SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 01-12

API Documentation with Executable Examples

Daniel Hoffman Paul Strooper

October 2001

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

API Documentation with Executable Examples

Daniel Hoffman* Paul Strooper†

Abstract

The rise of component-based software development has created an urgent need for effective API documentation. Experience has shown that it is hard to create precise and readable documentation. Prose documentation can provide a good overview but lacks precision. Formal methods offer precision but the resulting documentation is expensive to develop. Worse, few developers have the skill or inclination to read formal documentation.

We present a pragmatic solution to the problem of API documentation. We augment the prose documentation with executable test cases, including expected outputs, and use the prose plus the test cases as the documentation. With appropriate tool support, the test cases are easy to develop and read. Such test cases constitute a completely formal, albeit partial, specification of input/output behavior. Equally important, consistency between code and documentation is demonstrated by running the test cases. This approach provides an attractive bridge between formal and informal documentation. We also present a tool that supports compact and readable test cases, and generation of test drivers and documentation, and illustrate the approach with detailed case studies.

1 Introduction

With the growth of component-based software development approaches, the importance of Application Program Interface (API) documentation has grown as well. Class libraries and frameworks provide large and complex APIs, making effective documentation essential for successful use. While the method names and prototypes are expressed in the implementation language, the method behavior must be documented as well. Typically, this is done with brief prose descriptions, focusing on the situations that commonly arise in API use. Such documentation is inevitably imprecise and incomplete, leading to costly misunderstandings between API implementors and API users. The formal methods community recommends precise specifications, because such specifications can be complete and unambiguous. In some cases, the specifications can also be used to generate implementations or test oracles. Unfortunately such specifications are expensive to write and maintain. Worse, few developers are willing or able to read formal specifications.

*Dept. of Computer Science, University of Victoria, PO Box 3055 STN CSC, Victoria, B.C. V8W 3P6, Canada, dhoffman@csr.uvic.ca

†School of Inf. Tech. and Elec. Eng., Software Verification Research Centre, The University of Queensland, Brisbane, Qld. 4072, Australia, pstroop@itee.uq.edu.au

We present a pragmatic scheme for overcoming the problems of prose and formal specifications. The underlying idea is simple: augment traditional prose documentation with test cases designed specifically for use in documentation. Typically, there are a few cases for each likely question about API behavior. In practice, the test cases serve roughly the same role that FAQs ("frequently asked questions") do on many web sites.

Our "FAQ approach" to using test cases for documentation has four main benefits:

1. *Precise (though partial) documentation.* The test cases contain both inputs and expected outputs in executable form. Therefore, they are formal specifications of required behavior for selected inputs.

2. *Guaranteed consistency of code and documentation.* A single command can run all the test cases, automatically revealing inconsistencies between actual and documented behavior.

3. *Good fault detection.* While the primary purpose of the FAQ test cases is communication, they are also useful for quality assurance. For example, the test cases can provide the kind of unit tests advocated in Extreme Programming (Beck 1999a).

4. *Helpful examples of use.* When first using an API, programmers often spend a lot of time getting the first simple example to run. Our test cases provide complete, runnable examples suitable for copying and editing.

With our approach to test cases as documentation, readability of the test cases is of paramount importance. The next section shows how we develop compact, readable test cases with the *Roast* tool (Daley, Hoffman, and Strooper 2000; Hoffman and Strooper 2000). Sections 3 and 4 present detailed case studies of the FAQ approach, including documentation using test cases and, for comparison, in Z. Section 5 presents related work.

2 Tool support

To illustrate the benefits of tool support for FAQ test cases, we present a conventional test driver and a *Roast* driver (Daley, Hoffman, and Strooper 2000; Hoffman and Strooper 2000). Consider the test cases and output shown in Figure 1 for the Java `StringBuffer` class, which is part of Sun's JDK (Sun Microsystems 2001) and implements a mutable sequence of characters. In Figure 1(a), the first two lines of method `main` initialize the `StringBuffer s` and display the initial value. Test cases 1–4 show what happens when characters are inserted at the boundary positions: $\{-1, 0, \texttt{s.length()}, \texttt{s.length()} + 1\}$. As the output shows, the first and last of these positions are illegal. Some users are surprised to see that case 3 is legal, i.e., `s.insert(s.length(),`c`)` is equivalent to `s.append(`c`)`. When `StringBufferTest` is compiled and executed, it produces the output shown in Figure 1(b).

The driver in Figure 1 is reasonably compact, but is clumsy as a communication mechanism. The reader must jump back and forth between the method calls in the driver code and the driver output to determine the behavior for each case. Also, the only exception checking that is performed during test execution is that the calls to `insert` in cases 1 and

```
StringBuffer s = new StringBuffer("abc");
System.out.println("Starting value: " + s);

try { s.insert(-1,'W'); } // CASE 1
catch (Exception x) { System.out.println("Exception: case 1" ); }

s.insert(0,'X');          // CASE 2
System.out.println("Following case 2: " + s);

s.insert(s.length(),'Y');          // CASE 3
System.out.println("Following case 3: " + s);

try { s.insert(s.length()+1,'Z'); }  // CASE 4
catch (Exception x) { System.out.println("Exception: case 4" ); }
```

(a) Driver source code

```
Starting value: abc
Exception: case 1
Following case 2: Xabc
Following case 3: XabcY
Exception: case 4
```

(b) Driver output

Figure 1: `StringBufferTest` source code and output

4 throw an exception: which exception is not indicated. We could augment the driver to include code to perform additional checking, but this would make it bulky and unsuitable for documentation purposes.

We next introduce the *Roast* test driver generator and show how *Roast* test case templates can be used to compactly define the test cases shown in Figure 1.

2.1 Test case templates

Roast test case templates are embedded in Java test drivers and are identified by keywords preceded by the `#` character[1]. There are two types of *Roast* test cases: value-checking and exception-monitoring. The form of a value-checking test case is:

`#valueCheck` *actualValue* `#` *expectedValue* `#end`

where *actualValue* and *expectedValue* are expressions of the same type. For such a test case template, *Roast* generates code to compare *actualValue* and *expectedValue*, while monitoring the exception behavior. The generated code prints an error message if *actualValue* and *expectedValue* are different or if an exception is thrown during the comparison, and is silent otherwise.

The general form of an exception-monitoring test case is:

`#excMonitor` *action* `#` *expectedException* `#end`

where *action* is any fragment of Java code and *expectedException* is a Java exception. *Roast* generates code to execute *action*, while monitoring the exception behavior. The generated code prints an error message if *expectedException* is not thrown or if another exception is thrown. In an exception-monitoring test case template *expectedException* can be omitted, in which case an error message is printed if any exception is thrown.

The above templates are a generalized form of assertion, as found in languages such as C++ and Eiffel. The templates are designed for use in test drivers rather than for use in implementations, which is how assertions are typically used. The templates are more general in that they perform exception checking, and they allow comparison of two arbitrary values rather than simply checking for boolean conditions. As a result, meaningful error messages are generated containing the values of *actualValue* and *expectedValue*.

Roast test cases corresponding to the test cases shown in Figure 1 are shown in Figure 2. The test cases are more readable than in Figure 1 and the exception-checking test cases are more compact. No output file is needed because the inputs and expected outputs are contained side-by-side in the driver code and compared by *Roast* at test execution time.

2.2 Documentation generation

The flowchart in Figure 3 shows how code and documentation are generated for class `C`. The file `C.script` contains the source code, prose documentation, and *Roast* test cases. *Roast* generates `Driver.java` by expanding each `#valueCheck` and `#excMonitor` template; typically 10–15 lines of Java code are generated for each case. The file `C.java` contains

[1] Although it is possible to specify test cases as syntactically valid Java code, without using embedded test cases, this is clumsy and leads to test drivers that are hard to read and maintain.

```
StringBuffer s = new StringBuffer("abc");

// CASE 1
#excMonitor s.insert(-1,'W'); # new StringIndexOutOfBoundsException() #end

// CASE 2
#excMonitor s.insert(0,'X'); #end    #valueCheck s # "Xabc" #end

// CASE 3
#excMonitor s.insert(s.length(),'Y'); #end    #valueCheck s # "XabcY" #end

// CASE 4
#excMonitor s.insert(s.length()+1,'Z'); # new StringIndexOutOfBoundsException() #end
```

Figure 2: *Roast* `Stringbuffer` test script

the source code and prose documentation, and HTML links to the test cases. Javadoc is used to generate HTML suitable for browsing, including both the prose and the test cases. `C.java` and `Driver.java` are compiled and run, to ensure that `C` behaves as indicated in the test cases.

In FAQ documentation, a series of questions are posed and then linked to test cases, like those in Figure 2, that answer the questions. We have found that posing good questions takes experience; writing the corresponding test cases is easy. This approach is illustrated in detail in the following two sections.

3 StringBuffer case study

To illustrate the FAQ approach, we document the `replace` method from the Java `StringBuffer` class. We compare the API documentation for `replace` with the same documentation augmented with test cases, and with a Z specification (Spivey 1992).

3.1 API documentation

Figure 4 shows the API documentation for `replace`. The call s`.replace(`$start$`,`end`,`r`)` modifies the source string s by removing the substring $s[start\,..\,end-1]$ and inserting the replacement string r at position $start$. Although `replace` seems straightforward, there are a few subtle points. The substring is identified by the half-open range $[start, end)$, familiar to users of the C++ Standard Template Library (Musser and Saini 1996), but often confusing to others. In the special case where $start = end$, the substring is empty, but it is not entirely clear at what position the replacement string will be inserted. Finally, the situations where $start$ and end are out of range are handled asymmetrically. The API documentation can easily be clarified with a few concrete examples.

3.2 FAQs in test case form

Typical questions that users might have about the behavior of `replace` are:

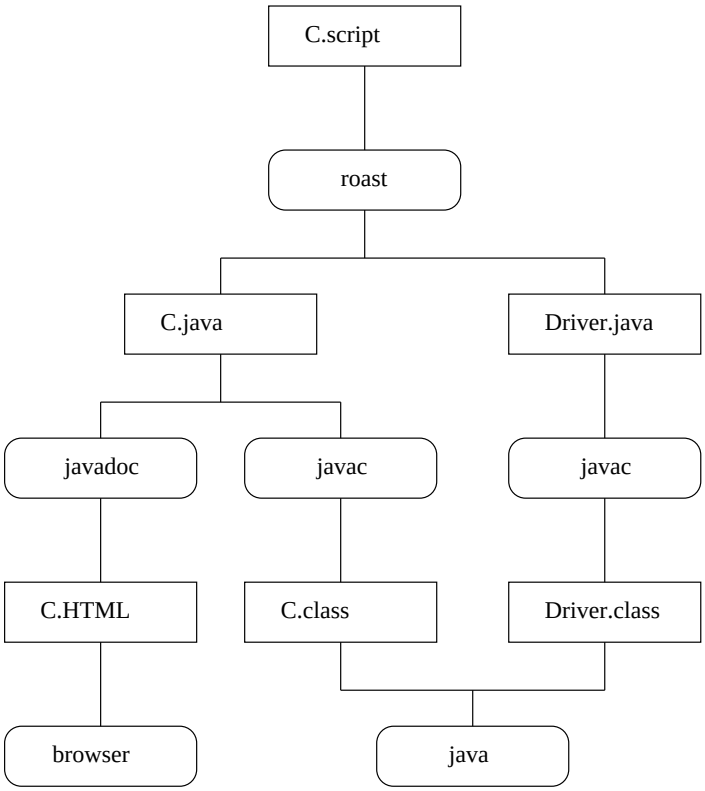

Figure 3: *Roast* system flowchart

`public StringBuffer` **`replace`**`(int start,int end,String str)`

Replaces the characters in a substring of this `StringBuffer` with characters in the specified `String`. The substring begins at the specified `start` and extends to the character at index `end` − 1 or to the end of the `StringBuffer` if no such character exists. First the characters in the substring are removed and then the specified `String` is inserted at `start`. (The `StringBuffer` will be lengthened to accommodate the specified String if necessary.)

Parameters:

`start` - The beginning index, inclusive.
`end` - The ending index, exclusive.
`str` - String that will replace previous contents.

Returns:

This string buffer.

Throws:

StringIndexOutOfBoundsException - if `start` is negative, greater than `length`(), or greater than `end`.

Figure 4: `StringBuffer` API documentation for the `replace` method

1. What start values are legal?
2. What end values are legal?
3. Can the source string be empty?
4. Can the replacement string be empty?

Figure 5 shows test cases that answer these questions. The first question is answered with four test cases. The first and fourth show the exception that is thrown when *start* is outside the range $[0\,..\,s.length()]$. The second and third cases show the effect at the boundaries of this range. The second question is answered by four test cases, showing that *end* may have any value greater than or equal to *start* and that a value of *end* larger than the length of s is treated the same as one equal to the length of s. The third and fourth questions are answered in the positive, each with a simple case showing the effect.

Using the approach shown in Figure 5, we documented 13 out of the 34 `StringBuffer` methods. Each of the 21 methods that we did not document was a simple variation of one of the methods that we did document. For example, there are 10 versions of `insert` that vary only in the type of element that is inserted (`char`, `int`, etc.), and we only documented one of these. For each of the methods we documented, we added 3–10 test cases to the API documentation. In doing so, we discovered a surprising number of problems. For example, the API documentation for `insert` states that `StringIndexOutOfBoundsException` is thrown if the offset is invalid, but in fact `ArrayIndexOutOfBoundsException` is thrown. As a result, the test driver shown in Figures 2 generates a failure message for each of the `#excMonitor` test cases. In the 13 methods tested, we found 10 such inconsistencies in the documentation of the exception behavior. In addition, the API documentation for one of the methods (`substring`) is clearly incomplete, as one of the sentences ends half-way through.

```
StringBuffer s = null;
Exception BoundsException = new StringIndexOutOfBoundsException();

// What start values are legal?
s = new StringBuffer("abcde");
    #excMonitor s.replace(-1,1,"XYZ"); # BoundsException #end
s = new StringBuffer("abcde"); s.replace(0,2,"XYZ");
    #valueCheck s # "XYZcde" #end
s = new StringBuffer("abcde"); s.replace(s.length(),s.length()+2,"XYZ");
    #valueCheck s # "abcdeXYZ" #end
s = new StringBuffer("abcde");
    #excMonitor s.replace(s.length()+1,s.length()+3,"XYZ"); # BoundsException #end

// What end values are legal?
s = new StringBuffer("abcde"); s.replace(3,s.length(),"XYZ");
    #valueCheck s # "abcXYZ" #end
s = new StringBuffer("abcde"); s.replace(3,3,"XYZ");
    #valueCheck s # "abcXYZde" #end
s = new StringBuffer("abcde"); s.replace(3,s.length()+100,"XYZ");
    #valueCheck s # "abcXYZ" #end
s = new StringBuffer("abcde");
    #excMonitor s.replace(2,1,"XYZ"); # BoundsException #end

// Can the source string be empty?
s = new StringBuffer(); s.replace(0,0,"XYZ");
    #valueCheck s # "XYZ" #end

// Can the replacement string be empty?
s = new StringBuffer("abcde"); s.replace(1,3,"");
    #valueCheck s # "ade" #end
```

Figure 5: `replace` FAQ test cases

3.3 Z specification

To compare the prose description and test cases with a formal specification, we now present a Z specification of `replace`. We assume the reader is familiar with the basics of the Z notation (Spivey 1992).

Since Z sequences are indexed starting from 1, we first define the type seq_0 to represent sequences starting at index 0 (in Z, we define this as a finite, partial function whose domain is a segment $0 \mathinner{..} n$ for some natural number n).

$$\text{seq}_0\, X == \{f : \mathbb{N} \pfun X \mid \text{dom}\, f = 0 \mathinner{..} \#f - 1\}$$

We model the state of the `StringBuffer` class using a Z schema as a sequence of characters.

State

$$str : \text{seq}_0\ Char$$

To define `replace`, we will also use versions of the Z mathematical toolkit (Spivey 1992) operations $\frown$ (concatenation) and *squash*, except that we need to define them for sequences starting at index 0 instead of index 1. The function *squash* takes a finite function defined on the natural numbers and compacts it into a sequence. The definitions are:

$[X]$

$$\_ \frown \_ : seq_0 X \times seq_0 X \to seq_0 X$$
$$squash : (\mathbb{N} \pfun X) \to seq_0 X$$

$$\forall s, t : \text{seq}_0\, X \bullet$$
$$\quad s \frown t = s \cup \{n : \text{dom}\, t \bullet n + \#s \mapsto t(n)\}$$
$$\forall f : \mathbb{N} \pfun X \bullet$$
$$\quad squash\, f = f \circ (\mu\, p : 0 \mathinner{..} \#f - 1 \rightarrowtail \text{dom}\, f \mid p \circ succ \circ p^{\sim} \subseteq (\_ \leq \_))$$

With these definitions, we can specify the `replace` operation.

replace

$$State$$
$$State'$$
$$start?, end? : \mathbb{Z}$$
$$newStr? : \text{seq}_0\ Char$$

$$(start? < 0) \lor (start? > \#str) \lor (start? > end?)$$
$$\quad \longrightarrow StringIndexOutOfBoundsException$$

$$str' = ((0 \mathinner{..} start? - 1) \lhd str) \frown newStr? \frown squash((end? - 1 \mathinner{..} (\#\,\text{dom}\, str) - 1) \lhd str)$$

For brevity, we have abused the Z notation. To specify exceptions, we have added an "exception part" between the declaration and the predicate part of the schema. Each statement in the exception part consists of a condition and an exception that is to be thrown when that condition is true. When none of the exception conditions are true,

the predicate part of the schema is applicable. This approach to specifying exceptions has been taken from (McDonald and Strooper 1998) and can be translated in a straightforward manner to standard Z.

Although the Z specification is concise, it is non-trivial, especially the use of *squash* to ensure that the indices of the third sequence appended to the result are correct. In addition, we note that the above specification does not fully specify the behavior of the Java implementation, which is partly because we have not defined the mapping from our formal specification language (Z) to our implementation language (Java). In particular:

- there is no support in Z for defining exceptions,
- we have not modeled the return value of the function, which is a reference to the string buffer object itself (there is no convenient way to model this in Z), and
- we have not modeled the fact that the string buffer size will be changed.

4 Command line case study

This section presents a case study based on a Java module for processing Unix command-line arguments. Although modern GUIs have made command-line interfaces old-fashioned, they are still in widespread use, especially by system and network administrators.

This section is based on one solution to the command-line problem. The value of this solution is its concreteness: it has been thoroughly documented, implemented and tested. In so doing, we made many decisions about module behavior. We expect that most readers would have made some of those decisions differently. Our focus here, however, is on how to document decisions about module behavior, not on the decisions themselves.

4.1 Command Line module overview

In Unix, arguments are entered on the command line, processed by the shell, and passed to a Java `main` method as an array of strings. For example, a user might enter

```
lpr -P rp -p foo
```

to request that file `foo` be sent to the printer queue `rp`. The `-p` flag specifies that a standardized header be placed on each page of output. The array passed to `main` will have the following value:

```
{"-P", "rp", "-p", "foo"}
```

The Command Line module offers a generic service for parsing command-line arguments, for use by programmers developing Java applications. The argument array contains zero or more *flags* followed by zero or more *suffix arguments*. A flag can be any string beginning with '-'. Some flags are optional and others are required. Some flags take a *flag argument*; others do not. Often there are restrictions on the flag argument type, e.g., from 1 to 3 decimal digits. The suffix arguments (typically filenames) are always optional and have no type restrictions.

In the example above, `"-P"` is a flag with flag argument `"rp"`, `"-p"` is a flag with no flag arguments, and `"foo"` is a suffix argument.

The Command Line user will specify, for each legal flag:

- flag name, e.g., -f,
- flag required or optional,
- flag argument: required or prohibited, and
- flag argument type:
  - `INTEGER`, and maximum length,
  - `FIXEDPOINT`, and maximum lengths to the left and right of the decimal point,
  - `ALPHA`, and maximum length, or
  - `ANY`, and maximum length.

The command-line arguments will be passed to the Command Line module as a `String` array. If the arguments satisfy the user specification, then access is provided to the flags and arguments present. Otherwise, an error message is made available.

The function prototypes for four classes in the Command Line module are shown in Figure 6. In the `CommandLine` class, the constructor takes an array of flag specifications and an array of argument strings. The method `isValid` returns true if the argument strings satisfy the flag specifications. Otherwise, `getErrorMessage` returns a suitable message. The call `isArgPresent(`f`)` returns true if flag f was present; `getArgFlag(`f`)` returns the flag argument following f. Finally, `getSuffixArgs` returns all the suffix arguments.

In the `FlagSpec` class, the constructor takes the four values needed to specify a flag. The fourth field is of type `ArgType`, an abstract class. An `ArgType` subclass must implement `isValid`, which takes a string that represents an argument and returns true (false) indicating that the string is (is not) a valid argument of that type. In the `IntegerArgType` class, the constructor takes a single integer n and `isValid(`s`)` returns true if s consists of from 1 to n decimal digits.

The other classes in the Command Line module—the exception classes, `FixedPointArgType`, `AlphabeticArgType`, and `AnyArgType`—have been omitted for brevity.

4.2 FAQs in test case form

Given the method prototypes and prose description just presented, many questions remain about the Command Line module behavior:

1. How do you find out what was on the command line?
   (a) Which flags were present?
   (b) What were the arguments to the flags?
   (c) What were the suffix arguments?
2. Is the flag order significant?

```
public class CommandLine {
   public CommandLine(FlagSpec[] flagSpec, String[] args)
      throws ParameterException;
   public boolean isValid();
   public String getErrorMessage() throws ValidArgsException;
   public boolean isArgPresent(String flagName) throws ParseException;
   public String getFlagArg(String flagName)
      throws ParseException, FlagNotPresentException, NoArgException;
   public String[] getSuffixArgs() throws ParseException;
}

public class FlagSpec {
   public FlagSpec(String flagName, boolean isRequired,
      boolean argRequired, ArgType argType);
   public String flagName;
   public boolean isRequired, argRequired;
   public ArgType argType;
}

abstract class ArgType {
   abstract boolean isValid(String s);
}

class IntegerArgType extends ArgType {
   public IntegerArgType(int maxLength);
   public boolean isValid(String s);
}
```

Figure 6: Command Line module: function prototypes

3. Are there any constraints on the number of suffix arguments?
4. Are there any constraints on the value of a suffix argument?
5. What if the arguments have errors?
   (a) How is the error status communicated?
   (b) What other information is available about the arguments?
6. What if a required flag is omitted?
7. What if a flag is repeated?

The answers to these questions are not obvious because each question has multiple defensible answers. We saw many such answers in the implementations of particular Unix commands and in other generic command-line modules we found on the web. Figure 7 contains portions of a driver that provides answers to these questions.

The driver begins by creating specifications for three flags:

1. `-a`: optional, with no flag argument
2. `-b`: required, with no flag argument
3. `-c`: optional, with an `INTEGER` argument, of maximum length 3

Then, there are blocks of test cases for questions 1, 4, and 5 in the list presented earlier. The first block covers the typical uses by showing how to determine which flags were present, the value of the flag arguments, and the values of the suffix arguments.

The second block of test cases handles questions about suffix arguments. There is significant ambiguity here regarding the rules for distinguishing flag and suffix arguments. The first case shows that, even though `-y` has a leading "-", it is interpreted as a suffix argument. This follows the common policy in Unix commands that all arguments following the first suffix argument (`x` in this case) are interpreted as suffix arguments. The second case shows that this policy is followed even though `-b` is a declared flag. The third case shows that the value of a suffix argument need not resemble a typical file name.

The third block of test cases shows what happens when the command line is in error: `isValid` is false, `getErrorMessage` is non-null, and attempts to provide information about why the arguments are refused. The detailed error message returned by `getErrorMessage` is not tested, since the message itself is not important and likely to change.

Note that Figure 7 contains tests cases for only the CommandLine class; separate cases (very simple ones) are needed for the `Argtype` classes.

While the driver illustrated in Figure 7 focuses on communicating module behavior to the user, it has value in defect detection as well. The full driver contains 86 lines of code and executes 34 test cases, achieving 86.1% statement coverage of the `CommandLine` class. For comparison purposes, we wrote another driver, taking full advantage of all the *Roast* features. This driver is aimed solely at finding defects. It generates argument arrays of varying lengths and places legal, illegal, required, and optional flags, with and without arguments, at boundary positions in each argument array. This driver is complex, especially

```
FlagSpec[] flagSpecs = {
new FlagSpec("-a",false,false,null), //flag,isReq,argReq,argType
new FlagSpec("-b",true,false,null),
new FlagSpec("-c",false,true,new IntegerArgType(3))
};
CommandLine cut = null;

// ***** How do you find out what was on the command line?
#excMonitor cut = new CommandLine(flagSpecs,
    new String[] { "-b","-c","5","suffixArg" }); #end
#valueCheck cut.isArgPresent("-a") # false #end
#valueCheck cut.isArgPresent("-b") # true #end
#valueCheck cut.isArgPresent("-c") # true #end
#valueCheck cut.getFlagArg("-c") # "5" #end
#valueCheck (cut.getSuffixArgs())[0] # "suffixArg" #end

// ***** Are there any contraints on the value of a suffix argument?
#excMonitor cut = new CommandLine(flagSpecs,
    new String[] { "-b","x","-y" }); #end
#valueCheck cut.isValid() # true #end

#excMonitor cut = new CommandLine(flagSpecs,
    new String[] { "-b","x","-b" }); #end
#valueCheck cut.isValid() # true #end

#excMonitor cut = new CommandLine(flagSpecs,
    new String[] { "-b","x","!$ %" }); #end
#valueCheck cut.isValid() # true #end

// ***** What if the arguments have errors?
#excMonitor cut = new CommandLine(flagSpecs,
    new String[] { "-a","-c","5" }); #end
#valueCheck cut.isValid() # false #end
#valueCheck (cut.getErrorMessage() != null) # true #end
#excMonitor cut.isArgPresent("-a"); # new ParseException() #end
```

Figure 7: Command Line Driver: answers to selected FAQs

the test oracle. It contains 261 lines of code and executes 307 test cases, achieving 91.1% statement coverage. While statement coverage is a crude measure of test effectiveness, the coverage numbers suggest that FAQ drivers can be useful in defect detection.

4.3 A specification in Z

To compare the prose description (see Section 4.1) and test cases with a formal specification, we again present a Z specification of the Command Line module.

We define *String* as a sequence of characters.

$$String == \text{seq}\, Char$$

We model the `FlagSpec` and `ArgType` classes (see Figure 6) as Z schemas.

$$\begin{array}{|l} \textit{FlagSpec} \\ \hline name : String \\ isReq, hasArg : \mathbb{B} \\ arg : ArgType \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{ArgType} \\ \hline isValid : String \to \mathbb{B} \\ \hline \end{array}$$

Next we define the specification state of the `CommandLine` class.

$$\begin{array}{|l} \textit{State} \\ \hline errorMsg : String \\ flags : \mathbb{F}\, String \\ argFlags : String \pfun String \\ suffix : \text{seq}\, String \\ \hline \text{dom}\, argFlags \subseteq flags \\ \hline \end{array}$$

The state contains four components: *errorMsg* stores the error message generated, or the empty string ($\langle\rangle$) to indicate that there was no error; *flags* stores the set of all flags; *argFlags* stores the set of flag arguments as a partial function from flags to their arguments (for a flag f with argument a, f is in the domain of *argFlags* and $argFlags(f) = a$); and *suffix* stores the suffix arguments as a sequence of strings.

We can then model the `CommandLine` constructor.

$$
\begin{array}{|l}
\textit{CommandLine} \\
\hline
fs? : \operatorname{seq} FlagSpec \\
args? : \operatorname{seq} String \\
State' \\
\hline
fs? = null \lor args? = null \longrightarrow ParameterException \\
\exists\, i : \operatorname{dom} args? \bullet args?(i) = \langle\,\rangle \longrightarrow ParameterException \\
\exists\, i, j : \operatorname{dom} fs? \bullet i \neq j \land fs?(i).name = fs?(j).name \longrightarrow ParameterException \\
\hline
\textbf{let} \\
\quad (f == \{i : \operatorname{dom} fs? \bullet fs?(i).name\}; \\
\quad\ rF == \{i : \operatorname{dom} fs? \mid fs?(i).isReq = true \bullet fs?(i).name\}; \\
\quad\ aF == \{i : \operatorname{dom} fs? \mid fs?(i).hasArg = true \bullet fs?(i).name \mapsto fs?(i).arg\} \\
\bullet\ \theta State' = parseFlag(f, rF, aF, args?, \varnothing, \varnothing))
\end{array}
$$

We have abused the Z notation in that we have used the value *null* to model a null pointer; note that this is not the same as the empty sequence ($\langle\,\rangle$). To model this properly in Z, we would have to use a free type.

The exception part of the specification states that *ParameterException* is thrown if either input parameter is a null pointer, if there is a command-line argument that is the empty string, or if there are two flag arguments in the flag specification that have the same name. In the predicate part, we first construct three sets: f, the set of all flags in the flag specification, rF, the set of required flags, and aF, the set consisting of mappings from flags with arguments to their argument type. The predicate part of the specification is defined using the recursive function *parseFlag*, which is defined in Figure 8. It takes the three sets, the command-line arguments, and partially constructed sets of flags and argument flags as inputs, and returns the class state as its output. Note that the auxiliary arguments (the fifth and sixth arguments to *parseFlag*) are used to construct the state incrementally.

Initially, the sets of flags and argument flags are empty. Each recursive call removes one or two arguments from the list of arguments, depending on whether or not the next flag has an argument, augmenting the partially constructed sets of flags and flag arguments. The recursion terminates when an error is discovered or when the first non-flag argument is encountered. In the latter case, all the remaining arguments are returned as suffix arguments.

With the above specifications for the state schema and the `CommandLine` constructor, the specification for the other class methods is straightforward and shown in Figure 9.

4.4 Discussion

The comparison clearly shows that the Z specification is complex and would be hard to understand by people with little training in formal methods. People with training in formal methods might prefer it over the prose documentation, because it provides a complete specification of the behavior of the Command Line module. However, people that reviewed the formal specification and the test cases commented that the test cases helped them with understanding the specification because they provided concrete examples

$$
\begin{array}{l}
parseFlag : \mathbb{F}\, String \times \mathbb{F}\, String \times (String \pfun ArgType) \times \mathrm{seq}\, String \times \mathbb{F}\, String \\
\quad \times (String \pfun String) \rightarrow State \\
\hline
\forall f, rf, f1 : \mathbb{F}\, String;\ af : (String \pfun ArgType);\ args : \mathrm{seq}\, String; \\
\quad af1 : (String \pfun String);\ out : State \bullet \\
((f, rf, af, args, f1, af1), out) \in parseFlag \Leftrightarrow \\
\quad \textbf{if}\ (args \neq \langle\rangle \land args(1)(1) = \text{`-'})\ \textbf{then} \\
\qquad \textbf{if}\ args(1) \notin f\ \textbf{then} \\
\qquad\quad out.errorMsg = INVALIDFLAG \frown args(1) \\
\qquad \textbf{else if}\ args(1) \in f1\ \textbf{then} \\
\qquad\quad out.errorMsg = DUPLICATEFLAG \frown args(1) \\
\qquad \textbf{else if}\ (args(1) \in \mathrm{dom}\, af)\ \textbf{then} \\
\qquad\quad \textbf{if}\ \#args = 1\ \textbf{then} \\
\qquad\qquad out.errorMsg = MISSINGFLAGARG \frown args(1) \\
\qquad\quad \textbf{else if}\ af(args(1)).isValid(args(2)) = false\ \textbf{then} \\
\qquad\qquad out.errorMsg = INVALIDFLAGARG \frown args(2) \\
\qquad\quad \textbf{else} \\
\qquad\qquad out = parseFlag(f, rf, af, tail(tail(args)), \\
\qquad\qquad\quad f1 \cup \{args(1)\}, af1 \cup \{args(1) \mapsto args(2)\}) \\
\qquad \textbf{else} \\
\qquad\quad out = parseFlag(f, rf, af, tail(args), f1 \cup \{args(1)\}, af1) \\
\quad \textbf{else} \\
\qquad \textbf{if}\ rf \subseteq f1\ \textbf{then} \\
\qquad\quad out.errorMsg = \langle\rangle \land out.flags = f1 \land out.argFlags = af1 \land out.suffix = args \\
\qquad \textbf{else} \\
\qquad\quad (\exists s : String \bullet s \in rf \setminus f1 \land \\
\qquad\qquad out.errorMsg = REQUIREDFLAGMISSING \frown s)
\end{array}
$$

Figure 8: Definition of *parseFlag*

isValid

$valid! : \mathbb{B}$

$State$

$valid! = true \Leftrightarrow errorMsg = \langle\rangle$

getErrorMessage

$m! : String$

$State$

$errorMsg = \langle\rangle \longrightarrow ValidArgsException$

$m! = errorMsg$

isArgPresent

$flagname? : String$

$present! : \mathbb{B}$

$State$

$errorMsg \neq \langle\rangle \longrightarrow ParseException$

$present! = true \Leftrightarrow flagname? \in \operatorname{dom} argFlags$

getFlagArg

$flagname? : String$

$arg! : String$

$State$

$errorMsg \neq \langle\rangle \longrightarrow ParseException$

$flagname? \notin flags \longrightarrow FlagNotPresentException$

$flagname? \in flags \setminus \operatorname{dom} argFlags \longrightarrow NoArgException$

$arg! = argFlags(flagname?)$

getSuffixArgs

$arg! : \operatorname{seq} String$

$State$

$errorMsg \neq \langle\rangle \longrightarrow ParseException$

$arg! = suffix$

Figure 9: Z specification of `CommandLine` methods

of the use of the module before trying to understand the module in its full generality. This suggests that test cases can not only be useful in augmenting prose documentation, but also to augment and clarify formal specifications.

5 Related work

The use of examples in documentation is an old idea. Today, use cases (Jacobsen 1992) are probably the best known technique for software documentation based on examples. While use cases are usually informal and not executable, they can be made executable, as research on SCR requirements documents has shown (Miller 1998). Our test cases can be thought of as executable API use cases.

Hsia et al. present a systematic, formal method for scenario analysis that supports requirements analysis and change, and acceptance testing (Hsia, Gao, Samuel, Kung, Toyoshima, and Chen 1994b). The method is extended to serve as a starting point for a formal model for scenario-based acceptance testing (Hsia, Gao, Samuel, Kung, Toyoshima, and Chen 1994a; Hsia, Kung, and Sell 1997). The systematic approach allows a set of complete and consistent scenarios to be derived for acceptance testing. Similarly, Chang et al. describe a method for generating test scenarios for integration and system testing from formal, Object-Z specifications and usage profiles (Chang, Liao, Seidman, and Chapman 1998; Chen, Chang, and Chapman 1999).

Using test cases in documentation involves test case selection, a central topic in testing research (White and Cohen 1980; Weyuker and Ostrand 1980; Richardson and Clarke 1985). Our approach is also consistent with proposals for extreme programming (Beck 1999b; Beck 1999a), where API test cases play a central role (Jeffries 1999). Like *Roast*, the JUnit testing framework (Fowler 1999) supports the testing of Java classes and and has been applied in a number of application domains, including Enterprise JavaBeans (Nygard and Karsjens 2000).

In an approach similar to ours, Deveaux et al. (Deveaux, Frison, and Jézéquel 2001) combine embedded textual documentation and semi-formal specification to support self-testable classes in Java (the same approach has also been applied to Eiffel). The main difference between the two approeaches is that the tests in our approach are included mainly for documentation purposes (which means that readability is a prime concern), whereas in their approach the tests are used primarily for verification and validation. Another difference is that their approach is based around semi-formal specifications using design by contract (Jézéquel and Meyer 1997; Meyer 1997).

Techniques for programming by example have long been studied in the artificial intelligence research community. For example, Winston (Winston 1975) examines the importance of "hit" and "near miss" examples in machine learning. In this AI work, however, a *machine* generalizes from examples, while our goal is to get *humans* to generalize from examples.

Engelmann and Carnine (Engelmann and Carnine 1991), provide an extensive treatment of how to select examples and counter-examples to produce a chosen generalization in the mind of the reader. They emphasize efficiency—using as few examples as possible—and accuracy—choosing examples to minimize the probability of misunderstanding. Their

work is directly relevant to ours because the goals are the same: precise communication with humans of a general rule through a small number of specific examples.

There is considerable argument as to whether formal methods require mathematical sophistication. Some argue that the mathematics for specification is easy (Hall 1990), while others argue that this is not quite the case (Finney 1996). The only substantial experimental study that we are aware of is (Finney, Rennolls, and Fedorec 1998), which evaluated the effects of natural language comments, variable naming, and structuring on the comprehensibility of Z specifications. Kneuper correctly points out that it is not only the ability of the developers to use formal methods that needs to be considered, but also their willingness to do so (Kneuper 1997). We concur and note that while it is unlikely that formal specifications will be used for API documentation in the next 5–10 years, the test cases that we have presented are formal, partial specifications that are easily understood by developers.

Finally, we note that our mixing of prose, test cases, and code, and the processing of these, contains some similarity with literate programming (Knuth 1984; Knuth 1992), although the details and motivation are quite different. With literate programming, the purpose of the mixing of documentation and code is to allow humans to better understand how the program is implemented. With our approach, the purpose is to allow humans to better understand what the program is supposed to do.

6 Summary

6.1 Discussion

Despite occasional claims to the contrary, a set of examples is rarely a complete specification, for the same reason that testing cannot prove a program correct. There are significant advantages to a formal specification: precision, completeness, and machine processability to name a few. In particular, preconditions and nondeterminism are difficult to express with test cases. Nonetheless, it is important to recognize the role that examples can play and, in fact, have played for centuries in mathematics.

The most important difference between formal methods and our approach involves the goals.

- With formal methods, the goal is a complete description of the required behavior in all circumstances.
- With our approach, we envision a family of plausible behaviors determined by the method prototypes, the prose documentation, and the domain knowledge of the reader. The purpose of the test cases is to indicate which behavior in the family is the one actually provided.

These are radically different goals. If the domain knowledge of the reader is considerable, prose and test cases can be very effective. If it is not, formal methods may be superior. From the FAQ perspective, the formal specification attempts to answer every *possible* question while our approach attempts to answer every *likely* question.

The two approaches can be used together. Formal preconditions are often short and readable while postconditions are often long and complex. Thus, an effective hybrid might express preconditions formally and use prose plus test cases for postconditions. For example, the Eiffel libraries are documented using a mix of prose and formal notation (in the form of assertions) (Meyer 1994). The preconditions are often formal and complete, whereas the formal parts of the postconditions are typically partial, if present at all. Further, even if a formal specification is developed, the FAQ test cases can be helpful in explaining and testing the specification.

6.2 Conclusions

The rise of component-based software development has created an urgent need for effective API documentation. Prose documentation can provide a good overview but lacks precision. Formal methods offer precision but the resulting documentation is expensive to develop. Worse, few developers have the skill or inclination to read formal documentation. We present a pragmatic solution: augment the prose documentation with executable test cases and use the prose plus the test cases as the documentation. This approach provides an attractive bridge between formal and informal documentation.

Our "FAQ approach" to using test cases for documentation has four main benefits:

1. Precise (though partial) documentation.
2. Guaranteed consistency of code and documentation by running the test cases.
3. Good fault detection.
4. Helpful examples of API use.

This approach depends critically on the test cases being compact and readable. We have shown that, with a testing tool such as *Roast*, the test cases themselves can satisfy these properties.

Most important, our approach is ready for use today. While the FAQ approach to documentation is new, we have had considerable practical experience with writing automated test cases with Roast. We have written such cases in multiple languages, including C, C++, Ada, and Java, and in a variety of industrial domains, including container class libraries, safety-critical systems, and concurrent systems (Hoffman 1989; Hoffman and Strooper 1997; Hoffman, Nair, and Strooper 1998; Murphy, Townsend, and Wong 1994; McDonald, Hoffman, and Strooper 1998; Harvey and Strooper 2001; Long and Strooper 2001). We know the Roast tool is teachable because we have used it extensively in undergraduate teaching at the Universities of Queensland and Victoria. Students write test cases, and read cases we write in documentation and in exam questions. We have found that students learn to use Roast with minimal effort: after a few lectures or just simply using the manual and on-line examples.

Finally, we note that many recent text and reference books have adapted an FAQ style, to the extent that prose explanations are mixed with fully worked and runnable code. Some notable examples include the Standard Template Library Tutorial and Reference Guide (Musser and Saini 1996) and the Java Language Specification (Gosling, Joy, and Steele 1996).

Acknowledgements

Thanks to Nigel Daley for development of the Command Line code, Andrew Harcourt for his extensions to Roast to support FAQ specifications, Ian Hayes and Alena Griffiths for their suggestions on previous versions of the Z specification for Command Line, and David Hemer, Tim Miller, and Hagen Völzer for their constructive comments on earlier versions of this paper.

References

Beck, K. (1999a, October). Embracing change with extreme programming. *Computer*, 70–77.

Beck, K. (1999b). *Extreme Programming Explained*. Addison-Wesley.

Chang, K., S.-S. Liao, S. Seidman, and R. Chapman (1998). Testing object-oriented programs: from formal specification to test scenario generation. *The Journal of Systems and Software 42*(2), 141–151.

Chen, C.-Y., K. Chang, and R. Chapman (1999). Test scenario and test case generation based on Object-Z formal specification. In *Proceedings of SEKE'99*, pp. 207–211.

Daley, N., D. Hoffman, and P. Strooper (2000). Unit operations for automated class testing. Technical Report 00–04, Software Verification Research Centre, The Univ. of Queensland.

Deveaux, D., P. Frison, and J.-M. Jézéquel (2001). Increase software trustability with self-testable classes in java. In *Proceedings 2001 Australian Software Engineering Conference*, pp. 3–11. IEEE Computer Society.

Engelmann, S. and D. Carnine (1991). *Theory of Instruction: Principles and Applications* (second ed.). Eugene, Oregon: ADI Press.

Finney, K. (1996). Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering 22*(2), 158–159.

Finney, K., K. Rennolls, and A. Fedorec (1998). Measuring the comprehensibility of Z specifications. *The Journal of Systems and Software 42*(1), 3–15.

Fowler, M. (1999). *Refactoring – Improving the Design of Existing Code*, Chapter 4: Building Tests. Addison-Wesley.

Gosling, J., B. Joy, and G. Steele (1996). *The Java Language Specification*. Addison-Wesley.

Hall, J. (1990). Seven myths of formal methods. *IEEE Software 7*(9), 11–19.

Harvey, C. and P. Strooper (2001). Testing java monitors through deterministic execution. In *Proceedings 2001 Australian Software Engineering Conference*, pp. 61–67. IEEE Computer Society.

Hoffman, D. (1989, October). A CASE study in module testing. In *Proc. Conf. Software Maintenance*, pp. 100–105. IEEE Computer Society.

Hoffman, D., J. Nair, and P. Strooper (1998). Testing generic ada packages with APE. In *Proceedings ACM SIGAda Annual International Conference (SIGAda'98)*, pp. 255–262. ACM Press.

Hoffman, D. and P. Strooper (1997). ClassBench: A methodology and framework for automated class testing. *Software: Practice and Experience 27*(5), 573–597.

Hoffman, D. and P. Strooper (2000). Tools and techniques for Java API testing. In *Proceedings 2000 Australian Software Engineering Conference*, pp. 235–245. IEEE Computer Society.

Hsia, P., J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen (1994a). Behavior-based acceptance testing of software systems: A formal scenario approach. In *Proceedings of COMPSAC'94*, pp. 293–298. IEEE Computer Society Press.

Hsia, P., J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen (1994b). A formal approach to scenario analysis. *IEEE Software 11*(2), 33–41.

Hsia, P., D. Kung, and C. Sell (1997). Software requirements and acceptance testing. *Annals of Software Engineering 3*, 291–317.

Jacobsen, I. (1992). *Object-Oriented Software Engineering*. New York: Addison-Wesley.

Jeffries, R. (1999, March/April). Extreme testing. *Software Testing & Quality Engineering*, 23–26.

Jézéquel, J.-M. and B. Meyer (1997). Design by contract: the lessons Ariane. *IEEE Computer 30*(2), 129–130.

Kneuper, R. (1997). Limits of formal methods. *Formal Aspects of Computing 9*, 379–394.

Knuth, D. (1984). Literate programming. *The Computer Journal 27*(2), 97–110.

Knuth, D. (1992). *Literate Programming*. Center for the Study of Language and Information.

Long, B. and P. Strooper (2001). A case study in testing distributed systems. In *Proceedings 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, pp. 20–30. IEEE Computer Society.

McDonald, J., D. Hoffman, and P. Strooper (1998, November). Programmatic testing of the Standard Template Library container classes. In *Proceedings of IEEE Intl. Conf. Automated Software Engineering*, pp. 147–156.

McDonald, J. and P. Strooper (1998). Translating Object-Z specifications to passive test oracles. In *International Conference on Formal Engineering Methods (ICFEM98)*, pp. 165–174. IEEE.

Meyer, B. (1994). *Reusable Software The Base Object-Oriented Component Libraries*. Prentice Hall.

Meyer, B. (1997). *Object-Oriented Software Construction* (Second ed.). Prentice Hall.

Miller, S. (1998). Specifying the mode logic of a flight guidance system in CoRE and SCR. In *2nd ACM Workshop on Formal Methods in Softwre Practice*.

Murphy, G., P. Townsend, and P. Wong (1994). Experiences with cluster and class testing. *Commun. ACM 37*(9), 39–47.

Musser, D. and A. Saini (1996). *STL Tutorial and Reference Guide*. Addison-Wesley.

Nygard, M. and T. Karsjens (2000, May). Test infect your Enterprise JavaBeans. *Java World*.

Richardson, D. and L. Clarke (1985). Partition analysis: a method combining testing and verification. *IEEE Trans. Soft. Eng. SE-11*(12), 1477–1490.

Spivey, J. (1992). *The Z Notation: a Reference Manual* (second ed.). New York: Prentice-Hall.

Sun Microsystems (2001). *Java Development Kit*. http://java.sun.com.products/jdk: Sun Microsystems.

Weyuker, E. and T. Ostrand (1980). Theories of program testing and the application of revealing subdomains. *IEEE Trans. Soft. Eng. SE-6*(3), 236–246.

White, L. and E. Cohen (1980). A domain strategy for computer program testing. *IEEE Trans. Soft. Eng. SE-6*(3), 247–257.

Winston, P. (1975). *The Psychology of Computer Vision*. McGraw-Hill.