Patterns of Input Processing Software

Neil B. Harrison
Lucent Technologies
11900 North Pecos st.
Denver CO 80234
(303) 538-1541
nbharrison@lucent.com

Background: Setting the Stage

Many programs accept user input, and it is always necessary to interpret the input to figure out what the user wants. Furthermore, users can make mistakes, so programs must be able to handle erroneous input without crashing. It is also important to let the user know about the error in a graceful manner. There are numerous software patterns that deal with processing user input.

The CHECKS Pattern Language of Information Integrity [Cunningham 95] presents a language of patterns that describe how to capture and check values, and detecting and correcting mistakes. This set of patterns focus more on the internal structure of software that supports processing user input. In this way, these input processing patterns and the CHECKS patterns complement each other: the CHECKS Pattern language states, "The CHECKS pattern language tells how to make such checks without complicating programs and compromising future flexibility." It also states that, "They are written as if they are part of a larger language..." These patterns are also part of that larger language.

All these patterns share this common context; of course, each pattern may add more context. These patterns concern programs that accept human input in a specified format. (It excludes such programs as signal processing patterns and the like.) The input can be things such as database queries, input into forms (think web-based applications), languages to be compiled or interpreted, and others. In each case, the program deciphers the input, and performs actions according to the input. The program usually takes action immediately upon receipt of the input, but some actions may be deferred, waiting for time to elapse, or for more input to come in.

In each of these programs, there is a defined language. Database queries follow a specified syntax and semantics, as do programming languages to be compiled. Even forms-based inputs have a language; it is implicitly defined by the forms themselves.

Forces (general):

The following forces are common to all the patterns. Additional forces specific to a pattern are given with each pattern.

1. One of the few givens in software (as in life) is that things change. No sooner do we finish a program than we begin to work on changes to it. The following things may change:

- The language. Often, this is adding more features to the program, and new language is needed to support it.
- The interface. It is not uncommon to add a different interface, such as a web interface, to an existing product.
- The processing of the input. New features, of course, require new processing. Or the processing may change with no changes to the input language or the interface. For example, you might fix a bug.
- Any combination of the above.

2. Processing should be reasonably efficient. Naturally, different programs have different performance requirements. However, in most cases, we care some about performance.
3. If an organization has several programs, it is desirable to maintain a consistency of look and feel across the programs.
4. The original designer isn't going to be around forever. Therefore, the software design should be clean and readily understood by others.

The following patterns help address these forces. Although the patterns may be used singly, they work together naturally. They are described in detail in the following pages.

1. Separate Parsing and Processing
2. Separate Lexing and Parsing
3. Parse Complete Statements
4. Parser Catches Errors
5. The Enter Key

These patterns are well known, and have been used in many programs. You can find examples of each of these patterns in any of the following:

- Many compilers
- Many lex and yacc applications.
- The author's personal use in various programs.

1. Separate Parsing and Processing

Problem:

How do you organize the software to handle input processing, particularly in the face of inevitable change?

Forces:

1. The code for all parsing has a lot in common.
2. You often don't know just what kind of input you are receiving until it passes through semantic analysis (i.e., parsing). Note that this may not apply if the interface is forms driven; the form may provide the semantics in place.
3. It is common that processing changes are linked to syntactic and semantic changes. In particular, adding a new capability to a program usually requires additions to the language as well as additions to the mainline processing.
4. Tools exist to help create input processing software. GUI builders can help create forms; lex and yacc help define and process grammars.
5. Input processing takes on different forms depending on the method of the input, yet a program may need to be changed to support a new input method. For example, a program that has taken textual line-based input may be changed to support GUI input. You want to change as little code as possible to support the new interface.

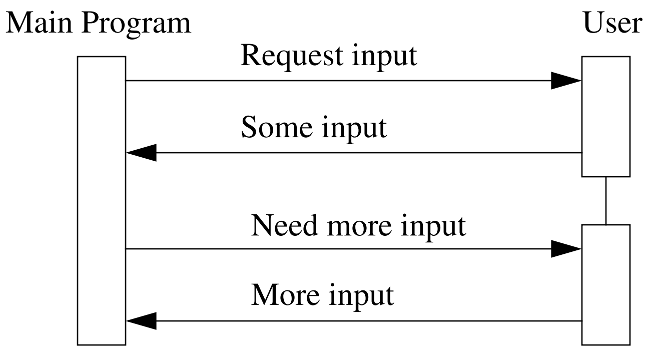
Solution:

Keep the parsing software separate from the mainline processing software. The mainline software should have no parsing code whatsoever; it concentrates on doing what it is supposed to do. Note that conceptually, this is separating interface from implementation: a given command or operation has a specific syntax and semantics, namely its interface; this is handled by the parser. Processing the operation is its implementation.

This may seem obvious, in fact, trivial, to many people. However, most of us have seen programs in which the input processing was intermingled with main processing. For example, Shell programs in UNIX tend to be this way. Such programs are difficult to maintain, and can be a nightmare to enhance for a different user interface.

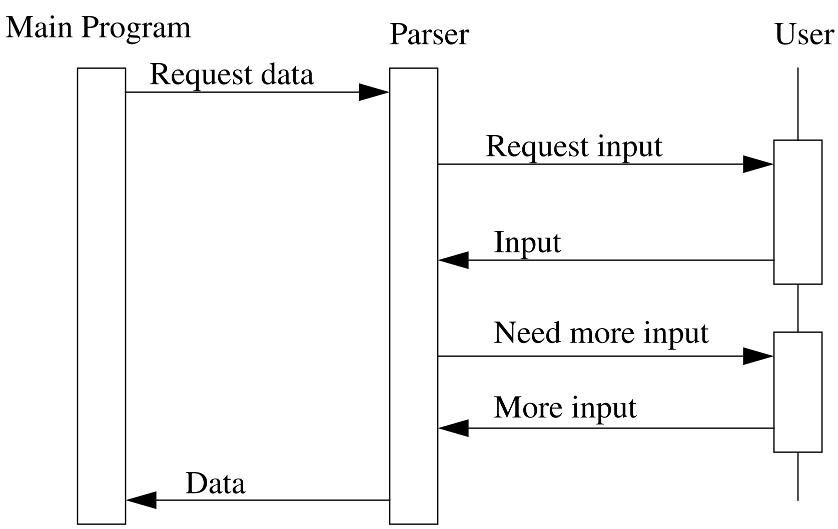
Example:

What we DON'T want:



This looks very simple and clean, but what happens when the language or input method changes? You have to make numerous changes throughout the main program. This is time consuming and error-prone.

Instead, we want a program like this:



In order to do this, there must be a well thought-out interface between parsing and mainline processing. In the OO world, this interface can readily be defined using objects. One approach is that parser creates objects to be processed, and then hands them to the main program for processing. One can think of the parsing subsystem as a large sophisticated object factory.

Resulting Context:

The Separate Parsing and Processing pattern has the following advantages:

1. It is easy to change the user interface.
2. Since the parser is separate from the main program, it lends itself well to the use of tools to create the parser.
3. Parsing languages such as programming languages often is implemented most easily with non-OO programming (such as recursive descent parsing). Yacc, for example, generates C code. The separation of parsing from processing allows the main program to be written in OO, if desired, while the parsing is not.
4. This separation helps concentrate error handling in one place. (See Parser Catches Errors, later in this collection.)

This pattern also has some drawbacks, namely:

When you add a new command or operation to the program, this pattern forces code to be added in two places, the parser and the mainline processing. Therefore, it is easier to forget something, or to get the two parts out of synch with each other. This problem can be considerably ameliorated, however, by using the Command pattern [Gamma94] as follows:

1. Define objects that represent the commands or operations.
2. When the parser recognizes a command or operation, it creates the corresponding object, and passes that object to the mainline processing. (See earlier discussion of parser as object factory.) Note that object passing helps define the interface between the parser and the mainline processing.
3. The mainline processing is now responsible for that object; its processing and its life cycle.

Note that this puts most of the actions concerning a given command in its own object definition. The parsing code is still separate from the object, but it becomes close to the object by virtue of creating the object.

Related Patterns:

- All the later patterns in this language are closely related to, and help implement this pattern.
- Look at Abstract Factory [Gamma 94] and Factory Method [Gamma 94] for further insight into object factories.
- The parser may be implemented as a Singleton [Gamma 94].
- The Model-View-Controller [Buschmann 96] is quite similar to this pattern.

2. Separate Lexing and Parsing

Problem:

How do you organize the input processing software to make it easy to implement and maintain?

Context:

You have applied the Separate Parsing and Processing pattern. Now you want to organize the input processing part of the software in the most effective way.

Forces:

1. In most input languages, there is a set of atomic elements common to the language. These often include a set of keywords, a set of numbers (such as integers, floating point, etc.), special symbols, and/or others. For example, most programming languages have all of the above. On the other hand, a banking application may have a language restricted to names, account numbers, money numbers, and a small menu of choices. And the menu may be a set of buttons on an ATM machine, or a pull-down menu on a form.
2. Input must be validated. This includes validation of atomic elements (for example, the user types “innnteger” instead of “integer”), and of how the atoms are put together (for example, the C language requires arithmetic expressions in algebraic, not reverse Polish, notation.)
3. If the language changes, the software should be organized so as to keep the software changes localized, as much as possible.
4. Changes to a language may happen in any of the following areas:
    - Changes to the structure of the language, such as adding new commands.
    - Changes to the atoms. For example, if an atom in the language is “color”, a British version of the program may change the atom to “colour”.
    - Changes to admissible values of atoms. For example, the maximum value of a certain numeric atom may increase as the program capacity increases.
5. As stated before, a program may have to support different interfaces. Such interfaces may be better suited to different types of atomic element validation. For example, a keyboard interface accepts anything the user types (much of which can be invalid), while a forms interface may limit user input through pull-down menus and push buttons, with a few fill-in fields. These inherently provide a much higher level of lexical enforcement than free-form keyboard input.
6. There are some lexical analysis generation tools (i.e., lex).
7. A language should be self-consistent; it should have a consistent look and feel within itself.

Solution:

Separate the lexical analysis function from the semantic checking, or parsing. Together they make up the input processing. This is in effect, separating the “vocabulary” from the “grammar”. Note that such a vocabulary doesn’t have to be limited to words. For example, a windows-based vocabulary may include such things as confirmation pop-up windows

Example:

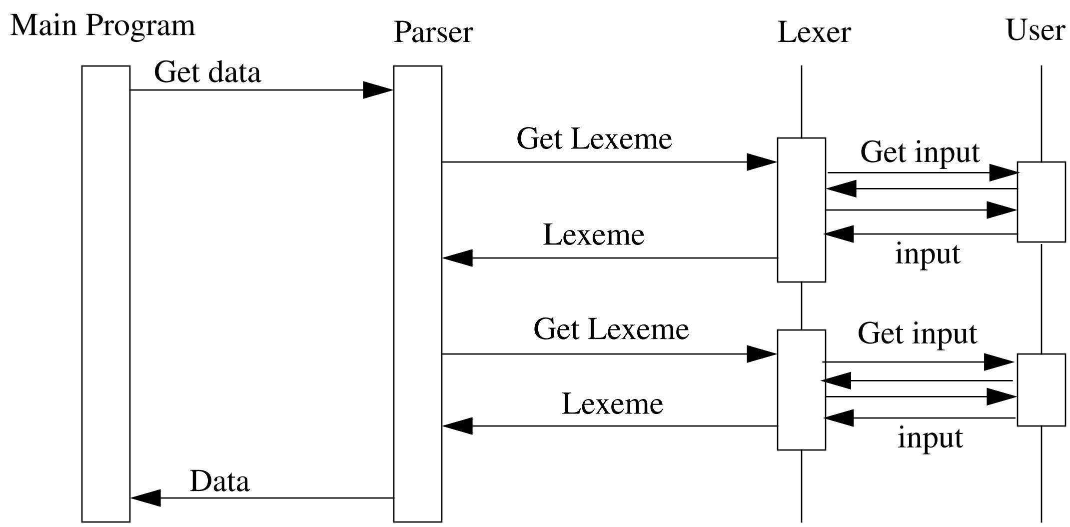
If you are designing a text-based input processing system, you might do the following to implement this pattern:

1. Define the set of atoms in the language. These atoms, sometimes called lexemes (not to be confused with lemmings), are either constant (i.e., keywords) or variable (such as strings or integers). The constant keywords tend to define the structure and context of the language, while the variable lexemes convey the information from the user.
   For example, programming languages have keywords such as “integer”, “=”, “:=”, etc. Variables for a banking application would be text string (for name, address, etc.), long integer (for account number and PIN), and number with two digits behind the decimal point (for monetary amounts). Note that account number and PIN are the same lexeme; it is up to the parser to determine from the context what a long integer means. (If account number and PIN were separate lexemes, it would force some semantic analysis into the lexer, which complicates matters considerably.)
2. Create a lexer function to read the input, and determine which lexeme the input corresponds to. The lexer passes the lexemes to the parser; depending on whether the lexeme is constant or variable, the lexer may or may not pass an associated value.

   Note that the lexer may receive input that does not match any defined lexeme. In this case, the lexer will detect an error. The Exceptional Value pattern from the CHECKS pattern language may be especially effective here.
3. The parser interprets the lexemes. Therefore, the main program calls the parser, and the parser calls the lexer several times, until it either has a valid input, or it recognizes an error in the input.

   Note that it is entirely possible to input erroneous data that the lexer interprets as a different lexeme than what was intended. If the user means to type “integer”, and types “innnteger” instead, the lexer may interpret it as a variable lexeme of type String, rather than a constant keyword of type “Integer”. In this case, the semantic analysis of the parser will detect the error (unless the input language is written pretty poorly).

The data flow looks like this:



Resulting Context:

Application of this pattern results in the following advantages:

1. Common elements in the language can be handled and verified in one place; wherever an integer is requested, for example, the program uses common code to input an integer.
2. When adding new elements to the language, one is naturally encouraged to use the existing lexemes as building blocks (since they are already there.) This helps preserve the self-consistency of the language. (This becomes more complete with the application of the subsequent patterns, “Parse Complete Statements” and “Parser Catches Errors”.)
3. It makes it easy to take advantage of existing tools (such as lex and yacc).
4. Changes to lexemes can be made easily, without perturbing the rest of the program.

This pattern also has the following drawbacks:

1. There is an additional layer of code to go through on input, so the program will be somewhat less efficient.
2. Input errors may be caught in either the lexer or the parser, so both need to handle errors. They have to handle them in a consistent manner, since the user doesn’t care (or even know) whether the error is in lexing or parsing. This adds a slight layer of complexity to error handling.

Related Patterns:

1. See “Parser Catches Errors” for further discussion of error handling.

3. Parse Complete Statements

Problem:

A parser collects a volume of input, and puts it into a form that can be easily used by the main processing part of the program. How much should the parser collect and package before handing the package to the main processor?

Context:

Parsing and Processing (pattern 1) is being applied. You are trying to figure out where to draw the separation line.

It is not necessary that Separate Lexing and Parsing (pattern 2), but it is conceptually helpful at this point. If you have applied this pattern, you are more focused on the problem at hand, stated above.

Forces:

1. If the main processor gets too little data at once, it will have to store that data somehow, and go back to the parser to ask for more.
2. If the main processor gets too much data at once, it may have to do too much. In the worst case, it may have to do a sequence of unrelated actions, and will have to so some semblance of parsing of the already parsed data, to figure out what to do.
3. Input is context-dependent. When the user types in a number, there is some surrounding context that gives meaning to the number. For example, when the user types in “5” on an order form, the context may tell us that the user wants five of the specified item.

Solution:

Define “complete statements”. These are units of input that can be processed more or less completely. In many cases, the input of the program breaks naturally into semi-independent units of work, such as queries to a database. In other cases, the units of work are explicitly specified by the input language, such as found in many programming languages.

The parser continues to request input until it has a complete statement. Then, and only then, does it hand over control to the main program.

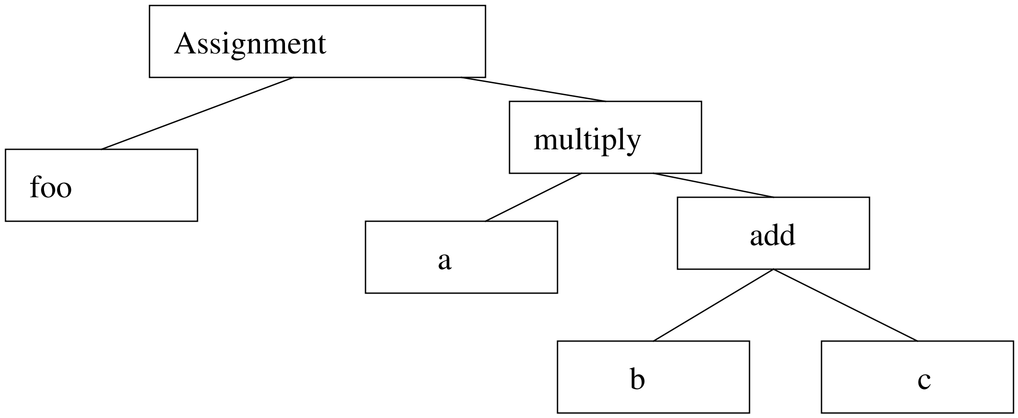
The parser assembles a structure that resembles the complete statement: it has all the relevant elements, stored in a form that can be naturally processed. There are different types of structures (or objects in OO), for the different types of statements in the language.

Example:

Consider an assignment statement in a programming language. The input is the following:

```
foo = a * ( b + c );
```

The parser reads it, and creates an assignment object in such a way that the main processor doesn't have to worry about parenthesis, operator precedence, or other complexities of the expression. It looks like this:



Note that in this example, the objects representing the input look somewhat different from the input itself. In many cases, the objects will resemble the input more closely.

Resulting Context:

The Parse Complete Statements pattern has the following advantages:

1. It solves the problems associated with getting too much or too little data. When the main processor function gets something back from the parser, it is a complete unit, ready to be digested. The main processor doesn't need to worry about receiving something that is incomplete. (For handling things that are wrong, see the next pattern, Parser Catches Errors.)
2. Many languages naturally break into statements that can be parsed as a unit. Graphical input is also often naturally somewhat transaction-based.
3. This makes it easy to add completely new elements to the language.

It also has the following drawback:

1. This pattern does force a transaction-oriented style of language; if the input from the application does not naturally fall into this language style, then force-fitting it results in a language that is difficult for people to use.
2. You can't always apply this pattern. In some cases, the logical statements or transactions may be grammatically complete, but it is necessary to get several of them to complete the thought. In other cases, it may be impossible for the parser to determine what a complete statement is.

Related Patterns:

This pattern has much in common with the Whole Value pattern of the CHECKS language. It differs in that Parse Complete Statements does not imply an interface for converting different formats.

This pattern works best when "The Enter Key" pattern is also applied.

4. Parser Catches Errors

Problem:

Users make errors when inputting data. How do you organize the software to most effectively (and easily!) handle the errors?

Context:

You are designing an approach to handling all the different types of errors a user may make. You need to handle the errors, and give information back to the user about the errors as well.

You have applied the first three patterns, Separate Parsing and Processing, Separate Lexing and Parsing, and Parse Complete Statements.

Forces:

1. Input errors can be different types, namely:
    - Syntax; unrecognized lexeme
    - Semantic; correct lexemes, but in an unrecognized sequence.
    - Value; such as numbers out of range.
    - Extended context; the validity of this input depends on previous input, or something else. Examples are undeclared variables in a programming language, or trying to withdraw more money than your checking account has in it.
2. You need to provide feedback to the user as soon as possible, meaning before they input anything else. Backing up to a previous statement is both difficult for the programmer and confusing to the user.
3. You need to provide adequate information about the error to the user.
4. Error checking tends to clutter up code. It detracts from the main idea of the code. It is easily overlooked. Because of this, ironically, it is error-prone, particularly when it is scattered throughout the program.

Solution:

Concentrate as much error handling as possible in the lexer and the parser. The idea is that when the main processing functions receive data back from the parser, it should be a complete package, that has already been cleaned and checked for errors. Then the main processing can operate on the data, knowing that the data is sound.

The work of error checking is shared between the parser and lexer. Put the error checking as close to the input source as possible; this means that the lexer checks for unrecognized lexemes, and the parser checks for unrecognized sequences of lexemes. Design a way of handling input errors that both can use. Note that error messages must indicate the location of the error, and other useful information, such as the offending lexeme.

What if the input is syntactically and sematically correct, but the value (especially numeric) is invalid? One approach works well in OO programming. As stated earlier, the parser may be responsible for creating data objects. Numeric values can be validated as the objects are created, perhaps in initialization functions. Alternatively, values can be checked as part of the construction process of an Abstract Factory. Values which need to be different over time, or other dynamic values can be checked in a Singleton Abstract Factory. For example, if only a certain number of objects of a given type is allowed, a Singleton Abstract Factory can keep track of how many have already been created.

In some cases, the input may be correct, but still invalid, and validation during instantiation isn't convenient. For example, some systems ask for a user name and a password. In such a case, there may be a separate validation function, which is called from within the parser. Once again, keep that validation outside of the main processing.

There are some situations where validity of input depends on context established by previous input. In many programming languages, for example, variables must be declared before they are used. In automated teller machines (ATMs), a transaction may consist of account number and PIN, and another transaction is withdrawal from the account. You don't want to allow the user to overdraw the account! In this case, it is probably impractical to put the checking inside the parser; but remember these are guidelines, not dictates to be slavishly followed.

Advantages:

1. The main processing code is generally free from error handling distractions, so it is easier to understand and maintain.
2. With input error handling concentrated in just a few places, it is easier to provide a common error handler.
3. It is easy to notify the user of an error, and the error is usually the thing the user just put in. So there is less need for the user to back up.
4. Forms-based interfaces often have error checking embedded in them. So changing to a forms-based interface is easier if the error checking is already in the lexer and parser.

Drawbacks:

1. There are always exceptions to the rule. Force-fitting all error handling into the parser doesn't work; the solution isn't perfect.

Related Patterns:

See Abstract Factory and Singleton, as described above.

This pattern is very closely tied to the CHECKS pattern language; in fact, it kind of surrounds much of it. Name the specific patterns.

There are some ideas in the Patterns of Logging (see Diagnostic Logger and Typed Diagnostics) that can be useful when dealing with error handling in the lexer and parser. [Harrison 97]

5. The Enter Key

Problem:

How does the parser know when input is complete enough to process?

Context:

You have applied Separate Parsing and Processing. You are applying Parse Complete Statements, and want to know how do so.

Forces:

1. You want to separate parsing and processing by parsing complete statements.
2. It isn't always easy, though, to keep parsing and processing separate. Sometimes the input is somewhat ambiguous about where to break statements.
3. If the user makes a mistake, or changes his or her mind, backing up can be troublesome. This is true even if error handling in concentrated in the parser.
4. Sometimes, because of the nature of the language, it may be difficult to know where the end of a statement is.

Solution:

Put an "Enter Key" in the language. It may be the "Enter" or "Return" key on a keyboard, or an "Enter" button on a screen, or something else that marks the end of a statement. It is visible to the user, and indicates that a thought is complete, and it is all right to process it. Ideally, the enter key should be used only for this purpose.

Examples:

In many programming languages, each statement ends with a terminating character, such as a semicolon. (Early programming language statements were marked by the end of the punched card.) This is sometimes called "syntactic sugar"; that is, it isn't absolutely necessary for the language, but it does help a lot. It makes the parsing easier, and actually helps the users divide up their thoughts into manageable pieces. (Note that the semicolon used in C and C++ is an imperfect example, because it is used elsewhere in the language. It isn't as clear as it could be.)

Imagine a web-based order form. Many forms have several fields to fill in, with an "enter" button at the bottom. In fact, many forms also have a "clear" button, so you can change your mind before you make a financial commitment. Some systems may be more complicated, allowing you to browse several rooms, adding things to your virtual shopping cart. Then you commit just before leaving the virtual store. Such cases require a somewhat more sophisticated "enter key" model, but principle is still the same.

Resulting Context:

The Enter Key has the following advantages:

1. Parsing and processing can more readily be separated. Where it was ambiguous before, the Enter Key makes it explicit.

2. Changes, because the user changes his or her mind, or makes an error, can be easily accommodated as long as they happen before the Enter Key is pressed.
3. The Enter Key has an interesting effect on the language and the processing. It helps make the processing look like the language. It also helps compartmentalize the user's actions and thoughts. In many cases, this is a natural way to think of the language anyway; this makes it obvious.

Drawbacks:

1. Some applications just don't fit this model well. Imagine, for example, menu-driven applications such as found on kiosks with touch screens. In many such systems, you make a selection, and the computer immediately processes it. Don't force-fit the Enter Key into applications where it is not welcome.

Acknowledgments

The author wishes to thank Mark Bradac for his helpful comments and insights in this area.

References:

[Buschmann 96]Buschmann, Frank et al, *Pattern-Oriented Software Architecture. A System of Patterns*, Chichester, England: John Wiley & Sons, 1996.

[Cunningham 95] Cunningham, Ward. "The CHECKS Pattern Language of Information Integrity," in *Pattern Languages of Program Design*, Reading MA: Addison-Wesley, 1995, pp 145-155

[Gamma 94] Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA: *Addison-Wesley*, 1994.

[Harrison 97] Harrison, Neil. "Patterns for Logging Diagnostic Messages," in *Pattern Languages of Program Design 3*, Reading, MA: Addison-Wesley, 1997.