Prose + Test Cases = Specifications

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

Paul Strooper
Dept. of Comp. Sci. and Elec. Eng.
The University of Queensland
Brisbane, Qld. 4072, Australia
pstroop@csee.uq.edu.au

Abstract

The rise of component-based software development has created a need for API documentation. Experience has shown that it is hard to create and maintain 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 write and modify. 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 current prose documentation with test cases, including expected outputs, and use the prose plus the test cases as the documentation. Typically there are one or two simple test cases for each likely question about API behavior. With this approach, the documentation is precise, albeit partial. Consistency between code and documentation is guaranteed by running the test cases. The readability of the test cases is of paramount importance because communication with API users is their primary purpose.

We present a test script language that supports compact, readable test cases and generation of test drivers, and illustrate the approach with a detailed case study.

1. Introduction

With the growth of component-based software development approaches, the importance of 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 serious misunderstandings between API implementors and users. The formal methods community recommends precise specifications, usually written in a logic-based language. Such specifications can be complete and unambiguous, and suitable for processing by automated theorem provers. In some cases, the specifications can also be used to generate implementations or test oracles. Unfortunately such specifications are costly to write and maintain. Worse, few of today's developers are willing or able to read formal specifications.

We present a pragmatic scheme for overcoming the weaknesses of prose and formal specifications. The underlying idea is simple: augment traditional prose documentation with

0-7695-0774-3/00 $10.00 © 2000 IEEE

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. We illustrate the idea with the test cases and output shown in Figure 1. The test cases are for the standard Java `Vector` class and we only discuss the test cases, not the prose documentation that would supplement them.

The first three lines initialize the `Vector v` and display the initial value. Note that `Vector` stores Java `Objects` and that is why we use the class `I` to wrap integers inside an object. Test cases 1–4 show what happens when elements are inserted at the boundary positions: $\{-1, 0, \texttt{v.size()}, \texttt{v.size()} + 1\}$. 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., `v.add(v.size(),`x`)` is equivalent to `v.add(`x`)`. Test case 5 illustrates a key feature of a number of `Vector` methods: that element comparison is done using the `equals` method, which is defined in `I` in this case. When `VectorTest` is compiled and executed, it should produce the output shown at the bottom of Figure 1.

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

1. precise (though partial) documentation,
2. guaranteed consistency of code and documentation,
3. good fault detection, because many API questions focus on special cases, and
4. helpful examples of use, suitable for copying and editing.

With our approach, readability of the test cases is of paramount importance. Indeed, the cases must be more compact and readable than those shown in Figure 1.

The next section shows how to develop compact, readable test cases with the Roast tool. Roast test cases include both input and expected output. Driver generation is automated, including exception monitoring. Section 3 presents a detailed case study. Section 4 discusses the strengths and weaknesses of various approaches and presents related work.

2. Test case templates

The driver in Figure 1 is reasonably compact, but does little checking of the required behavior. For example, the tester must ensure that the correct vectors are printed every time the driver is run. Similarly, the only exception checking that is performed during test execution is that the calls to `v.add` in cases 1 and 4 throw the exception `ArrayIndexOutOfBoundsException`. We could augment the driver to include code to perform additional checking for us, but without tool support this would make the test driver bulky and therefore unsuitable for documentation purposes.

In this section, we introduce the *Roast* test driver generator [3, 5] 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

[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.

```
public class VectorTest {
public static void main(String[] args) {
   Vector v = new Vector();
   v.add(new I(2)); v.add(new I(4)); v.add(new I(6));
   System.out.println(v);

   // What happens when elements are added in boundary positions?
   try { v.add(-1,new I(0)); } // case 1
   catch (ArrayIndexOutOfBoundsException x)
   { System.out.println("Bounds exception: -1" ); }

   v.add(0,new I(0)); // case 2
   System.out.println(v);

   v.add(4,new I(0)); // case 3
   System.out.println(v);

   try { v.add(6,new I(0)); } // case 4
   catch (ArrayIndexOutOfBoundsException x)
   { System.out.println("Bounds exception: 6" ); }

   // How is element comparison performed?
   v.remove(new I(0)); // case 5
   System.out.println(v);
}
}

class I {
public I(int x) { this.x = x; }
public boolean equals(Object i) { return ((I)i).x == x; }
public String toString() { return x+""; }
private int x;
}

/* OUTPUT
   [2, 4, 6]
   Bounds exception: -1
   [0, 2, 4, 6]
   [0, 2, 4, 6, 0]
   Bounds exception: 6
   [2, 4, 6, 0]
*/
```

Figure 1. `VectorTest` **source code and output**

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.

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 the name of 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*. Moreover, templates in test drivers allow us to produce summary statistics about the number of test cases and the number of failures.

2.2. Roast `Vector` test driver

The following two (hypothetical) test cases illustrate the two types of test case templates.

```
#excMonitor v.add(new I(2)); v.add(new I(4)); v.add(new I(6)); #end
#valueCheck v # [2,4,6] #end
```

The first adds 2, 4, and 6 to the vector `v`, but since it is an exception-monitoring test case and the *expectedException* field is omitted, code is generated to ensure that none of the calls to `v.add` generate an exception. The second case checks that after these calls, `v` has the correct value. However, *Roast* does not handle this second case as it stands, because `Vector` is not a standard data type.

For a standard type, such as `int` and `String`, *Roast* uses its own comparison routine to compare *actualValue* and *expectedValue* in a value-checking test case. The tester can override this comparison routine by specifying a *type* field in a value-checking test case. The tester then has to provide a routine for comparing values of that type and for printing an error message if the two values are not equal. Thus, in the *Roast* test driver for `Vector`, we rewrite the second case above as follows:

```
#valueCheck v # new int[] {2,4,6} # VectorCompare #end
```

We then define a class `VectorCompare` to compare a vector, `v` in this case, with an array of integers. The reason we use an array, rather than another `Vector`, for *expectedValue* is that it easy construct a specific array in a Java expression, whereas this is harder to do with a `Vector`.

Roast test cases corresponding to Figure 1 are shown in Figure 2. The test cases are more

```
public class RoastVector {
public static void main(String[] args) {
   Vector v = new Vector();
   String boundsException =
      new String("java.lang.ArrayIndexOutOfBoundsException");

   #excMonitor v.add(new I(2)); v.add(new I(4)); v.add(new I(6)); #end
   #valueCheck v # new int[] {2,4,6} # VectorCompare #end

   #excMonitor v.add(-1,new I(0)); # boundsException #end // case 1

   #excMonitor v.add(0,new I(0)); #end // case 2
   #valueCheck v # new int[] {0,2,4,6} # VectorCompare #end

   #excMonitor v.add(4,new I(0)); #end // case 3
   #valueCheck v # new int[] {0,2,4,6,0} # VectorCompare #end

   #excMonitor v.add(6,new I(0)); # boundsException #end // case 4

   #excMonitor v.remove(new I(0)); #end // case 5
   #valueCheck v # new int[] {2,4,6,0} # VectorCompare #end
}
}

class I {
public I(int x) { this.x = x; }
public boolean equals(Object i) { return ((I)i).x == x; }
public String toString() { return x+""; }
public int x;
}
```

Figure 2. *Roast* `Vector` **test script**

readable than in Figure 1 and the exception-checking test cases are more compact. The implementation of `VectorCompare` consists of a simple for loop and is omitted for brevity.

2.3. Test program generation

The system flowchart in Figure 3 shows how *Roast* generates an executable test program from a test driver with embedded test case templates. Ovals indicate human readable files and boxes indicate executable programs. The test script for class C is prepared in a file, C`.script` in Figure 3, by the test programmer. *Roast* reads the script and generates a Java driver, `driver.java` in Figure 3. This driver and the implementation C`.java` are then compiled. When the resulting test program is executed, the test cases in the test script are run against the implementation and any errors are reported.

2.4. Additional *Roast* features

To add flexibility, *Roast* includes a number of additional features [3]:

- A set of four *unit operations* that provide a framework for a disciplined approach to class testing. Test cases are represented abstractly by *test tuples*, and each unit

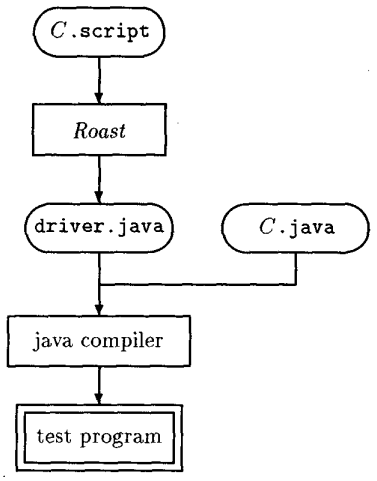

Figure 3. *Roast* **system flowchart**

operation receives a tuple, processes it, and passes it on to the next operation. The four unit operations are *Generate*, which constructs the test tuple set, *Filter*, which removes unwanted tuples from the tuple set, *Execute*, which executes the test case associated with a test tuple, and *Check*, which implements the test oracle.

- Extensive support for generating boundary values and combinations of these boundary values [6].
- Support for generating log messages during debugging and regression testing. While the standard messages generated by *Roast* during test case execution are sufficient during regression testing when no failures occur, during debugging it is often convenient to generate a large number of log messages. Generating all these log messages during normal regression testing would produce a lot of extraneous output, obscuring the important messages. Therefore *Roast* provides a GUI log message viewer with filtering operations to display only the messages of interest.

3. 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 software developers and in tools such as `make`.

This section is based on one solution to the command-line problem. The value of this solution is 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.

3.1. 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 `String`. 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.

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 must be provided to the flags and arguments present. Otherwise, an error message must be made available.

The function prototypes for four classes in the Command Line module are shown in Figure 4. 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 whether or not the string is 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—exception classes `FixedPointArgType`, `AlphabeticArgType`, and `AnyArgType`—have been omitted for brevity.

```
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 4. Command Line module: function prototypes

3.2. 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?
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?

Figure 5 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 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 be a 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 get information about the arguments are refused.

Note that Figure 5 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 5 focuses on communicating module behavior to the user, it has some 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 the Roast features described at the end of Section 2. 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 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 the "FAQ driver" will be useful in defect detection.

4. Discussion and related work

4.1. Discussion

The paper title is a provocative exaggeration. Prose + test cases $\neq$ specifications, 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, documentation based on prose plus test cases may serve effectively for developers who are unlikely to write a careful prose specification, and even less a formal specification.

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

```
public class CommandLineDriver {
public static void main (String[] args)
{
    FlagSpec[] flagSpecs = {
        //flag,isReq,argReq,argType
        new FlagSpec("-a",false,false,null),
        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"); # "ParseException" #end
}
}
```

Figure 5. Command Line Driver: answers to selected FAQs

in all circumstances. With our approach, a family of plausible behaviors is predetermined by the function prototypes, the prose overview, 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. A formal specification cannot exploit the domain knowledge of the reader. If it is considerable, prose and test cases can be very effective. If it is not, formal methods may be vastly 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. For example, 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 post conditions. For example, the Eiffel libraries are documented using a mix of prose and formal notation (in the form of assertions) [8]. The preconditions are often formal and complete, whereas the formal parts of the postconditions are typically partial, if present at all.

4.2. Related work

The use of examples in documentation is an old and obvious idea. Today, use cases [7] are probably the best known technique for software documentation based on examples. While use cases are usually informal and not executable, scenarios can be made executable, as research on SCR requirements specifications has shown [9]. Our test cases can be thought of as executable API use cases.

Using test cases in documentation involves test case selection, a central topic in testing research [12, 11, 10]. Our approach is also consistent with proposals for extreme programming [2, 1], where API test cases play a central role.

Techniques for programming by example have long been studied in the artificial intelligence research community. For example, Winston [13] 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 [4], 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 through examples.

5. Conclusions

With the growth of component-based software development approaches, the importance of API documentation has grown as well. In this paper, we have argued for an approach to specification based on a combination of prose and executable test cases. The test cases have the advantage that they provide compact and practical documentation, with automatic proof of consistency between documentation and implementation by running the test driver. The test cases provide some form of quality insurance, in that they can expose faults.

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. However, we have not shown how the prose can be combined with the test cases

in a coherent specification document. In the case study, the prose and the test cases were presented separately and the test cases were embedded in a test driver that contained some additional Java code of little interest to the specification reader.

Another area for further investigation is to compare our approach to specification with fully formal specifications and with specifications of prose alone. While such a comparison would be hard for specifications in general, it is easier when considering the specifications of APIs, which is the target of our approach. In addition, it is worth investigating what role, if any, the test cases in the specification can play during verification. As indicated, it is likely that they are a good starting point for a more substantial test driver.

Acknowledgements

Thanks to Nigel Daley for development of the Command Line code, and to Ian Hayes and Alena Griffiths for their suggestions on previous versions of the paper.

References

[1] K. Beck. Embracing change with extreme programming. *Computer*, pages 70–77, October 1999.

[2] K. Beck. *Extreme Programming Explained.* Addison-Wesley, 1999.

[3] N. Daley, D. Hoffman, and P. Strooper. Unit operations for automated class testing. SVRC Technical Report 00–04, University of Queensland, 2000.

[4] S. Engelmann and D. Carnine. *Theory of Instruction: Principles and Applications.* ADI Press, Eugene, Oregon, second edition, 1991.

[5] D. Hoffman and P. Strooper. Tools and techniques for Java API testing. In *Proceedings 2000 Australian Software Engineering Conference*, pages 235–245. IEEE Computer Society, 2000.

[6] D.M. Hoffman, P.A. Strooper, and L. White. Boundary values and automated component testing. *Journal of Software Testing, Verification, and Review*, 9(1):3–26, 1999.

[7] I. Jacobsen. *Object-Oriented Software Engineering.* Addison-Wesley, New York, 1992.

[8] B. Meyer. *Reusable Software—The Base Object-Oriented Component Libraries.* Prentice Hall, 1994.

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

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

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

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

[13] P. Winston. *The Psychology of Computer Vision.* McGraw-Hill, 1975.