
include_tests = ["*"];
exclude_tests = ["foo_*", "bar_*"];
```

Doing that might be useful if you know that the *foo* and *bar* components currently are unstable, but you want to test the project's remaining components.

As a final example, you can execute the tests for just the *foo* and *bar* components by using the following configuration:

```
include_tests = ["foo_*", "bar_*"];
exclude_tests = [];
```

6.3 Performance Test Suite

Figure 6.2 shows a (pseudocode) API of a server application that processes invoices. A client application (somehow) obtains details of invoices and then invokes `submitInvoices()` to send them, in batches, to the server.

Let's assume you are part of a project team that has been asked to implement that client-server system. Before committing to the project, your manager decides to carry out performance tests of various products that will be used in the project: the database product, the middleware product, and so on.[2] His goal is to determine if the performance targets for the project are feasible. In particular, he wants to determine this feasibility *before* he commits significant resources to implementing the project. Your manager has asked one member of his team to write a performance test for the database, and he has asked another member, you, to write a performance test for the middleware product. In particular, he wants you to find the answer to the following question: "How many invoices per second is the middleware product capable of transmitting from the client to the server?"

You will probably find it trivial to implement a server for your performance test. In particular, the implementation of the `submitInvoices()` operation does *nothing* because you are testing the performance of *just*

[2] *Middleware* is software that simplifies the building of client-server applications. CORBA, JMS and Web Services are examples of (competing) middleware standards.

Figure 6.2: API of an invoice processing server

```
struct InvoiceItem {
    long          productCode;
    float         quantity;
    float         price;
    String        description;
};
struct Invoice {
    String        customerName;
    String[]      billingAddress;
    String[]      shippingAddress;
    String        creditCardNumber;
    InvoiceItem[] items;
    float         totalPrice;
};
interface InvoiceProcessor {
    void submitInvoices(Invoice[] invoices);
    ... // other operations
};
```

the middleware product. However, writing the client for your performance test turns out to be a bit more interesting, as I now discuss.

A pseudocode outline of your client is shown in Figure 6.3. The client connects to the server. Then it initializes an array of invoices. Having done that, it starts a timer, invokes `submitInvoices()` one million times, stops the timer, and reports the average throughput, that is, the number of invoices sent to the server per second.

Figure 6.3: Pseudocode of a performance test

```
invoiceProcessorObj = ...; // connect to the server
Invoice[] invoices = ...;
numIterations = 1000 * 1000;
startTime = getCurrentTime();
for (int i = 0; i < numIterations; i++) {
    invoiceProcessorObj.submitInvoices(invoices);
}
endTime = getCurrentTime();
elapsedTime = endTime - startTime;
throughput = invoices.length * numIterations / elapsedTime;
print("Throughout is " + throughput + " invoices per second");
```

Common sense dictates that the throughput will depend on the size of each invoice, which, obviously, can vary. Perhaps you also suspect that the throughput will depend on the batch size, that is, the number of invoices sent in each call to `submitInvoices()`.[3] These issues mean that you will need to run the performance test multiple times, for different sizes of invoices and different batch sizes. And to be able to do that, you do not want to hard-code information about invoice size or batch size into the test client. Instead, you want that information to be configurable, which is where Config4* comes in useful.

If you are familiar with XML, then you may know that XPath is a syntax used to specify nodes in an XML document. We can borrow that concept, and apply it (albeit with a different syntax) to specify individual fields within a complex parameter that is passed to an operation. Consider the following examples:

```
InvoiceProcessor.submitInvoices.invoices
InvoiceProcessor.submitInvoices.invoices[2].customerName
InvoiceProcessor.submitInvoices.invoices[2].billingAddress[0]
```

The first line above specifies the `invoices` array parameter passed to the `submitInvoices()` operation in the `InvoiceProcessor` interface. The second line specifies the `customerName` field of the `invoices` array indexed by 2. And the third line specifies the first line of that invoice's billing address.

With that syntax in mind, now have a look at the `parameter_rules` configuration variable in the `test_10` scope of Figure 6.4.

The `parameter_rules` variable is arranged as a two-column table. The first column uses `"*"` as a wildcard character to reduce the verbosity of the syntax discussed above. The first line in the table specifies that the *length* of the `invoices` array is 10. The second line specifies that the *value* of all `customerName` fields is `"John Smith"`. In general, the table is used to specify the *length* of arrays, and either the *value* or *length* of strings.

The `parameter_rules` table provides an intuitive and flexible way to configure the size of parameters to an operation when running a performance test. The `test_20` and `test_30` scopes uses the concatenation operator (`"+"`) to reuse the value of `test_10.parameter_rules` but prefix it with a different *length* of the `invoices` array. If code that processes `parameter_rules` uses the first matching pattern found, then this prefixing provides a simple and concise way to run the performance test for different batch sizes.

[3] CPU speed, network latency and network bandwidth are also likely to affect throughput, but I ignore those issues in the following discussion.

Figure 6.4: Parameter rules for a performance test

```
test_10 {
  parameter_rules = [
    # wildcarded attribute name             attribute's value
    #-------------------------------------------------------------
      "*.invoices.length()",                 "10",
      "*.invoices[*].customerName.value()", "John Smith",
      "*Address.length()",                   "5",
      "*Address[0].value()",                 "29 Street name",
      "*Address[1].value()",                 "Name of suburb",
      "*Address[2].value()",                 "Reading",
      "*Address[3].value()",                 "Berkshire RG1 2LD",
      "*Address[4].value()",                 "United Kingdom",
      "*.creditCardNumber.length()",         "16",
      "*.invoices.items.length()",           "4",
      "*.description.length()",              "10",
  ];
}
test_20 {
  parameter_rules = ["*.invoices.length()", "20"]
                  + test_10.parameter_rules;
}
test_30 {
  parameter_rules = ["*.invoices.length()", "30"]
                  + test_10.parameter_rules;
}
```

The question to now ask is the following: How much effort is required to write code that can use `parameter_rules` to initialise the `invoices` parameter in the performance test?

You can find the answer to that question by looking at the pseudocode shown in Figure 6.5. For conciseness, the pseudocode assumes that the `parameter_rules` table has been read from the configuration file and is available in the `parameterRules` instance variable. The test client shown in Figure 6.3 on page 40 would initialise the `invoices` parameter with the following statement.

```
invoices = allocateInvoiceArray(
                    "InvoiceProcessor.submitInvoices.invoices");
```

Figure 6.5: Pseudocode to process `parameter_rules`

```
int getArrayLength(String name) {
    String nameDotLen = name + ".length()";
    for (int i = 0; i < parameterRules.length; i += 2) {
        String pattern   = parameterRules[i + 0];
        String attrValue = parameterRules[i + 1];
        if (Configuration.patternMatch(nameDotLen, pattern)) {
            return integer.parseInt(attrValue);
        }
    }
    return 0; // default value
}

String allocateString(String name) {
    String nameDotLen = name + ".length()";
    String nameDotVal = name + ".value()";
    for (int i = 0; i < parameterRules.length; i += 2) {
        String pattern   = parameterRules[i + 0];
        String attrValue = parameterRules[i + 1];
        if (Configuration.patternMatch(nameDotVal, pattern)) {
            return attrValue;
        } else if (Configuration.patternMatch(nameDotLen, pattern)) {
            int length = Integer.parseInt(attrValue);
            StringBuffer result = new StringBuffer();
            for (int j = 0; j < length; j++) {
                result.append("x");
            }
            return result.toString();
        }
    }
    return ""; // default value
}

String[] allocateStringArray(String name) {
    int length = getArrayLength(name);
    String[] result = new String[length];
    for (int i = 0; i < length; i++) {
        result[i] = allocateString(name + "[" + i + "]");
    }
    return result;
}
... continued on the next page
```


Figure 6.5 (continued): Pseudocode to process `parameter_rules`

```
... continued from the previous page
InvoiceItem allocateInvoiceItem(String name) {
    InvoiceItem result = new InvoiceItem();
    result.productCode = 0;
    result.quantity = 0;
    result.price = 0;
    result.description = allocateString(name + ".description");
}

InvoiceItem[] allocateInvoiceItemArray(String name) {
    int length = getArrayLength(name);
    InvoiceItem[] result = new InvoiceItem[length];
    for (int i = 0; i < length; i++) {
        result[i] = allocateInvoiceItem(name + "[" + i + "]");
    }
    return result;
}

Invoice allocateInvoice(String name) {
    Invoice result = new Invoice();
    result.customerName = allocateString(name + ".customerName");
    result.billingAddress = allocateStringArray(
                                name + ".billingAddress");
    result.shippingAddress = allocateStringArray(
                                name + ".shippingAddress");
    result.creditCardNumber = allocateString(
                                name + ".creditCardNumber");
    result.items = allocateInvoiceItemArray(name + ".items");
    result.totalPrice = 0;
}

Invoice[] allocateInvoiceArray(String name) {
    int length = getArrayLength(name);
    Invoice[] result = new Invoice[length];
    for (int i = 0; i < length; i++) {
        result[i] = allocateInvoice(name + "[" + i + "]");
    }
    return result;
}
```

Some readers may be discouraged by the verbose and repetitive nature of the pseudocode in Figure 6.5. Once you understand how the pseudocode works, then it becomes obvious that the verbosity will be proportional to the quantity and complexity of data-types used as parameters. However, it might be possible to use one of two techniques to eliminate the need to write such verbose, repetitive code.

First, perhaps the public API of the server is defined using a specification language (similar in spirit to CORBA IDL), and perhaps there is a code generation tool (similar in spirit to `idlgen`) for that specification language. If that is the case, then it should be straightforward to write a code generation tool to generate all the verbose, repetitive code.

Second, perhaps the programming language you are using to write the test client provides reflection capabilities. If so, then you could write a utility class that uses reflection to navigate over the strings, arrays, and nested structures contained within the parameter, and initialise each one.

Once you have found a viable technique for initialising parameters without having to manually write lots of repetitive code, you will discover that a simple performance test client like that shown in Figure 6.3 on page 40 can be very flexible.

6.4 Summary

In this chapter, I have discussed two ways in which Config4* can be useful for implementing test suites.

First, in a regression test suite, a configuration file might contain `include_tests` and `exclude_tests` variables that specify a list of wildcarded test names. This provides a simple yet effective way to specify an arbitrary subset of tests that should be run.

Second, writing a performance test suite is conceptually simple, but often time consuming due to the need to write repetitive code to initialise parameter values. Some people hard-code parameter values into a test program, but this results in an inflexible performance test. A more flexible approach is to use a configuration file to store wildcarded metadata about the sizes and values of parameters. Unfortunately, handwritten code to retrieve such metadata and use it to initialise parameters can be verbose and error-prone. However, if you have access to a code generation tool, then you could use it to generate such code. Alternatively, if your performance test suite is written in a language that provides a reflection API, then you could use this to write a utility function that

can initialise an arbitrary type of parameter from metadata in a configuration file.

Part II

Configuration-driven Object Creation

Introduction to Part II

Consider the configuration file below:

```
object_1 {
    type = "Person";
    name = "John Smith";
    age = "42";
}
object_2 {
    type = "Car";
    model = "Porsche 996 GT3";
    registration_number =  "R13 MEW";
}
```

It is possible to imagine an application that parses the above configuration file and iterates over all the `object_<int>` scopes. For each scope, the application creates an object of the type specified by the `type` variable within the scope, and sets instance variables of the newly created object to the values specified by variables within the scope.

Some readers may question the value of an application written in such a way. After all, it is probably syntactically shorter to just hard-code the creation of objects into the application:

```
obj1 = new Person();
obj1.setName("John Smith");
obj1.setAge(42);

obj2 = new Car();
obj2.setModel("Porsche 996 GT3");
obj2.setRegistrationNumber("R13 MEW");
```

The code can be even more concise if the values for instance variables are passed as parameters to constructors:


```
obj1 = new Person("John Smith", 42);
obj2 = new Car("Porsche 996 GT3", "R13 MEW");
```

Despite the relative verbosity of using a configuration file to specify the creation and initialisation of objects, this technique can offer important benefits for some niche programming tasks. Part II of this manual explores issues associated with using this technique in Config4*-based applications.

Chapter 7

Limitations of the "uid-" Prefix

7.1 Introduction

Let's assume we want to write an application that creates objects based on information provided in scopes in a configuration file:

```
object_1 {
    type = "Person"; # The type of object to create
    ... # Other name=value pairs specify values for instance variables
}
object_2 {
    type = "Car"; # The type of object to create
    ... # Other name=value pairs specify values for instance variables
}
```

When editing the configuration file to add more `object_<int>` scopes, it will be tedious to keep track of which numbers have already been used. To avoid this problem, we might decide to use the `"uid-"` prefix instead.

```
uid-object {
    type = "Person";
    ...
}
uid-object {
    type = "Car";
    ...
}
```


At first sight, this appears to be an improvement. However, the `"uid-"` prefix has some some subtle limitations that can affect the ergonomics of a configuration file. This chapter explores those limitations, and suggests an alternative approach that, sometimes, can work better.

To make the issues discussed in this chapter more concrete, I base the discussion around the design of a hypothetical application for monitoring security devices—such as burglar alarms and security cameras—that can be connected to a computer network.

Each scope in a configuration file specifies details for a security device: (1) its network address as a string of the form *ip-address:port*; and (2) a brief description of its physical location, for example, "front entrance" or "loading bay". When the application examines a configuration scope, it creates a "device driver" object based on information in the scope, and adds that object to a collection. To monitor the status of security devices, the application simply iterates over the collection and invokes an operation on each object that queries the status of the corresponding security device.

In this chapter, I discuss two different approaches that can be taken with Config4* to store details of each security device. The first approach illustrates limitations of the use of the `"uid-"` prefix. The second approach avoids using the `"uid-"` prefix and, in doing so, avoids its limitations.

7.2 Approach 1: With "uid-" Entries

Figure 7.1 shows how a configuration file might use the `"uid-"` prefix to store details of different types of security devices.

The format of the configuration file in Figure 7.1 is straightforward: there is an `uid-camera` scope for each camera on the network and, likewise, an `uid-burglar-alarm` scope for each burglar alarm on the network.

Let's assume that the security monitor application wants to display a warning message about a malfunctioning camera. Several options come to mind for the format of such a message.

The first option is for the message to identify the camera by reporting the expanded form of its `uid-camera` name, for example, `"Camera 'uid-000000042-camera' is malfunctioning"`. However, such a message is not user-friendly: a user would have to laboriously search through the configuration file for the 43nd occurrence of the `"uid-"` prefix to identify

Figure 7.1: Security configuration with "uid-" entries

```
uid-camera {
    network_address = "192.128.42.006:5000";
    location = "front entrance";
};
uid-camera {
    network_address = "192.128.42.009:5000";
    location = "loading bay";
};
uid-burglar-alarm {
    network_address = "192.128.42.021:5000";
    location = "loading bay";
}
```

the relevant camera.[1]

A second option is for the message to identify the camera by reporting one of its attributes, such as its location: "The camera at location 'front entrance' is malfunctioning". Such a message seems to be user-friendly, but it unambiguously identifies the relevant camera *only if* the location attribute has a unique value. This may not be the case if there are several cameras placed in the same location (perhaps for redundancy purposes, or perhaps to provide different views of the same location). If there are no existing attributes that are guaranteed to be unique, then you could introduce one, such as id shown in Figure 7.2.[2]

The need to introduce the id attribute in Figure 7.2 is ironic because the whole point of the "uid-" prefix is to *avoid* users having to invent unique identifiers. By introducing the id attribute, we reintroduce problems that the "uid-" prefix was intended to avoid. In particular, users must ensure that each id has a unique value. This may not be a significant problem if there are just a handful of scopes in a configuration file, but it can become a problem if a configuration file contains hundreds or thousands of scopes. This problem can be eased somewhat if the value of id can be an arbitrary string rather than, say, just an integer. In this case, a user might set a device's id to be its location suffixed by a number. For example, three devices at the front entrance might have id values: "front entrance: 1",

[1] The counter for uid entries starts at 0, so 42 is the 43rd occurrence of a uid entry.

[2] Actually, the network_address attribute is likely to be unique, but I will ignore that for the moment because I want to focus on what can be done if there are *not* any unique attributes.

Figure 7.2: Security configuration with id attributes

```
uid-camera {
    id = "1";
    network_address = "192.128.42.006:5000";
    location = "front entrance";
};
uid-camera {
    id = "2";
    network_address = "192.128.42.009:5000";
    location = "loading bay";
};
uid-burglar-alarm {
    id = "3";
    network_address = "192.128.42.021:5050";
    location = "loading bay";
}
```

"front entrance: 2" and "front entrance: 3", while two devices at the loading bay might have id values: "loading bay: 1" and "loading bay: 2". Such a convention can help reduce the difficulty of ensuring unique id values for each of hundreds of devices (assuming there are only a handful of devices at each location). If this approach is taken, then the application can produce messages of the form, "Camera 'front entrance: 2' is malfunctioning". This is more user-friendly than "Camera 'uid-000000042-camera' is malfunctioning".

The schema definition for the configuration file shown in Figure 7.2 is straightforward, and is shown in Figure 7.3.

Figure 7.3: Schema for the configuration file shown in Figure 7.2

```
String[] schema = new string[] {
    "uid-camera                 = scope",
    "uid-camera.id              = string",
    "uid-camera.network_address = string",
    "uid-camera.location        = string",

    "uid-burglar-alarm                 = scope",
    "uid-burglar-alarm.id              = string",
    "uid-burglar-alarm.network_address = string",
    "uid-burglar-alarm.location        = string"
};
```

It is important to note that the Config4* schema language does *not* provide any way to ensure that each `id` variable has a unique value. Because of this, a developer implementing the security-monitoring application would need to write code that checks for clashes in the values of the `uid-camera.id` variables. You can implement this code as follows. First, you create an empty hash table that will provide an `id` → `scope` mapping. Then, you populate this hash table by iterating over all the `uid-camera` scopes to obtain the value of the `id` variable within each scope. Before adding each `id` and `scope` to the hash table, you check if the hash table already contains an entry with the same `id` value. If it does, then you report the clash as an error.

7.3 Approach 2: Without "uid-" Entries

As I explained in Section 7.2, using the `"uid-"` prefix in the configuration file for the security-monitoring application was not as beneficial as we might have hoped: we *still* had to introduce an artificial `id` variable inside each scope. Since the `"uid-"` prefix is not providing as much benefit as we would like, we might decide to avoid its use altogether, as shown in Figure 7.4.

Figure 7.4: Security configuration without `id` attributes

```
camera {
    1 {
        network_address = "192.128.42.006:5000";
        location = "front entrance";
    }
    2 {
        network_address = "192.128.42.009:5000";
        location = "loading bay";
    }
}
burglar-alarm {
    1 {
        network_address = "192.128.42.021:5050";
        location = "loading bay";
    }
}
```

This configuration file foregoes both the `"uid-"` prefix and the `id`

variable. Instead, the configuration file uses a unique scope name for each camera or burglar alarm. The user has the responsibility of ensuring that there is no clash of these scope names, but this is hardly more of a burden than ensuring there was no clash of the values of `id` variables. Figure 7.5 shows how this configuration file can be written in an semantically identical but more compact and intuitive syntax.

Figure 7.5: More concise security configuration without `id` attributes

```
camera.1 {
    network_address = "192.128.42.006:5000";
    location = "front entrance";
};
camera.2 {
    network_address = "192.128.42.009:5000";
    location = "loading bay";
};
burglar-alarm.1 {
    network_address = "192.128.42.021:5050";
    location = "loading bay";
}
```

A security-monitoring application that uses this type of configuration file can report a problem with, for example, `"The 'camera.2' device is malfunctioning"`, which seems clear enough. Thus, in terms of usability, this approach is arguably better than the first approach (discussed in Figure 7.2).

Of course, the scope names for individual devices do not have to be integers, so a user is free to use more meaningful names:

```
camera.front-entrance-1 {
    ...
};
camera.loading-bay-1 {
    ...
};
burglar-alarm.loading-bay-1 {
    ...
}
```

The main drawback of this approach is that the Config4* schema language is not flexible enough, by itself, to validate the contents of such a configuration file. Instead, a developer must perform schema validation

in a piecemeal manner, as I now discuss.

First, the developer uses the schema shown in Figure 7.6 to validate the top-levels of the configuration file. Notice that this schema ignores the scopes nested within the `camera` and `burglar-alarm` scopes.

Figure 7.6: A top-level schema for the configuration in Figure 7.5

```
String[] schema = new string[] {
    "camera = scope",
    "@ignoreScopesIn camera",

    "burglar-alarm = scope",
    "@ignoreScopesIn burglar-alarm"
};
```

Second, to validate the details of each camera, the developer calls `listFullyScopedNames()` to obtain the names of the scopes nested within `camera`.

```
String[] names = cfg.listFullyScopedNames("", "camera",
                     Configuration.CFG_SCOPE, false);
```

The developer can then validate each of those scopes with the schema shown in Figure 7.7.

Figure 7.7: A schema for a camera scope in Figure 7.5

```
String[] schema = new string[] {
    "network_address = "string",
    "location        = "string"
};
```

The developer can validate burglar alarms scopes in a similar manner, that is, by calling `listFullyScopedNames()` to obtain a list of the scopes nested within `burglar-alarm`, and then validating each of those nested scopes.

7.4 When to use the "uid-" Prefix

In Section 7.2, I explained why you might *not* want to use the `"uid-"` prefix. This raises the question: is the `"uid-"` prefix *ever* useful? In particular, under what circumstances can the `"uid-"` prefix achieve its

intended goal of eliminating the burden for users to create identifiers with unique values? Unfortunately, I cannot give a definitive answer to that question. This is because the `"uid-"` prefix was introduced relatively late in the development cycle of Config4*, so I have not had the opportunity to use it sufficiently often to feel confident that I know all its strengths and weaknesses. However, the *recipes* configuration file shown in Figure 7.8 provides two examples of when the `"uid-"` prefix *can* be used without problems.

1. The `"uid-"` prefix can be used on the name of a scope if that scope *naturally* contains a variable whose value: (a) is guaranteed to be unique across all similar scopes; and (b) is suitable for use in human-readable messages. For example, it is reasonable to require each `uid-recipe` scope to contain a `name` variable whose value is unique and meaningful to humans. An application can unambiguously report a problem about a specific `uid-recipe` by referring to the value of its `name` variable.

2. The `"uid-"` prefix can be used on the name of an item (that is, a scope or variable) if the item is, essentially, anonymous. This is illustrated by the `uid-step` variables within each `uid-recipe` scope.

In the security-monitoring application, the `uid-burglar-alarm` and `uid-camera` scopes contain a `network_address` variable, whose value *is* unique. Because of this, the application could unambiguously identify a device with a message such as, `"The camera at network address 192.128.42.006:5000 is malfunctioning"`.

However, a network address is a low-level piece of information, and a security-monitoring application might prefer to identify a device with a more human-friendly description. It is this desire for a *meaningful to humans*, unique identifier that motivates either: (1) the introduction of the `id` variable in Figure 7.2; or (2) finding an alternative to use of the `"uid-"` prefix, as shown in Figure 7.5.

7.5 Summary

In this chapter, I have explored how an application might have a configuration file that contains a separate scope for creating each of an arbitrary number of objects. The obvious approach is to use the `"uid-"` prefix on the names of scopes. For example:

Figure 7.8: File of recipes

```
uid-recipe {
    name = "Tea";
    ingredients = ["1 tea bag", "cold water", "milk"];
    uid-step = "Pour cold water into the kettle";
    uid-step = "Turn on the kettle";
    uid-step = "Wait for the kettle to boil";
    uid-step = "Pour boiled water into a cup";
    uid-step = "Add tea bag to cup & leave for 3 minutes";
    uid-step = "Remove tea bag";
    uid-step = "Add a splash of milk if you want";
}
uid-recipe {
    name = "Toast";
    ingredients = ["Two slices of bread", "butter"];
    uid-step = "Place bread in a toaster and turn on";
    uid-step = "Wait for toaster to pop out the bread";
    uid-step = "Remove bread from toaster and butter it";
}
```

```
uid-camera { ... };
uid-camera { ... };
uid-camera { ... };
```

Unfortunately, this approach works well only if the scopes *naturally* contain a variable whose value: (1) is guaranteed to be unique across all similar scopes; and (2) is suitable for use in human-readable messages. If this is not the case, then you may find yourself introducing an artificial `id` variable inside each scope:

```
uid-camera { id = "..."; ... };
uid-camera { id = "..."; ... };
uid-camera { id = "..."; ... };
```

If you find yourself in that situation, then it may be better to forego the use of the `"uid-"` prefix, and instead employ a unique identifier as a sub-scope:

```
camera.id1 { ... };
camera.id2 { ... };
camera.id3 { ... };
```

Chapter 8

The Spring Framework

8.1 Introduction

The Spring framework (www.springsource.org) is a popular Java library for configuration-driven object creation. In particular, Spring-based applications can create Java objects of arbitrary types from configuration information in an XML file. In this chapter, I explore how Spring might be different if it obtained configuration information from a Config4* file instead of from an XML file.

The purpose of this exploration is *not* to advocate that Spring should be retrofitted with support for Config4*. After all, there is no need to fix something that is not broken. Rather, the purpose of this chapter is to show the suitability of Config4* for future projects that might need a Spring-like capability.

8.2 Terminology

Java is an Indonesian island that is famous for its export of coffee beans. This has resulted in many Americans using *java* as a slang term for coffee. This, in turn, has resulted in the Java programming language using coffee-inspired terminology for programming concepts. Of particular note, the term *bean* (as in a *coffee bean*) is often used to denote a class (or object) whose public API adheres to several conventions, including the following:

- The class has a public default (that is, parameterless) constructor.

- Rather than making a *field* (that is, an instance variable) called `foo` public, the field is kept private, but there are public operations called `getFoo()` and `setFoo()` that can be used to read and update the field.

By the way, the combination of a private field and its public `get` and `set` operations is called a *property*.

8.3 Reducing the Verbosity of Spring Beans

Now that I have explained the terms *bean* and *property*, you might be able to understand the extract of a Spring XML file shown in Figure 8.1.

Figure 8.1: Example of Spring beans

```
<bean id = "employee1" class = "com.foo.bar.Employee">
    <property name = "firstName" value = "John"/>
    <property name = "lastName" value = "Smith"/>
    <property name = "age" value = "24"/>
    <property name = "manager" ref = "owner"/>
</bean>

<bean id = "owner" class = "com.foo.bar.Employee">
    <property name = "firstName" value = "Jane"/>
    <property name = "lastName" value = "Doe"/>
    <property name = "age" value = "42"/>
</bean>
```

Each `bean` element contains configuration information that can be used to create and initialise a Java object. The `class` attribute specifies the type of object to be created. Typically, Spring will create the object by using Java's reflection capabilities to invoke the default constructor of the specified class.[1] Then Spring processes each of the `property` elements nested inside the `bean` element. For each property, Spring uses reflection to invoke a `set<Name>()` operation on the newly-created bean. When doing this, Spring uses reflection to determine the type of the parameter passed to the `set<Name>()` operation so it can convert the stringified value obtained from the XML file to that appropriate type.

[1] Spring has the ability to create an object by invoking a non-default constructor or by invoking a factory method. However, a discussion of those capabilities is outside the scope of this chapter.

Each bean has an `id` attribute that is required to have a unique value. The `manager` property in the `employee1` bean does not have a `value` attribute. Instead, it has a `ref` attribute, the value of which specifies the unique `id` of another bean. Thus, when Spring is creating the `employee1` bean, it (recursively) creates the `owner` bean too.

We can transform the XML syntax in Figure 8.1 to Config4* syntax in a straightforward manner. Each XML element becomes a correspondingly named scope in the Config4* file. If there can be multiple occurrences of the XML element, then the `"uid-"` prefix is used on the name of the corresponding Config4* scope. Thus, the `bean` and `property` elements become `uid-bean` and `uid-property` scopes. Each XML attribute becomes a variable in the Config4* file. The result of this transformation is shown in Figure 8.2.

Figure 8.2: Simple representation of beans in Config4* syntax

```
uid-bean {
    id = "employee1"; class = "com.foo.bar.Employee";
    uid-property { name = firstName; value = "John"; }
    uid-property { name = lastName; value = "Smith"; }
    uid-property { name = age; value = "24"; }
    uid-property { name = manager; ref = "owner"; }
}

uid-bean {
    id = "owner"; class = "com.foo.bar.Employee";
    uid-property { name = firstName; value = "Jane"; }
    uid-property { name = lastName; value = "Doe"; }
    uid-property { name = age; value = "42"; }
}
```

Unfortunately, this straightforward transformation has resulted in a Config4* file that is more verbose than the original XML file. However, there is room for some improvements as I now discuss.

In Section 7.3 on page 55, I turned an `uid-camera` scope that contained an `id` variable with a unique value into a scope with a name of the form *camera.id*. The same technique can be applied to Figure 8.2. In fact, the technique can be applied twice. First, we can replace the `uid-bean` scope and its `id` variable with a scope that has a name of the form *bean.id*. Second, we can replace the `uid-property` scope and its `name` variable with a scope that has a name of the form *property.name*.

These changes result in the configuration file shown in Figure 8.3.

Figure 8.3: Enhanced representation of beans in Config4* syntax

```
bean.employee1 {
    class = "com.foo.bar.Employee";
    property.firstName.value = "John";
    property.lastName.value = "Smith";
    property.age.value = "24";
    property.manager.ref = "owner";
}

bean.owner {
    class = "com.foo.bar.Employee";
    property.firstName.value = "Jane";
    property.lastName.value = "Doe";
    property.age.value = "42";
}
```

This revised Config4* file is more concise than the original XML file. Of course, the word `"property"` is written repeatedly in each bean, so that invites the possibility of writing the beans with an explicitly-opened `property` scope to save a few more keystrokes, as shown in Figure 8.4.

In summary, a straightforward translation of XML syntax into Config4* syntax can result in a more verbose file. However, with some simple tweaking, it is possible to produce a Config4* file that is more concise than its XML counterpart.

8.4 The Benefits of `@include`

Unfortunately, XML does not provide a mechanism for one XML file to include the contents of another. Because of this, the designers of Spring had to implement their own mechanism. The syntax is illustrated below:

```
<import resource="another-file.xml"/>
```

If Spring were to be redesigned to use Config4* instead of XML, then the ability to include another file would be obtained without any developer effort via the `@include` statement.

Figure 8.4: Enhanced representation of beans in Config4* syntax

```
bean.employee1 {
    class = "com.foo.bar.Employee";
    property {
        firstName.value = "John";
        lastName.value = "Smith";
        age.value = "24";
        manager.ref = "owner";
    }
}

bean.owner {
    class = "com.foo.bar.Employee";
    property {
        firstName.value = "Jane";
        lastName.value = "Doe";
        age.value = "42";
    }
}
```

8.5 The Benefits of `@copyFrom`

Sometimes, a Spring configuration file contains several `bean` element in which *most* properties have identical values. In such a case, it can be useful to reuse some of the details of one `bean` when defining the other beans. Spring uses the term *bean inheritance* to refer to this form of reuse, and an example of it is shown in Figure 8.5.

The `widget1` bean defines properties `t`, `u`, `v`, `w`, `x` and `y`. The `widget2` bean uses the `parent` attribute to specify that it will *inherit* (that is, *reuse*) some of the details from the `widget1` bean. In this case, `widget2` inherits the `class` attribute plus *most* of the properties. However, `widget2` redefines the `v` property and also defines an additional property: `z`.

If Spring were to be redesigned to use Config4* instead of XML, then the semantics of bean inheritance would be obtained without any developer effort via the `@copyFrom` statement, as you can see in Figure 8.6.

8.6 The Benefits of Pre-set Variables

Conceptually, the contents of a Spring XML file can be split into two types of configuration: *static* and *runtime*.

Figure 8.5: Example of Spring bean inheritance

```
<bean id = "widget1" class = "com.foo.bar.Widget">
    <property name = "t" value = "..."/>
    <property name = "u" value = "..."/>
    <property name = "v" value = "..."/>
    <property name = "w" value = "..."/>
    <property name = "x" value = "..."/>
    <property name = "y" value = "..."/>
</bean>

<bean id = "widget2" parent="widget1">
    <property name = "v" value = "..."/>
    <property name = "z" value = "..."/>
</bean>
```

Figure 8.6: Configt4* equivalent of Spring's bean inheritance

```
bean.widget1 {
    class = "com.foo.bar.Widget";
    property.t.value = "...";
    property.u.value = "...";
    property.v.value = "...";
    property.w.value = "...";
    property.x.value = "...";
    property.y.value = "...";
}

bean.widget2 {
    @copyFrom "bean.widget1";
    property.v.value = "...";
    property.z.value = "...";
}
```

Static configuration. The values of `id` and `class` attributes, and the values of *most* properties are likely to remain static for the duration of the project or change only rarely.

Runtime configuration. A *small* number of properties—with names like `host`, `port` and `logDir`—are likely to change for each runtime environment in which you deploy the application.

In a large, Spring-based application, the Spring XML file might contain thousands of lines of static configuration and only a few lines of runtime

configuration. With such an application, it is undesirable to tell administrators, "Here is a 2000-line Spring XML file; you need to be concerned with only fives lines in it: the `logDir` property on line 42, the `host` property on line 837, ..." From a usability perspective, it would be better to tell administrators to modify a 5-line Java properties file containing *only* runtime configuration variables, and arrange for the application to (somehow) merge the contents of that properties file with the 2000-line Spring XML file.

Spring provides a mechanism to merge the contents of a properties file with a Spring XML file. However, before explaining the mechanism, I need to provide some background information on Spring.

Spring creates Java objects from `bean` information in an XML file in a multi-step process, a slightly simplified version of which is as follows:

1. Spring parses the XML file, and stores the information in an internal format. This internal format is called *bean configuration metadata*, or *metadata* for short.

2. Spring iterates over metadata to find beans whose `class` attribute indicate they implement the `BeanFactoryPostProcessor` interface (which is defined by the Spring library). For each such bean, Spring instantiates the bean and invokes two operations (defined in the `BeanFactoryPostProcessor` interface) on it. The invocation of these operations gives the bean the opportunity to modify metadata.

3. Spring is now ready to instantiate the "ordinary" beans defined by the bean configuration metadata.

Included in the Spring library is the `PropertyPlaceholderConfigurer` class, which implements the `BeanFactoryPostProcessor` interface. This class defines a `location` property that specifies the location of a Java properties file. When a `PropertyPlaceholderConfigurer` bean is instantiated (in step 2 of the above algorithm), it iterates over the metadata, and replaces occurrences of `"${property.name}"` with the value of the named property found in the properties file. Figure 8.7 illustrates use of a `PropertyPlaceholderConfigurer` bean.

Obviously, the Spring developers had to write code to implement the `PropertyPlaceholderConfigurer` class. If Spring were to be redesigned to use Config4* instead of XML, then there would be no need to implement the `PropertyPlaceholderConfigurer` class. This is because Config4* provides several ways to merge runtime configuration with static configuration, as you can see in Figure 8.8.

Figure 8.7: Example use of the `PropertyPlaceholderConfigurer` bean

```
<bean class = "org.springframework.beans.factory.config.PropertyPlace
holderConfigurer">
    <property name = "location" value = "/path/to/file.properties"/>
</bean>

<bean id = "widget1" class = "com.foo.bar.Widget">
    <property name = "logDir" value = "${log.dir}"/>
    <property name = "logLevel" value = "${log.level}"/>
</bean>

<bean id = "tcpServer" class = "com.foo.bar.TcpServer">
    <property name = "host" value = "${tcp.host}"/>
    <property name = "port" value = "${tcp.port}"/>
</bean>
```

Figure 8.8: Merging runtime and static configuration with Config4*

```
@include getenv("FOO_CONFIG", "") if exists;

#--------
# Default values for runtime configuration
#--------
log.dir   ?= getenv("FOO_HOME") + "/log";
log.level ?= "2";
tcp.host  ?= exec("hostname");
tcp.port  ?= "8020";

bean.widget1 {
    class = "com.foo.bar.Widget";
    property.logDir.value   = .log.dir;
    property.logLevel.value = .log.level;
}

bean.tcpServer {
    class = "com.foo.bar.TcpServer";
    property.host.value = .tcp.host;
    property.port.value = .tcp.port;
}
```

The static configuration file can use the conditional assignment operator (`"?="`) to provide default values for runtime configuration variables. These variables can then be used to specify values for bean properties. The default values of runtime configuration variables can be overridden in two ways.

One way, which is illustrated in Figure 8.8, is to use an `@include` statement to access information in a runtime configuration file.

The other way is use preset configuration variables (which is discussed in the *Overview of the Config4* API* chapter of the *Config4* Getting Started Guide*). In essence, during initialisation, the application iterates over command-line options of the form `"-set name value"`, and invokes `cfg.insertString(name, value)` for each such option. Doing this, "presets" those variables in the `Configuration` object. Afterwards, the application invokes `cfg.parse("...")` to parse a configuration file.

8.7 Summary

In this chapter, I have explored how the Spring framework might have turned out differently if Config4* had been available when Spring was first being developed. This exploration identified two main benefits:

1. A Config4* syntax for defining Spring beans would have been more concise than the corresponding XML syntax. This conciseness would have benefited users because it would have made it easier to write and maintain bean definitions.

2. The designers of Spring had to write code to implement several significant pieces of functionality because XML parsers did not provide such functionality. Because Config4* *does* provide such functionality, the developers of Spring would have had to write less code if Config4* had been available to them.

I am *not* advocating that the Spring framework library be retrofitted with support for Config4*. There are probably tens of millions of lines of existing XML-based Spring configuration files in use across thousands of organisations. I do not think the potential improvements that would arise from the use of Config4* would justify the effort required for those organizations to migrate their existing projects to use Config4* syntax.

Rather, the purpose of chapter has been to illustrate that Config4* is a suitable alternative to XML for future projects that have complex configuration requirements. The next time you are about to start work

on a new XML-based project, it might be worthwhile to stop and explore the question, "Might it be better to use Config4* instead of XML?" You might discover that using Config4* will reduce the complexity of implementing the project and improve its user-friendliness.

Part III

The Config4JMS Case Study

Introduction to Part III

Config4JMS is a library, built with the aid of Config4*, that simplifies use of the Java Message Service (JMS). You can find Config4JMS in the `config4jms` subdirectory of Config4J. Config4JMS offers two significant benefits.

- Applications built with Config4JMS are significantly easier to write, more portable and more flexible than applications build against the raw JMS API.
- Config4JMS makes it easy for an inexperienced developer to "play with" JMS concepts and proprietary features provided by a JMS vendor. This can shave several days or even weeks off the time required for a developer to learn to use a JMS product.

There are several thousand lines of Java source code in the Config4JMS library, and the library provides a feature-rich but concise API. These characteristics mean that Config4JMS can be used as the basis for a case study of how Config4* can be used "in anger". That is the purpose of this part of the manual.

I start this case study, in Chapter 9, by providing an overview of JMS.

Chapter 9

Overview of JMS

9.1 Introduction

Message-oriented middleware (MOM) is software for building distributed applications that communicate by sending messages to each other. JMS is a standardised Java API for MOM products. This chapter provides an overview of JMS. Then, the next chapter explains how Config4JMS provides a simplified "wrapper" for JMS.

9.2 Terminology and Concepts

In JMS terminology, an application that sends a message is called a *producer*, and an application that receives a message is called a *consumer* (or sometimes a *subscriber*). It is common for a JMS-based application to act as both a producer and a consumer at the same time. JMS offers two different approaches for message-based communication: *queues* and *topics*.

A queue provides one-to-one communication. A producer sends a message to a queue, and the message is stored on the queue until it can be delivered to one consumer.

A topic provides one-to-many communication. A producer application sends a message to a topic. The message will be delivered to all the consumer applications that are currently registered as subscribers of that topic. By default, a subscriber registration lasts only while a consumer application is running—when a consumer application terminates,

any subscriber registrations it had will lapse automatically. However, if a consumer application registers itself as a *durable subscriber* to a topic, then the topic will store messages for a durable subscriber that is currently not running, and will deliver those messages later when the durable subscriber is restarted.

9.3 Portability

It is possible to imagine many different qualities of service that might be provided by a MOM product. For example, is there a maximum size of message that can be sent to a queue or topic? Is there a maximum number of messages that can be stored in a queue? Should a not-yet-delivered message be discarded if it has not been delivered within a specific period of time? Should messages transmitted across a network be sent in an encrypted format? Or perhaps in a compressed format?

JMS takes a two-pronged approach to standardising such qualities of service.

- Most of the interfaces in the JMS specification contain `set<Name>()` operations that can be invoked to set a desired quality of service.

- The JMS standardisation committee realised there was too much variation across the proprietary features in existing MOM products to be able to standardise *everything* with `set<Name>()` operations. However, they felt most of the proprietary features that could not be standardised in this manner were confined to the concepts of a *destination* (the genetic term for a queue or topic) and a *connection* (so called because it is often implemented as a socket connection from an application to a MOM product).

  The committee decided that portability for JMS applications could be achieved by specifying the proprietary qualities of service for connections and destinations *outside* application code. It was envisaged that an administrator would pre-create destination objects with proprietary qualities of service, and advertise these destination objects in a naming service. A JMS application would then retrieve pre-created destination objects from the naming service. Likewise, an administrator would pre-create a *connection factory* with proprietary qualities of service, and advertise it in a naming service. A JMS application would retrieve this factory object from

the naming service and use it to create connections. A connection object created in this manner would have whatever qualities of service its factory had.

The JMS specification uses the term *administered objects* to refer to connection factories and destinations because of the intention that such objects would be created by an administrator rather than by application code.

Another portability issue that the JMS standardisation committee had to consider was thread safety. The entire API of one MOM product might be thread-safe, but another MOM product might provide a mixture or thread-safe and thread-unsafe operations. The approach taken to deal with this was to introduce the concept of a *session* object that application developers should assume is *not* thread-safe. The developer of a multi-threaded application is required to create multiple session objects: one for each thread.

9.4 Problems with JMS

JMS is good but, like any technology, it is not perfect. In this section I discuss some irritations and imperfections in JMS that I have discovered. I do this to motivate the development and benefits of Config4JMS.

9.4.1 Books and Manuals Advocate the Legacy API

The JMS 1.0 specification defined one set of interfaces for communication via queues, and a separate set of interfaces for communication via topics.

The JMS 1.1 specification provides a unified set of interfaces that can be used for communication with both queues or topics. For backwards compatibility, the JMS 1.1 specification continues to support the original (queue-specific and topic-specific) interfaces.

There are several reasons why developers should use the JMS 1.1 API in new applications. First, the JMS 1.1 specification warns that the original API might be deprecated in a future version of the specification. Second, the unified API provides more opportunities for optimisation in JMS products. Third, the unified API in JMS 1.1 is more concise and easier to use than the original API.

Despite the significant benefits of the unified API, books and product manuals *continue* to use the original API in worked examples. This can

result in developers who are new to JMS needlessly learning an outdated, verbose API instead of the newer, more concise API.

9.4.2 Confusingly Many Initialisation Steps

A portable JMS application must complete many initialisation steps before it can do "real" work. For example, an application acting as both a producer and a consumer typically carries out nine initialisation steps:

1. Connect to a naming service.
2. Retrieve one or more `Destination` objects from the naming service.
3. Retrieve a `ConnectionFactory` object from the naming service.
4. Invoke `createConnection()` on the `ConnectionFactory` object to create a `Connection` object.
5. Create two `Session` objects by invoking `createSession()` twice on the `Connection` object.
6. Invoke `createProducer()` on one of the `Session` objects (passing a `Destination` object as a parameter) to create a `Producer` object.
7. Invoke `createConsumer()` on the other `Session` object (passing a `Destination` object as a parameter) to create a `Consumer` object.
8. Create and register a `MessageListener` object on the `Consumer` object. (A `MessageListener` is a callback interface whose `onMessage()` operation is invoked whenever a `Consumer` receives a message.)
9. Invoke `start()` on the `Connection` object.

Psychological research indicates that most people can remember only about seven new pieces of information at a time [Mil56]. Because of this limit, the nine-step initialisation sequence provides a hurdle for new developers to master. This is a shame, because once initialisation is complete, using JMS is straightforward.

9.4.3 Requiring Programmers to Learn Administration Skills

The first initialisation step discussed in Section 9.4.2 is to connect to a naming service that has been populated with `ConnectionFactory` and

`Destination` objects. As I explained in Section 9.3 on page 76, the intention is that those objects will be created with proprietary qualities of services (via proprietary administration tools) by an administrator; in this way, application code is not polluted with the setting of proprietary qualities of service.

When an application is being deployed in a production environment, there may well be an administrator available to create the `Destination` and `ConnectionFactory` objects, and advertise them in a naming service. However, such an administrator is unlikely to be available during initial application development. This means that JMS developers have to learn how to carry out such administration tasks themselves. In fact, a developer who is starting to learn JMS will have to learn those administration skills *before* being able to write a portable, "Hello, World"-type JMS application. This is yet another hurdle for new developers to master.

9.4.4 Only Partial Portability in JMS

In Section 9.3 on page 76, I outlined the approach used in the JMS specification to provide portability of JMS-based applications. Unfortunately, the approach used is only partially successful. It is very easy for developers to feel tempted—or sometimes be *required*—to use vendor-proprietary functionality in a JMS-based application.

In the relatively short period of time I have spent using JMS, I noticed four reasons why a developer might resort to using a proprietary API.

First, obtaining `Destination` and `ConnectionFactory` objects from a naming service is inconvenient—especially for a developer new to JMS who does not want to have to spend time learning JMS administration commands before being able to write an application. It is usually more convenient to use vendor-proprietary functions to create `Destination` and `ConnectionFactory` objects directly.

Second, it is common for JMS products to offer qualities of service above and beyond those defined in the JMS specification. If these qualities of service are related to, say, `Session`, `Producer` or `Consumer`, then it is natural for a vendor to provide proprietary `set<Name>()` operations on those types. Put simply, not all proprietary qualities of service can be encapsulated in administered objects.

Third, when a JMS-related operation fails, it throws a (subtype of) `JMSException`. The developer of an application might wish to process a caught exception in one of several ways, depending on the nature of the exception. However, the JMS specification states that two out of the

three pieces of information provided by `JMSException` are proprietary to a JMS vendor. Because of this, a developer may need to rely on vendor-proprietary information contained in an exception when deciding how to process it.

Finally, JMS specifies that a message is composed of three parts: (1) header fields, (2) arbitrary properties (that is, *name=value* pairs), and (3) a body. The intended use of (2) is to support flexible message selection in consumer applications. For example, a producer application running in, say, London, might add *location=London* to a message's properties before sending the message. A Consumer application could then use the message selector `"(location = 'London')"` to ensure it receives only messages with that property value. The JMS specification reserves property names starting with `"JMS_<vendor>"` for use by JMS vendors. Some vendors use such properties to specify a proprietary quality of service on a per message basis. A developer who wishes to make use of a proprietary, per-message quality of service will have to modify the code of a producer application so it sets the proprietary property prior to sending a message.

9.5 Critique: The 80/20 Principle

You may be familiar with some variations of the *Pareto Principle*, also known as the *80/20 Principle*:[1]

1. 80% of the wealth in a country is owned by 20% of the population.
2. 80% of CPU time is spent in 20% of the code.
3. A business gets 80% of its income from 20% of its customers.
4. 80% of the work in a company is done by 20% of the employees.

Sometimes the 80/20 Principle can suggest how to make large improvements for a relatively small investment of effort. For example, if you want to optimise an application for speed, then item 2 in the above list suggests you should use a profiling tool to identify the most CPU-intensive parts of the application, so you can focus your optimisation efforts on them.

There is a little-known variant of the 80/20 Principle that I often find useful:

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

80% of a product's complexity is in 20% of its functionality.

If you want to increase a product's ease of use, then you should identify its few areas of disproportionate complexity, so you can focus your ease-of-use efforts on them.

The problems discussed in Section 9.4 account for most of the complexity in JMS, but only a minority of its functionality. The goal of Config4JMS, which is discussed in the next chapter, is to put a "simplification wrapper" around those disproportionately complex parts of JMS.

Chapter 10

Config4JMS Functionality

10.1 Introduction

In this chapter, I discuss the functionality provided by Config4JMS. I start by explaining the structure of a Config4JMS configuration file. Then, I discuss the API of Config4JMS. Finally, I explain how use of Config4JMS can increase the portability of applications.

10.2 Syntax

Figure 10.1 shows an example of a configuration file used by Config4JMS.

The conditional assignment operator (`"?="`) is used to assign default values to the variables in the global scope. Those default values can be overridden by, for example, an application's command-line options.

The `chat` scope contains Config4JMS-related configuration information for an application. Most of the entries within the `chat` scope are nested sub-scopes that specify details of how to create JMS-related objects. For example, the `Topic.chatTopic` scope specifies how to create a `Topic` object called `chatTopic`, and `Session.prodSession` specifies how to create a `Session` object called `prodSession`.

JNDI is an acronym for the *Java Naming and Directory Interface*. The `jndiEnvironment` variable specifies details of how to connect to a naming service.

The `Topic.chatTopic` scope contains the following variable:

```
obtainMethod = "jndi#SampleTopic1";
```


Figure 10.1: Example Config4JMS configuration file

```
username        ?= "";
password        ?= "";
messageSelector ?= "";
chat {
    config4jmsClass = "org.config4jms.portable.Config4JMS";
    jndiEnvironment = [
        # name                                value
        #-----------------------------------------------
        "java.naming.factory.initial",       "...",
        "java.naming.provider.url",          "...",
        "java.naming.security.principal",    .username,
        "java.naming.security.credentials",  .password,
    ];
    Topic.chatTopic {
        obtainMethod = "jndi#SampleTopic1";
    }
    ConnectionFactory.factory1 {
        obtainMethod = "jndi#SampleConnectionFactory1";
    }
    Connection.connection1 {
        createdBy = "factory1";
        create {
            userName = .username;
            password = .password;
        }
    }
    Session.prodSession {
        createdBy = "connection1";
        create {
            transacted = "false";
            acknowledgeMode = "AUTO_ACKNOWLEDGE";
        }
    }
    Session.consSession {
        createdBy = "connection1";
        create {
            transacted = "false";
            acknowledgeMode = "AUTO_ACKNOWLEDGE";
        }
    }
... continued on the next page
```

Figure 10.1 (continued): Example Config4JMS configuration file

```
... continued from the previous page
    TextMessage.chatMessage {
        createdBy = "prodSession";
        JMSExpiration = "10 hours";
        JMSPriority = "7";
        properties = [
            # name       type      value
            #------------------------------
             "location", "string", "London",
        ];
    }
    MessageProducer.chatProducer {
        createdBy = "prodSession";
        create.destination = "chatTopic";
        deliveryMode = "PERSISTENT";
        timeToLive = "2 minutes";
    }
    MessageConsumer.chatConsumer {
        createdBy = "consSession";
        create {
            destination = "chatTopic";
            messageSelector = .messageSelector;
            noLocal = "false";
        }
    }
}
```

That variable specifies Config4JMS should obtain the `chatTopic` object by invoking `lookup("SampleTopic1")` on the naming service. An alternative setting for this variable might be:

```
obtainMethod = "file#/path/to/file/containing/a/serialised/java/object";
```

The `createdBy` variable in the `Connection.connection1` scope specifies that the `connection1` object is created by (invoking a factory method on) the object named `factory1`, which is of type `ConnectionFactory`. The `create` sub-scope specifies the parameter values to be used when invoking the factory operation.

In a similar way, the two `Session` objects are created by invoking a factory method on the `connection1` object.

The `MessageProducer.chatProducer` scope specifies that the object called `chatProducer` is created by the `prodSession` object, and that the

`destination` parameter passed to the factory operation is the `chatTopic` object. This scope also specifies values for two properties:

```
deliveryMode = "PERSISTENT";
timeToLive = "2 minutes";
```

Config4JMS invokes the `setDeliveryMode()` and `setTimeToLive()` operations on the object to set those properties. When doing do, Config4JMS converts `"PERSISTENT"` into the appropriate integer constant, and converts `"2 minutes"` into the appropriate number of milliseconds.

Although the example configuration file shows only two properties being set, Config4JMS can be used to set *any* of the properties defined in the JMS specification. As I will discuss in Section 10.4, it is possible for Config4JMS to also set properties that are proprietary to a JMS product.

10.3 API

The API of Config4JMS is defined in the `org.config4jms.Config4JMS` class, which is shown in Figure 10.2. For brevity, `throws` clauses are not shown in Figure 10.2.

Rather than discuss every operation individually, I will use the sample code in Figure 10.3 to illustrate basic usage of the Config4JMS API. Then afterwards, I will discuss any remaining operations not illustrated by the example.

10.3.1 Basic Usage

The code in Figure 10.3 is an extract from the `Chat.java` sample application (in the `samples/chat` sub-directory of Config4JMS).

The code populates a `HashMap` with *name=value* pairs for a username and password. In the full `Chat.java` application, the `HashMap` is populated via command-line options.

Then the `Config4JMS.create()` factory operation is invoked to create a `Config4JMS` object. The first two parameters to this factory operation specify a configuration file (such as that shown in Figure 10.1 on page 84) and a scope within that file. The third parameter is the `HashMap` containing *name=value* pairs. Config4JMS uses these to "preset" variables in a `Configuration` object before parsing the specified configuration file. In this way, these preset variables can override the default values of variables initialised with the conditional assignment operator (`"?="`) in the

Figure 10.2: Config4JMS API

```
package org.config4jms;
//--------
// Most operations can throw a Config4JMSException.
// However, the "throws" clause is omitted for brevity.
//--------
public abstract class Config4JMS {
    //--------
    // No public constructor. Use these create() operations instead.
    //--------
    public static Config4JMS create(String    cfgSource,
                                    String    scope,
                                    Map       cfgPresets);

    public static Config4JMS create(String    cfgSource,
                                    String    scope,
                                    Map       cfgPresets,
                                    String[]  typeAndNamePairs);
    //--------
    // Retrieve an object by name.
    //--------
    public Object            getObject(String type, String name);
    public ConnectionFactory getConnectionFactory(String name);
    public Connection        getConnection(String name);
    public Session           getSession(String name);
    public Destination       getDestination(String name);
    public Queue             getQueue(String name);
    public Queue             getTemporaryQueue(String name);
    public Topic             getTopic(String name);
    public Topic             getTemporaryTopic(String name);
    public MessageProducer   getMessageProducer(String name);
    public MessageConsumer   getMessageConsumer(String name);
    public QueueBrowser      getQueueBrowser(String name);

    //--------
    // Connection operations.
    //--------
    public void setExceptionListener(ExceptionListener listener);
    public void startConnections();
    public void stopConnections();
    public void closeConnections();

... continued on the next page
```


Figure 10.2 (continued): Config4JMS API

```
... continued from the previous page
    //--------
    // Message operations
    //--------
    public Message createMessage(String name);
    public void applyMessageProperties(String name, Message msg);

    //--------
    // List the names of objects of different types.
    //--------
    public String[] listConnectionFactoryNames();
    public String[] listConnectionNames();
    public String[] listSessionNames();
    public String[] listDestinationNames();
    public String[] listQueueNames();
    public String[] listTemporaryQueueNames();
    public String[] listTopicNames();
    public String[] listTemporaryTopicNames();
    public String[] listMessageProducerNames();
    public String[] listMessageConsumerNames();
    public String[] listQueueBrowserNames();
    public String[] listMessageNames();
    public String[] listBytesMessageNames();
    public String[] listMapMessageNames();
    public String[] listObjectMessageNames();
    public String[] listStreamMessageNames();
    public String[] listTextMessageNames();

    //--------
    // Frequently used miscellaneous operations
    //--------
    public void          createJMSObjects();
    public Configuration getConfiguration();
    public boolean       isNoConnection(JMSException ex);

    //--------
    // Rarely used miscellaneous operations
    //--------
    public Object  importObjectFromJNDI(String path);
    public Object  importObjectFromFile(String fileName);
}
```

Figure 10.3: Example Use of `Config4JMS`

```
HashMap            cfgPresets = new HashMap();
Config4JMS         jms = null;
MessageProducer    producer = null;
TextMessage        msg = null;

//--------
// Initialisation
//--------
try {
    cfgPresets.put("username", "Fred");
    cfgPresets.put("password", "123456");
    jms = Config4JMS.create("example.cfg", "chat", cfgPresets,
                new String[] {"MessageConsumer", "chatConsumer",
                              "MessageProducer", "chatProducer",
                              "TextMessage",     "chatMessage"});
    jms.createJMSObjects();
    jms.getMessageConsumer("chatConsumer").setMessageListener(this);
    producer = jms.getMessageProducer("chatProducer");
    jms.startConnections();
} catch(JMSException ex) {
    System.err.println(ex.toString());
    try {
        if (jms != null) { jms.closeConnections(); }
    } catch(JMSException ex) {
        System.err.println(ex.toString());
    }
    System.exit(1);
}

//--------
// Sample producer code
//--------
msg = (TextMessage)jms.createMessage("chatMessage");
msg.setText("Hello, World");
producer.send(msg);
```

configuration file. The last (and optional) parameter to `create()` is an array of pairs of strings. Each pair specifies the *type* and *name* of an object that is expected to be specified in the configuration file. In effect, this parameter specifies a contract between the configuration file and the Java code. (If the contents of the configuration file do not satisfy the

contract, then `Config4JMS.create()` throws an exception.) This contract enables a Java developer to *not* have to continually refer back to a configuration file to verify the types and names of the JMS objects being used in the Java code.

The `create()` operation parses the specified configuration file, performs schema validation on the specified scope, and ensures that the configuration file defines the expected objects of the specified *type* and *name*. The `create()` operation also copies the configuration information into a more convenient internal format.

The `createJMSObjects()` operation instructs Config4JMS to create all the objects defined in the configuration file.

After `createJMSObjects()` has been invoked, an application can call `get<Type>()` operations to retrieve specific objects by name. For example, `getMessageConsumer("chatConsumer")` returns the `chatConsumer` object of type `MessageConsumer`.

The `startConnections()` operation instructs Config4JMS to invoke the `start()` operation on all the `Connection` objects that it created from information in the configuration file. Likewise, `closeConnections()` causes `close()` to be invoked on all the `Connection` objects.

Each time `createMessage()` is called, it creates a new `Message` object. It sets headers specified in configuration (such as `JMSExpiration` and `JMSPriority`) and can also set type-specific *name=value* pairs as indicated in the `properties` configuration variable.

10.3.2 Other Operations

The example in Figure 10.3 illustrates most of the commonly-used operations provided by `Config4JMS`. I now quickly summarise its remaining operations.

Calling `applyMessageProperties()` on an existing `Message` object resets the object's headers and properties to values specified in the named configuration scope.

Each `list<Type>Names()` operation returns an array of the names of objects of the specified type. For example, when using the configuration file shown in Figure 10.1 on page 84, the `listSessionNames()` operation would return a list containing two strings: `"prodSession"` and `"consSession"`.

The `getConfiguration()` operation returns a reference to the Config4* `Configuration` object used internally by the `Config4JMS` object.

The `setExceptionListener()` operation instructs Config4JMS to register an `ExceptionListener` object on all the `Connection` objects created from information in the configuration file.

The `isNoConnection()` operation examines a `JMSException` to determine if it was caused by a broken socket connection. A developer might use this operation in combination with `setExceptionListener()` to write an application that can detect when its connection to JMS infrastructure is broken, and attempt to re-establish the connection. The `ReconnectableChat.java` sample application (in the `samples/chat` subdirectory of Config4JMS) illustrates this technique.

The `importObjectFromJNDI()` operation uses the specified `path` to lookup an object from the naming service that was configured via the `jndiEnvironment` configuration variable.

The `importObjectFromFile()` operation reads a serialised Java object from the specified `fileName`.

10.4 Accessing Proprietary Features

`Config4JMS` is an abstract base class. Its static `create()` operation uses reflection to instantiate a concrete subclass. The name of the concrete subclass is specified by the `config4jmsClass` variable in the runtime configuration file. The example configuration file shown in Figure 10.1 on page 84 sets that variable as follows:

```
config4jmsClass = "org.config4jms.portable.Config4JMS";
```

That class has been coded to recognise only configuration entries that reflect the standardised API of JMS. I have implemented another class that recognises the standard API *plus* proprietary enhancements of the SonicMQ implementation of JMS. You can use that class with the following setting:

```
config4jmsClass = "org.config4jms.sonicmq.Config4JMS";
```

I do not have any experience of other JMS products, but I imagine it should be possible to write additional subclasses of `Config4JMS` that support their proprietary enhancements.

If the value of `config4jmsClass` specifies the SonicMQ-specific class, then you can use SonicMQ-proprietary features in a straightforward manner, as I now discuss.

You can create instances of SonicMQ-proprietary types by creating appropriate configuration scopes:


```
MultiTopic.myTopic { ... }
XMLMessage.myMessage { ... }
```

You can set SonicMQ-proprietary properties in the same manner that you set JMS-standardised properties:

```
MessageConsumer.chatConsumer {
    createdBy = "consSession";
    create { ... }
    prefetchCount = "5";      # proprietary to SonicMQ
    prefetchThreshold = "2";  # proprietary to SonicMQ
}
```

The configuration file in Figure 10.1 on page 84 retrieves administered objects from the naming service by setting `obtainMethod` to a value of the form `"jndi#..."`, as I repeat below for ease of reference:

```
Topic.chatTopic {
    obtainMethod = "jndi#SampleTopic1";
}
ConnectionFactory.factory1 {
    obtainMethod = "jndi#SampleConnectionFactory1";
}
```

Within a `ConnectionFactory` sub-scope, if you set `obtainMethod` to the value `"create"`, then you can create a `ConnectionFactory` using a proprietary constructor and use properties to specify a proprietary quality of service for it:

```
ConnectionFactory.factory1 {
    obtainMethod = "create";
    create {                      # parameters to constructor
        brokerURL = "...";
        defaultUserName = "...";
        defaultPassword = "...";
    }
    faultTolerant = "true;        # proprietary property
    flowToDisk = "ON";            # proprietary property
}
```

Within a `Topic` or `Queue` sub-scope, setting `obtainMethod` to the value `"create"` enables you to create a topic or queue by invoking a factory operation on a `Session` object:

```
Topic.chatTopic {
    obtainMethod = "create";
    createdBy = "prodSession";
    create {                    # parameters to constructor
        topicName = "...";
    }
}
```

The JMS specification defines factory operations on `Session` for creating destination object. However, the JMS specification does not define allowable values for the `queueName` or `topicName` parameter; such values are vendor-proprietary, which is why using factory operations to create destination objects is considered proprietary.

10.5 Benefits

Config4JMS offers several significant benefits, most of which are due to it enforcing a separation of concerns:[1] it separates the initialisation of JMS from the "business logic" code that uses JMS.

10.5.1 Code Readability

As I explained in Section 9.4.2 on page 78, the initialisation of JMS is verbose enough to confuse developers who are new to JMS. By encapsulating the initialisation steps in a configuration file, a new developer can write that once, forget about it, and then focus on the "business logic" code in a Java file.

For example, in the source code of Figure 10.3 on page 89, the programmer need be concerned with just three JMS objects: a `TextMessage`, `MessageConsumer` and a `MessageProducer`. The other six JMS-related objects (a naming service, `Topic`, `ConnectionFactory`, `Connection`, and two `Session` objects) are really just "initialisation baggage" that has been encapsulated by Config4JMS.

The impact on code readability of encapsulating "initialisation baggage" can be dramatic. For example, the `Chat.java` demo supplied with Config4JMS contains significantly less code and is easier to understand than an equivalent demos written using the raw API of JMS.

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

10.5.2 Configurability

A lot of JMS behaviour is determined by qualities of service, for example, timeout values and whether messages are persistent. All these qualities of service can be expressed in a Config4JMS configuration file, which means that a Config4JMS-based application is highly configurable *by default*.

In contrast, if an application uses just the raw API of JMS, then its developer must explicitly write extra code to: (1) retrieve qualities of service information from a runtime configuration file, and (2) invoke the appropriate `set<Name>()` operations to apply them. If a developer is too busy or lazy to write such code, then the application will provide a hard-coded rather than configurable quality of service.

10.5.3 A Portable Way to Use Proprietary Features

Consider the following scenario. A producer application does not send any messages for an extended period of time, but then it sends a burst of, say, a hundred large messages, before going silent again. Such bursts of messages might cause a backlog of traffic to be sent via JMS, and this backlog might cause communications between applications to slow down while the backlog of messages is being transmitted.

The JMS specification does not offer any help to deal with a potential slowdown caused by a burst of many messages. However, some JMS products provide proprietary enhancements for dealing with this. For example, the SonicMQ[2] product, provides a *flow control* feature that can throttle back the rate of message flow from a producer, and a separate *flow to disk* feature that can be used by consumer applications that cannot process a sudden burst of messages fast enough.

If you are using SonicMQ when developing a JMS application, then you might be be tempted to make use of the *flow control* and *flow to disk* features. Unfortunately, hard-coding use of these proprietary features into the Java code of your application would make your application non-portable to other JMS products. But if you use Config4JMS, then you can specify the use of these features in a configuration file. In this way, your Java code remains portable. If you later migrate to another JMS product, then you need only modify the configuration file to remove use of the SonicMQ-proprietary features and, optionally, make use of features proprietary to the replacement JMS product.

[2] http://web.progress.com/en/sonic/sonicmq.html

10.5.4 Reusability of Demonstration Applications

A JMS product might contain a dozen or more demonstration applications. One application is hard-coded to demonstrate communication via *queues*. A second application is hard-coded to demonstrate communication via *topics*. A third application is hard-coded to demonstrate the use of *durable subscribers* with topics. Several more applications are hard-coded the demonstrate the use of JMS-compliant qualities of service (using a separate application to demonstrate each quality of service). Yet more applications are hard-coded to demonstrate the use of vendor-proprietary qualities of service. And so on.

By using Config4JMS, a JMS vendor can significantly reduce the number of demonstration applications that need to be provided with a product. For example, the `Chat.java` application supplied with Config4JMS can be used to demonstrate: (1) communication via queues; (2) communication via topics; (3) the use of durable subscribers with topics; (4) different JMS-compliant qualities of service; (5) different vendor-proprietary qualities of service. All that is needed is to modify the application's Config4JMS configuration file to specify the desired type of communication and the desired qualities of service.

Shipping a JMS product with a small number of highly configurable Config4JMS-based demonstration applications offers several benefits.

First, it benefits the JMS vendor because fewer demonstration applications have to be written, maintained and documented.

Second, it can shorten the learning curve for developers who are new to JMS or new to the proprietary features of a JMS product. This is because the develops can "play with" JMS concepts and proprietary features without having to write any code—instead, they just edit a configuration file. This shortening of the learning curve benefits not just developers employed by customers, but also newly employed technical staff of the JMS vendor.

Third, a Config4JMS-enabled demonstration application can be configured to obtain `Destination` and `ConnectionFactory` objects: (1) by invoking proprietary factory operations; or (2) from a pre-populated naming service. A developer who is learning to how use a JMS product can use technique (1) initially, and then switch to (2) later after learning how to do the required administration tasks. This means that a developer does *not* need to learn administration skills *before* being able to write a portable JMS application.

Finally, if a customer discovers a bug in a JMS product, then Config4JMS makes it easier for the customer to submit a test case. This is because a test case is likely to consist of a small amount of Java code (perhaps one of the demonstration applications supplied with the JMS product), plus a configuration file. In fact, the *adaptive configuration* capabilities Config4* might sometimes make it possible for a *single* configuration file to demonstrate a bug *and* a workaround for it. This possibility is illustrated by the configuration file shown in Figure 10.4.

Figure 10.4: Flexible test-case configuration file

```
workAroundBug ?= "false";
chat {
    config4jmsClass = "org.config4jms.acme.Config4JMS";
    jndiEnvironment = [ ... ];
    Topic.chatTopic { ... }
    ConnectionFactory.factory1 { ... }
    Connection.connection1 { ... }
    Session.prodSession {
        createdBy = "connection1";
        create {
            transacted = "false";
            acknowledgeMode = "AUTO_ACKNOWLEDGE";
        }
        @if (.workAroundBug == "true") {
            ... # set a proprietary property to one value
        } @else {
            ... # set the proprietary property to another value
        }
    }
    Session.consSession { ... }
    TextMessage.chatMessage { ... }
    MessageProducer.chatProducer { ... }
    MessageConsumer.chatConsumer { ... }
}
```

Let's assume that, using that configuration file, the buggy behaviour can be illustrated by running the following command:

```
java TestCase -cfg test-case.cfg -scope chat
```

Then, by using a command-line option to "preset" the `workAroundBug` variable to `true`, the workaround for the bug can be illustrated:

```
java TestCase -cfg test-case.cfg -scope chat -set workAroundBug true
```

10.6 Drawbacks

Config4JMS has only a few, relatively minor drawbacks. I discuss them here for the sake of completeness.

10.6.1 Only Two Implementations So Far

To date, there are only two implementations of Config4JMS: one that provides access to only the portable API of JMS; and another implementation that provides access to that portable API plus the proprietary features of the SonicMQ product. It would be good to see Config4JMS enhanced to provide additional implementations that support the proprietary features of other JMS vendor products.

I am not aware of any technical issues that might make it difficult to enhance Config4JMS in this way. I estimate that a person who is already knowledgeable about a particular JMS product could extend Config4JMS to support its proprietary features with a few days of effort (at most).

10.6.2 Lack of Support for Legacy API

Currently, Config4JMS supports only the unified API of JMS 1.1. It should be easy to add support for the legacy API of JMS 1.0, if the need ever arises. However, I view the lack of support for the legacy API as being a *benefit* because, as I explained in Section 9.4.1 on page 77, there are several drawbacks (and *no* benefits) associated with use of the legacy API.

10.7 Summary

In this chapter, I have explained how Config4JMS hides a lot of that complexity of JMS. The syntax of a Config4JMS configuration file is straightforward and the API is easy to use. Despite this simplicity, Config4JMS offers some significant benefits.

- The confusingly many initialisation steps for a JMS application can be encapsulated in a configuration file, and Java code can then focus on implementing the "business logic" code of an application. This separation of concerns helps to improve code readability.

- It is common for a JMS-based application to hard-code a particular quality of service. In contrast, a Config4JMS-based application expresses all qualities of service in a configuration file, which makes the application highly configurable by default.
- The use of proprietary features of a JMS product can be encapsulated in a configuration file, thus preserving the portability of Java application code.
- Config4JMS makes it easy for developers to "play with" JMS concepts or the proprietary features of a product without having to write much, if any, Java code. This can significantly reduce the learning curve for developers new to JMS or a particular JMS product.

In the previous chapter, I explained how one variant of the 80/20 Principle applied to JMS: 80% of a product's complexity is in 20% of its functionality. Config4JMS hides most of that complexity.

Chapter 11

Architecture of Config4JMS

11.1 Introduction

In this chapter, I discuss how Config4JMS makes effective use of Config4* and its schema language. However, to help readers understand *why* Config4JMS uses Config4* in the way it does, I first need to provide an overview of the architecture of Config4JMS.

11.2 Packages

The source code of Config4JMS is spread over four packages:

```
org.config4jms
org.config4jms.base
org.config4jms.portable
org.config4jms.sonicmq
```

The `org.config4jms` package contains just two classes. One of these, `Config4JMS`, is an abstract base class that defines the API of Config4JMS. The other class, `Config4JMSException.java`, inherits from `JMSException`.

The `org.config4jms.base` package contains some basic functionality that is used by the classes in both the `portable` and `sonicmq` packages.

The `org.config4jms.portable` package contains a concrete subclass of `Config4JMS`, plus supporting classes. The concrete subclass, which is also called `Config4JMS`, supports the standardised API of JMS.

The `org.config4jms.sonicmq` package contains a concrete subclass of `Config4JMS`, plus supporting classes. This concrete subclass, which is also called `Config4JMS`, supports the standardised API of JMS *plus* proprietary enhancements that are specific to the SonicMQ implementation of JMS.

I considered having the classes in the `sonicmq` package inherit from their counterparts in the `portable` package. However, I decided against this approach because I felt it might result in the anti-pattern known as the yo-yo problem.[1] Instead, I felt it was better (or, at least, less bad) to employ the "code reuse by copy-and-pasting" anti-pattern. If you wish to extend Config4JMS to support another implementation of JMS called, say, Foo, then you can do this by creating a package called `org.config4jms.foo`, copying all the files from the `portable` package into this new package, and then modifying the copied files to add support for Foo-proprietary features.

11.3 Important Classes

Abridged details of three important classes in the `org.config4jms.base` package are shown in Figure 11.1. I will discuss each of the three classes in turn.

11.3.1 The `Info` Class

There is a subclass of `Info` for each JMS-related type. For example, the `ConnectionInfo` class is for the JMS `Connection` type, and the `SessionInfo` class is for the JMS `Session` type. One entire set of subclasses of `Info` are defined in the `portable` package. Another entire set of subclasses of `Info` are defined in the `sonicmq` packages.

The configuration file in Figure 10.1 on page 84 defines may scopes for JMS objects. Config4JMS creates an instance of the appropriate `Info` subclass for *each* of those scopes. For example, Config4JMS creates a `ConnectionInfo` object for the `Connection.connection1` scope, and creates two `SessionInfo` objects: one for the `Session.prodSession` scope, and another for the `Session.consSession` scope.

A concrete subclass of `Info` implements its operations as follows.

- The `validateConfiguration()` operation performs schema validation of the configuration scope for the object. For efficiency, this

[1] http://en.wikipedia.org/wiki/Yo-yo_problem

Figure 11.1: Important classes in the `org.config4jms.base` package

```
public abstract class Info
{
    public abstract void validateConfiguration()
                                    throws Config4JMSException;
    public abstract void createJMSObject()
                                    throws Config4JMSException;
    public abstract Object getObject();
    ... // other operations omitted for brevity
}

public class TypeDefinition
{
    public TypeDefinition(
              String      typeName,
              String[]    ancestorTypeNames,
              String      className);
    ... // operations omitted for brevity
}

public class TypesAndObjects
{
    private TypeDefinition[]  typeDefinitions;
    private HashMap           objects;

    public void validateConfiguration(
                      Config4JMS  config4jms,
                      String      scope) throws Config4JMSException;
    ... // other operations omitted for brevity
}
```

operation also caches the values of configuration variables in instance variables.

- The `createJMSObject()` operation creates the JMS object and sets its properties, according to the (validated and cached) information in the configuration scope.

- The `getObject()` operation returns a reference to the newly created JMS object.

11.3.2 The `TypeDefinition` Class

The `Config4JMS` class is *not* hard-coded with knowledge of the numerous subclass of `Info`. Instead, `Config4JMS` uses Java reflection to create and manipulate `Info` objects from metadata.[2] This metadata is provided by instances of the `TypeDefinition` class (see Figure 11.1). I will illustrate this with three examples.

The `TypeDefinition` object below indicates that a `Session` (for example, a `Session.prodSession` scope) should be processed by creating an instance of the `org.config4jms.portable.SessionInfo` class. The `null` value indicates that `Session` is a base type, that is, it does not have any ancestor types:

```
new TypeDefinition("Session", null,
                   "org.config4jms.portable.SessionInfo");
```

This next `TypeDefinition` object indicates that `Topic` is a subtype of `Destination`, and an instance of `Topic` should be processed by creating an instance of the `org.config4jms.portable.TopicInfo` class:

```
new TypeDefinition("Topic", new String[]{"Destination"},
                   "org.config4jms.portable.TopicInfo");
```

This final example indicates that `Destination` has neither ancestor types nor an implementation class. In effect, it is an *abstract* base type:

```
new TypeDefinition("Destination", null, null);
```

The abstract nature of `Destination` means you cannot have `Destination` sub-scopes in a Config4JMS configuration file. However, an application *can* invoke, say, `getDestination("chatTopic")` on a `Config4JMS` object because `Topic` is a subtype of `Destination`. Likewise, invoking `listDestinationNames()` would return `"chatTopic"` among its results.

11.3.3 The `TypesAndObjects` Class

The `TypesAndObjects` class (see Figure 11.1) contains two instance variables:

[2] Readers not familiar with Java reflection can find an informative overview in a free training course: www.ciaranmchale.com/training-courses.html#training-java-reflection. A more detailed discussion of Java reflection can be found in an excellent book [FF05] upon which that training course is based.

```
private TypeDefinition[]  typeDefinitions;
private HashMap           objects;
```

The `typeDefinitions` array holds metadata for all JMS data types. The `org.config4jms.portable.PortableTypesAndObjects` class creates an array of `TypeDefinition` for all the standardised JMS types. Conversely, the `org.config4jms.sonicmq.SonicMQTypesAndObjects` class creates an array of `TypeDefinition` for all the standardised JMS types *plus* the SonicMQ-proprietary types.

The `objects` variable is a `HashMap` that provides a flexible way to retrieve an `Info` object. For example, a `TopicInfo` object created from the `Topic.chatTopic` scope is registered in the `HashMap` via three keys: `"chatTopic"`, `"Topic,chatTopic"` and `"Destination,chatTopic"`.

- Storing the `Info` object under the name `"chatTopic"` provides an easy way for Config4JMS to check (and complain) if a name has been reused for different JMS types. For example, the scopes `Topic.foo` and `Session.foo` would result in an exception being thrown due to the reuse of the name `foo`.

- Storing the `Info` object under both `"Destination,chatTopic"` *and* `"Topic,chatTopic"` makes it possible for an application to obtain `"chatTopic"` in the results from calling `listDestinationNames()` *or* `listTopicNames()`.

The `TypesAndObjects` class provides a lot of operations that manipulate its instance variables. In fact, a lot of Config4JMS functionality is implemented by having the `Config4JMS` class delegate to the `TypesAndObjects` class.

11.4 Algorithms Used in Config4JMS

With the knowledge of important infrastructure classes provided in Section 11.3, the implementation of Config4JMS is easy to understand.

11.4.1 Initialisation

Pseudocode for the initialisation of a `Config4JMS` object is shown in Figure 11.2.

The `create()` operation creates an empty `Configuration` object and copies all the *name=value* pairs from the `cfgPresets` variable into it.

Figure 11.2: Pseudocode implementation of `Config4JMS.create()`

```
package org.config4jms;
public abstract class Config4JMS
{
    public static Config4JMS create(
        String             cfgSource,
        String             scope,
        Map                cfgPresets) throws Config4JMSException
    {
        //--------
        // Parse the configuration file and retrieve the name of the
        // concrete subclass that we should create.
        //--------
        cfg = Configuration.create();
        ... // populate cfg with name=value pairs from cfgSource
        cfg.parse(cfgSource);
        className = cfg.lookupString(scope, "config4jmsClass",
                                "org.config4jms.portable.Config4JMS");

        //--------
        // Use reflection to create an instance of the specified class
        //--------
        c = Class.forName(className);
        cons = c.getConstructor(new Class[]
                                Configuration.class, String.class);
        return (Config4JMS)cons.newInstance(new Object[]cfg, scope);
    }

    protected Config4JMS(
        Configuration      cfg,
        String             scope,
        TypesAndObjects    typesAndObjects) throws Config4JMSException
    {
        this.cfg = cfg;
        this.scope = scope;
        this.typesAndObjects = typesAndObjects;
        naming = null;
        jndiEnvironment = cfg.lookup(scope, "jndiEnvironment",
                                     new String[0]);
        typesAndObjects.validateConfiguration(this, scope);
    }
    ...
}
```

Then the configuration file is parsed, and `lookupString()` is invoked to get the value of the `config4jmsClass` variable. Finally, Java's reflection capabilities are used to create an instance of the specified class, which must be a subclass of `org.config4jms.Config4JMS`.

The constructor of the subclass just invokes its parent class's constructor, passing a `TypesAndObjects` parameter that provides the metadata necessary to create JMS-based objects via reflection.

The base class constructor, which is shown in Figure 11.2, initialises instance variables and then performs schema validation of the configuration file by invoking `validateConfiguration()` on its `typesAndObjects` object.

11.4.2 Schema Validation

`TypesAndObjects.validateConfiguration()` performs schema validation in three steps.

Step 1. A schema for the top-level of the configuration scope is created. This schema is of the form:

```
String schema = new String[] {
    "config4jmsClass = string",
    "jndiEnvironment = table[name,string, value,string]",
    "ConnectionFactory = scope",
    "Connection = scope",
    "Session = scope",
    ...
};
```

Only the first two entries (`config4jmsClass` and `jndiEnvironment`) in the schema are hard-coded. The remaining schema entries are obtained by iterating over the `typeDefinitions` array, which holds metadata for all JMS data types. For each non-abstract data type in that array, a string of the form `"<type-name> = scope"` is added to the schema.

Once that schema definition has been created, it is used to validate the top-level configuration scope:

```
sv = new SchemaValidator(schema);
sv.validate(cfg, scope, "", false, Configuration.CFG_SCOPE_AND_VARS);
```

The `false` parameter indicates that the validation should *not* recurse into sub-scopes.

Step 2. Each sub-scope corresponding to a JMS data type can contain nested scopes but *not* variables. For example, a `Session` scope can contain nested scopes (one for each session object), but it cannot contain any variables. The code below performs this validation check.

```
schema = new String[0];
sv = new SchemaValidator(schema);
for (i = 0; i < typeDefinitions.length; i++) {
    typeDef = typeDefinitions[i];
    if (typeDef.getIsAbstract()) { continue; }
    typeName = typeDef.getTypeName();
    if (cfg.type(scope, typeName) == Configuration.CFG_NO_VALUE) {
        continue;
    }
    sv.validate(cfg, scope, typeName, false,
                Configuration.CFG_VARIABLES);
}
```

First an *empty* schema (that is, an array containing zero strings) is created. Then, when iterating over the `typeDefinitions` array, abstract types are ignored (because they cannot have a scope in the configuration file). A concrete type is also ignored if `cfg.type()` indicates there is no scope matching the type's name in the configuration file. If there is a scope for the type, then it is validated, but *only* for the variables it might contain.

Step 3. Nested `for`-loops are used to iterate over every *type.name* configuration sub-scope (for example, `ConnectionFactory.factory1`, `Connection.connection1`, `Topic.chatTopic`, `Session.prodSession`, `Session.consSession` and so on). For each such sub-scope, Java's reflection capabilities are used to create an instance of a `<type>Info` object for that scope. For example, a `TopicInfo` object is created for the `Topic.chatTopic` scope. Each `<type>Info` object is registered in the `objects` map multiple times (as discussed in Section 11.3.3 on page 102). Then, `validateConfiguration()` is invoked on the newly created `<type>Info` object so the object can validate (and cache in instance variables) the configuration variables in its own scope. The schema validation code within a `<type>Info` class is straightforward, as you will be able to see if you example the source code of any those classes.

The three-step algorithm could be simplified by combining steps 1 and 2 into a single step, as I now discuss.

Recall that step 1 creates the schema definition by iterating over the `typeDefinitions` array and, for each non-abstract data type in that array, adding a string of the form `"<type-name> = scope"` to the schema. That algorithm could be modified so that another string, this one of the form `"@ignoreScopesIn <type-name>"`, is also added to the schema. The resulting schema would be of the form:

```
String schema = new String[] {
    "config4jmsClass = string",
    "jndiEnvironment = table[name,string, value,string]",
    "ConnectionFactory = scope",
    "@ignoreScopesIn ConnectionFactory",
    "Connection = scope",
    "@ignoreScopesIn Connection",
    "Session = scope",
    "@ignoreScopesIn Session",
    ...
};
```

Once that schema definition has been created, it would then be used to perform a *recursive* schema validation of the top-level configuration scope:

```
sv = new SchemaValidator(schema);
sv.validate(cfg, scope, "", true, Configuration.CFG_SCOPE_AND_VARS);
```

The `true` parameter indicates that the validation *does* recurse into sub-scopes.

Those changes to step 1 of the algorithm are quite trivial (they require adding one line of new code and modifying another line of existing code), and they eliminate the need for step 2 (which accounts for 12 lines of code). Some readers may be wondering why Config4JMS does not use the simpler algorithm that I just described. There are two reasons for this:

- I developed Config4JMS *before* I added ignore rules to the schema language. By the time I had introduced support for ignore rules, I had left my employer and, in doing so, lost access to the license for the JMS product I had used to develop Config4JMS. I could easily make the two-line change to simplify the algorithm, but I would not be able to test the changes. I decided to not ship Config4JMS in an untested state.

- I think the discussion of the original and simplified algorithms is beneficial for readers because it illustrates how ignore rules can shorten and simplify schema validation code.

11.4.3 The `createJMSObjects()` Operation

`Config4JMS.createJMSObjects()` delegates to an identically-named operation on the `TypesAndObjects` class. That operation iterates over all the `<type>Info>` objects that had been created during step 3 of the schema validation algorithm, and invokes `createJMSObject()` on each object. The only complication is that JMS objects have to be created in a particular order. For example, a `ConnectionFactory` object must be created before it can be used to create a `Connection` object. Likewise, a `Connection` object must be created before creating `Session` objects.

The order-of-creation guarantee is provided in a simple way. When an array of `TypeDefinition` objects is being created (see Section 11.3.3 on page 102), the order of elements in the array specifies the order in which objects of those types will be created. That enables `createJMSObjects()` to ensure that JMS objects are created in the required order.

11.5 Comparison with Spring

The purpose of Config4JMS overlaps a bit with that of the Spring framework. In particular, both use information in a configuration file to create Java objects. This overlap is bound to invite comparisons between the two projects. In reality, the two projects are more different than alike, so a comparison of them would be akin to comparing apples and oranges.

One obvious difference between Spring and Config4JMS is the configuration syntax used: Spring uses XML while Config4JMS uses Config4*. However, that difference is relatively unimportant.

A much more significant difference—and what I consider to be the *primary* difference—is that Spring can create Java objects of *arbitrary* types, while Config4JMS is restricted to creating *only* JMS objects. Many other differences between Spring and Config4JMS can be traced back to that primary difference. For example:

- The general-purpose nature of Spring requires a `bean` definition to specify the fully-scoped (and hence verbose) name of its class. In contrast, the specialised nature of Config4JMS means it can get away with specifying the more concise local name of a class.

- The specialised nature of Config4JMS means that it can be hard-coded to automatically perform data-type conversion. For example, Config4JMS uses `lookupDurationMilliseconds()` of Config4* when setting the `JMSExpiration` or `timeToLive` properties of JMS objects. The general-purpose nature of Spring prevents the use of such techniques.

- The specialised nature of Config4JMS means it is a relatively small project: just a few thousand lines of code that compiles to an 80KB jar file. There is another 106KB for the Config4J jar file upon which Config4JMS depends, thus making for a total of 186KB in jar files. Spring offers significantly more functionality than Config4JMS, and this is reflected in it being approximately one hundred times bigger.

11.6 Future Maintenance

My inspiration for developing Config4JMS came about in early 2010, when my manager asked me to learn JMS. I began by reading the JMS specification and some product manuals. Unfortunately, the problems discussed in Section 9.4 on page 77 made learning JMS more difficult than I had expected, so I decided to write Config4JMS. I figured that: (1) implementing this class library would help me to learn JMS; and (2) the resulting class library might actually be useful. I was right on both counts. Development of Config4JMS took about two weeks of hard but enjoyable work.

Unfortunately, a few few months after developing Config4JMS, I was laid off during a restructuring of the company I worked for. My new career plans mean it is unlikely that I will be working much with JMS in the future, so I will have little motivation to maintain and extend Config4JMS. If any readers would like take over maintenance of Config4JMS, then that would be great. Please let me know if you are interested in taking on this responsibility.

11.7 Summary

In this chapter, I have provided an overview of the architecture of Config4JMS. The "lots of power from a small class library" feel of Config4JMS is due to a synergy between its use of Config4J and Java's reflection

capabilities. This synergy enables Config4JMS to provide a useful simplification and portability wrapper around JMS with a relatively small amount of code.

The Config4J schema language is not powerful enough to validate an entire Config4JMS file in one go. However, Config4JMS works around this in a straightforward manner: as I discussed in Section 11.4.2 on page 105, it breaks up schema validation into a sequence of smaller steps, each of which *is* within the capabilities of the schema language.

Bibliography

[FF05] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning, 2005.

[Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 101(2):343–352, 1956. A summary of this paper can be found in Wikipedia: http://en.wikipedia.org/wiki/The_Magical_Number_Seven, _Plus_or_Minus_Two. That Wikipedia article also contains links to HTML and PDF versions of the paper.