Reimagining Literate Programming

James Dean Palmer
Northern Arizona University
James.Palmer@nau.edu

Eddie Hillenbrand
Northern Arizona University
eh88@nau.edu

Abstract

In this paper we describe Ginger, a new language with first class support for literate programming. Literate programming is a philosophy that argues computer programs should be written as literature with human readability and understanding of paramount importance. While the intent of literate programming is to make understanding computer programs simpler, most literate programming systems are quite complex and consist of three different languages corresponding to 1) an implementation language, 2) a documentation language, and 3) a literate programming glue language. In Knuth's original implementation these were Pascal, TEX, and WEB respectively. Antithetical to the goals that literate programming espouses, this three language paradigm creates a truly challenging environment for new programmers. In this paper we reimagine literate programming as a core programming language feature and describe a novel system for literate programming based on G-expression transformations. We show that Ginger code can be used to naturally represent code, prose, and literate connections, which in turn unifies, simplifies and significantly extends the literate programming experience.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Documentation

General Terms Design, Documentation, Languages

Keywords Ginger, Literate Programming, Program Comprehension

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

1. Introduction

Literate programming is a programming paradigm that emphasizes human comprehension and readability by adopting the mantra that programs should be read and written as literature. While literate programming systems do support the development of documentation, literate programming is *not* a documentation system. Literate programming's intent is to support distinctly human cognitive abstractions for breaking up problems into tractable parts and communicating the relationships between these parts and their neighbors. In the same way that object oriented programming represents a paradigm or way of thinking and not a specific set of object oriented languages, literate programming intends to transform the way we think about software development in terms of cognitively rooted abstractions.

Literate programming systems describe these cognitive abstractions between implementation and description with a cognitive unit called a *chunk*. Chunks are not limited to the abstractions and forms of either the implementation or documentation language. Code and documentation chunks can be connected or nested to form a literate web that describes a program. Most literate programming systems act as heavy handed preprocessors that recognize very little about the underlying documentation and implementation languages they act on. While workable and perhaps even pragmatic, we believe current literate programming systems obscure and limit the true power of the paradigm by treating literate programming as simply a macro driven preprocessing step fundamentally divorced and different from both the documentation and implementation languages.

In this paper we describe Ginger, a language that is specifically designed to support literate programming. Unlike existing literate programming systems, Ginger uses homoiconic G-expressions to represent code, prose, and literate connections. The result is that code and documentation are represented both internally and externally in exactly the same form. Thus, a uniform interface exists for implementation and documentation chunks to manipulate, transform and inspect each other to such an extent that the boundary between implementation and cognitive description is blurred.

1.1 A Brief Overview of Literate Programming

Literate programming was conceived by Donald Knuth in the early 1980s as an alternative to structured programming. In Knuth's original vision, literate programs were essentially essays or exposition that describe software in ordinary prose while interleaving traditional source code. Literate programming tools can be used to *weave* literate programs into formatted human readable documentation with rich cross-referencing, indexing and bibliographies or *tangle* them into a format suitable for a compiler while preserving meaningful compiler warnings and errors and debugging tool support. The literate programming glue language that Knuth developed is called WEB and has given rise to a number of other systems that have improved or simplified various aspects of WEB. As Knuth worked with WEB, he realized early on that literate programming was changing how he wrote software and enabling him to write software of much greater complexity, quality, and sophistication [2]. Over the years literate programming practitioners have identified three distinguishing characteristics of literate programs:

Psychological arrangement: Literate programs intend to communicate complex ideas and algorithms using plot, narrative, rhythm, and distinctly human story telling conventions instead of the restrictive and rigid structure of a programming language.

Enhanced readability: Literate programs present programs in a form that maximizes readability and understanding by providing cross-references, indicies, bibliographies and syntax markup.

Versimilitude: Code and documentation are written together in the same document such that both documentation and code are active and evolve together.

At its core, literate programming is a philosophy that forces a fundamental shift in thinking and problem solving that focuses on communication. Literate programming changes the perspective of the programmer - emphasizing human communication over language dictated program structure. The paradigm forces programmers to consciously and continuously evaluate the presentation and readability of their code. This mentality fundamentally changes the way programmers approach software development. When it becomes difficult to explain the logic of a particular piece of code, it is often easier to rewrite the code than explain why the code is difficult to understand [1]. Writing software that better communicates its message tends to make software simpler, more flexible, and easier to maintain [20, 5, 16, 1].

1.2 The Need for Literate Software

A common attitude among software developers is that documentation is of little use [13]. At the same time, roughly 50% of the time spent on software maintenance is related to simply understanding the function of program code and may contribute anywhere from 30-90% of the total cost of the software over its entire life [4, 21]. The disconnect between an obvious need to improve communication and problem understanding and a disdain for software documentation may stem from a genuine inadequacy in traditional software documentation.

The large investment and poor returns associated with traditional program documentation has, in part, fueled agile methodology that deemphasizes artifacts that do not contribute to working code. Many believe that with its deemphasis of formal written artifacts, agile methodology is incompatible with literate programming. In a position paper by Pieterse, Kourie and Boake a case is made that, to the contrary, literate programming is *fundamentally compatible* with agile processes and goals [14]. They point out the positive role literate programming has in supporting communication between developers and other stake holders and the positive association between literate programming and high-quality low-defect software. One of their principle arguments is that literate programming documentation should simply not be considered a separate artifact and instead should be considered an intrinsic part of the deliverable and programming process.

1.3 Related Work

Knuth's seminal work on literate programming [8, 9] laid the foundation for a host of different literate programming systems including WEB, CWEB, Noweb [15, 7], Nuweb [10], Funnelweb [22], and others. Many of these efforts have sought to make literate programming more portable (supporting more target languages) and simpler to use. While the syntax used by literate programming systems may differ considerably they all define a cognitive unit called a *chunk*. Chunks are not limited to the abstractions and forms of the underlying programming language and provide a mechanism for supporting conceptual abstractions. Code and documentation chunks can be connected or nested to form a literate web that describes a program.

Many people often confuse *embedded documentation* systems, which include Perl's POD, Java's JavaDoc and Python's pydoc, with literate programming systems. These tools enable documenting interfaces at the actual function prototype definitions. The advantage to documenting in this way is that it becomes easier to keep the documentation closely aligned to the actual interface. This kind of documentation process has little to nothing in common with the literate programming process and embedded documentation tools generally lack necessary literate capabilities [3].

Somewhere between embedded documentation and true literate programming lives *semi-literate programming*. Semi-literate programming systems generally make sweeping simplifications that compromise what most literate programmers would call a truly literate system in order to simplify the literate programming process. The most common simplification is to disable arbitrary code reordering, thus fixing the direction of the narrative to the actual flow of the program. Examples of semi-literate systems include Haskell [6] and PyLit [11].

A few literate programming systems have taken a much different tack based on novel user interfaces. Edward Ream's literate editor, Leo, uses visual outlines that allow users to attach metadata and descriptions to program descriptions and data [17]. Unfortunately, truly literate programs may break Leo's hierarchical outline based paradigm. Stritzinger and Sametinger have developed a hypertext flavored browser for navigating literate documentation specifically for object oriented programming [19, 18]. In the same way that Leo has a bias toward heirarchical structures, Stritzinger and Sametinger's work has a bias toward object oriented relationships.

2. Literate Ginger

Unlike other literate programming systems which mix several, often incompatible, syntaxes together, literate Ginger programs are completely made up of G-expressions. A G-expression is made up of symbols, numbers, strings, literals, S-expression based lists, indented blocks and other G-expressions. A detailed description of G-expressions is given in [12]. One of the keys to literate programming in Ginger is a feature called *colon-quoting*, which begins a special kind of quote which ends at the end of the corresponding command, line or block. Unescaped parenthesis in a colon-quoted string break out of the block and their result is appended to the string. Consider this example,

```
define x 3.14
:println The value of x is (x).
```

which is semantically identical to

```
define x 3.14
println "The value of x is " (x) "."
```

and would output:

```
The value of x is 3.14.
```

Colon quotes don't require a function to act on. An alternate rendering of our last example using colon-quotes without a default function call would be:

```
define x 3.14
println (: The value of x is (x).)
```

Another colon-quote form is the *block colon-quote* which acts on blocks of text at the same indentation level:

```
:println
  This is a much longer colon-quote and
  shows the value of x is (x), but the
  value of y is (y).
```

Simple literate statements like,

```
:title Koch Snowflakes in Ginger
```

do not represent a special documentation language; they are simply calls to normal Ginger functions. We feel this syntax is easily on par with LaTeX in terms of simplicity, readability and ease of use.

2.1 A Simple Example

In the remainder of this section we will describe how literate programs are constructed in Ginger. We shall motivate this discussion with a simple but complete example of literate programming in Ginger. Please note that this example has been designed for brevity while at the same time illustrating key literate programming features.

```
:title Koch Snowflakes in Ginger

:section Introduction

:doc
  The following program demonstrates
  literate programming in Ginger in much
  the same spirit as the primes programs
  that appears in (:cite knuth:literate).
  This program will generate a Koch
  snowflake using turtle-style graphics.
  We shall begin as Knuth did, by reducing
  the entire program to its top-level
  description.

chunk *
  :$ program to display a Koch snowflake

:section Implementation Plan

:doc
  Sometimes the best beginning is the end.
  What we would like to do in this program
  is generate a fractal snowflake with
  "sides" of length 100 which we will store
  in a file called
  (:code koch-snowflake.png).

chunk (: create a snowflake)
  Koch-snowflake 100
  save-canvas "koch-snowflake.png"

:doc
  While (:code save-canvas) is implemented
  by the graphics library, we will need to
  define functions that implement the
  snowflake. These include the
  (:code Koch-snowflake) function we have
  already alluded to in the previous chunk
  and the (:code Koch-curve) function on
```

```
  which it is based.

chunk (: program functions)
  :$ Koch snowflake function
  :$ Koch curve function

:doc
  The program structure is then a simple
  matter of providing the function
  implementation and using that
  implementation to create the desired
  output.

chunk (: program to display a Koch snowflake)
  :$ program functions
  :$ create a snowflake

:doc
  In the remaining sections we will delve
  into the process of creating fractal
  curves and snowflakes.

:section Koch Curves and Bump Fractals

:doc
  A Koch curve is a "bump fractal."  The
  general recipe for generating a bump
  fractal is to draw the fractal at one
  level of recursion and then replace each
  (:code forward) call with a recursive
  call.  The Koch curve is based on a
  single triangular bump illustrated here:

image "bump.png" width: 1.8

:doc
  By thinking like a turtle we can easily
  come up with the corresponding drawing
  code which is relative to the horizontal
  measure or extent, (:code x).

:code
  forward x
  left-turn 60
  forward x
  right-turn 120
  forward x
  left-turn 60
  forward x

:doc
  We generate the recursive case by using
  the bump fractal recipe and replacing the
  (:code forward \(/ x 3\)) calls with
  (:code Koch-curve \(/ x 3\)) calls.

chunk (: recursive case)
  Koch-curve (/ x 3)
  left-turn 60
  Koch-curve (/ x 3)
  right-turn 120
  Koch-curve (/ x 3)
  left-turn 60
  Koch-curve (/ x 3)

:doc
  A Koch curve has infinite length since
  each recursive step generates four new
  segments one-third the length of the
  original segment. The total length of
  the curve becomes one-third longer with
  each recursive step (:cite koch:curve).
  Stated more formally, the length of
  the curve at step (:math n) is
  (:math \(4/3\)^n).  A related measure,
  the fractal dimension, describes how
  how the complexity of the fractal
  increases as it scales. The fractal
  dimension of a Koch curve is
  (:math log 4 / log 3) or approximately
  1.26.

  Though the fractal has infinite
  length and is composed of an infinite
  number or segments, the resolution of
  our display is finite. It is convenient
  to end the recursion at the smallest
  representable length - a pixel. Our base
  case is then to simply draw a line of
  length (:code x), where (:code x < 1).

chunk (: base case)
  forward x

:doc
  We combine the base case and the
  recursive cases to form our Koch-curve
  function that generates a single Koch
  curve whose horizontal measure is
  (:code x):

chunk (: Koch curve function)
  define Koch-curve (x)
    if (< x 1)
      :$ base case
    else:
      :$ recursive case
```

```
:section Koch Snowflakes

:doc
  The (:code Koch-snowflake) function is
  trivially implemented by repeating three
  Koch curves to form an equilateral
  triangle.

chunk (: Koch snowflake function)
  define Koch-snowflake (x)
    repeat 3
      Koch-curve x
      right-turn 120

:section Results

:doc
  After the program is executed, the
  following image is generated.

image "koch-snowflake.png" width: 1.25

:bibliography koch.bib
```

2.2 The Tangled Program

The code from Section 2.1 can be compiled with the Ginger compiler to create an executable program or to create high quality documentation. The executable code extracted from this example follows.

```
define Koch-curve (x)
  if (< x 1)
    forward x
  else:
    Koch-curve (/ x 3)
    left-turn 60
    Koch-curve (/ x 3)
    right-turn 120
    Koch-curve (/ x 3)
    left-turn 60
    Koch-curve (/ x 3)

define Koch-snowflake (x)
  repeat 3
    Koch-curve x
    right-turn 120

Koch-snowflake 100
save-canvas "koch-snowflake.png"
```

2.3 The Literate Result

While the executable code is far simpler, the literate rendering that follows is full of subtle detail and mental process completely missing in the bare implementation.

Koch Snowflakes in Ginger

1. Introduction

The following program demonstrates literate programming in Ginger in much the same spirit as the primes program that appears in [1]. This program will generate a Koch snowflake using turtle-style graphics. We shall begin as Knuth did by reducing the entire program to its top-level description.

⟨* 1⟩ ≡
  ⟨program to display a Koch snowflake 2⟩

2. Implementation Plan

Sometimes the best beginning is the end. What we would like to do in this program is generate a fractal snowflake with "sides" of length 100 which we will store in a file called `koch-snowflake.png`.

⟨create a snowflake 2⟩ ≡

```
Koch-snowflake 100
save-canvas "koch-snowflake.png"
```

While `save-canvas` is implemented by the graphics library, we will need to define functions that implement the snowflake. These include the `Koch-snowflake` function we have already alluded to in the previous chunk and the `Koch-curve` function on which it is based.

⟨program functions 3⟩ ≡
  ⟨Koch snowflake function 7⟩
  ⟨Koch curve function 8⟩

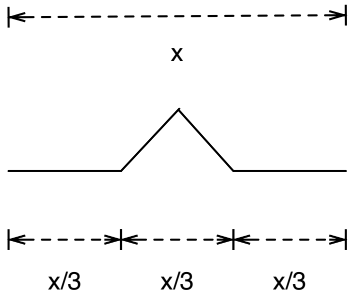
The program structure is then a simple matter of providing the function implementation and using that implementation to create the desired output.

⟨program to display a Koch snowflake 2⟩ ≡
  ⟨program functions 3⟩
  ⟨create a snowflake 2⟩

In the remaining sections we will delve into the process of creating fractal curves and snowflakes.

3. Koch Curves and Bump Fractals

A Koch curve is a "bump fractal." The general recipe for generating a bump fractal is to draw the fractal at one level of recursion and then replace each `forward` call with a recursive call. The Koch curve is based on a single triangular bump illustrated here:



By thinking like a turtle we can easily come up with the corresponding drawing code which is relative to the horizontal measure or extent, `x`.

```
forward (/ x 3)
left-turn 60
forward (/ x 3)
right-turn 120
forward (/ x 3)
left-turn 60
forward (/ x 3)
```

We generate the recursive case by using the bump fractal recipe and replacing the `forward (/ x 3)` calls with `Koch-curve (/ x 3)` calls.

⟨recursive case 5⟩ ≡

```
Koch-curve (/ x 3)
left-turn 60
Koch-curve (/ x 3)
right-turn 120
Koch-curve (/ x 3)
left-turn 60
Koch-curve (/ x 3)
```

A Koch curve has infinite length since each recursive step generates four new segments one-third the length of the original segment. The total length of the curve becomes one-third longer with each recursive step [2]. Stated more formally, the length of the curve at step n is $(4/3)^n$. A related measure, the fractal dimension, describes how how the complexity of the fractal increases as it scales. The fractal dimension of a Koch curve is $\log 4/\log 3$ or approximately 1.26.

Though the fractal has infinite length and is composed of an infinite number or segments, the resolution of our display is finite. It is convenient to end the recursion at the smallest representable length - a pixel. Our base case is then to simply draw a line of length `x`, where `x < 1`.

⟨base case 6⟩ ≡

```
forward x
```

We combine the base case and the recursive cases to form our Koch-curve function that generates a single Koch curve whose horizontal measure is `x`:

⟨Koch curve function 7⟩ ≡

```
define Koch-curve (x)
  if (< x 1)
    ⟨base case 5⟩
  else:
    ⟨recursive case 6⟩
```

4. Koch Snowflake

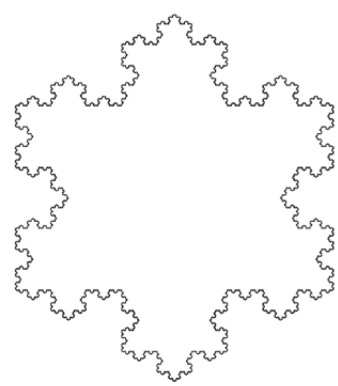
The `Koch-snowflake` function is trivially implemented by repeating three Koch curves to form an equilateral triangle.

⟨Koch snowflake function 8⟩ ≡

```
define Koch-snowflake (x)
  repeat 3
  Koch-curve x
  right-turn 120
```

5. Results

After the program is executed, the following image is generated.



References

[1] Donald E. Knuth. Literate programming. The Computer Journal, 27(2):97–111, 1984.

[2] H. von Koch, "Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire," Arkiv för Matematik, vol. 1, pp. 681-704, 1904.

2.4 Code Chunks

The base unit for most literate programming systems is the chunk. In Ginger, a *code chunk* is a labeled piece of code that may include ordinary Ginger code or references to other chunks. The `chunk` function is used to implement code chunks and takes two arguments: the chunk name (a string) and the code itself (a G-expression that may include chunk references). Chunk references are formed with the `$` function, which takes the chunk's name as its single argument.

Section 2.1 contains several different examples of code chunks and chunk references. Lines 16-17 illustrate a code chunk that simply references another code chunk. In this case the * denotes a special code chunk that serves as the root of the program. The chunk on lines 142-147 mixes reference to other chunks with ordinary Ginger code.

2.5 Documentation Chunks

Documentation chunks need not be explicitly defined like code chunks. They are simply the blocks of codes that surround code chunks and develop the documentation. One of the most common documentation chunks is defined with the `doc` function, which takes a single G-expression argument. Other functions like `title` and `section` also produce documentation chunks.

Superficially, many of the documentation commands look and act like TEX or LATEX commands but often the syntax is slightly different and Ginger's document model has been deeply influenced by docbook and HTML.

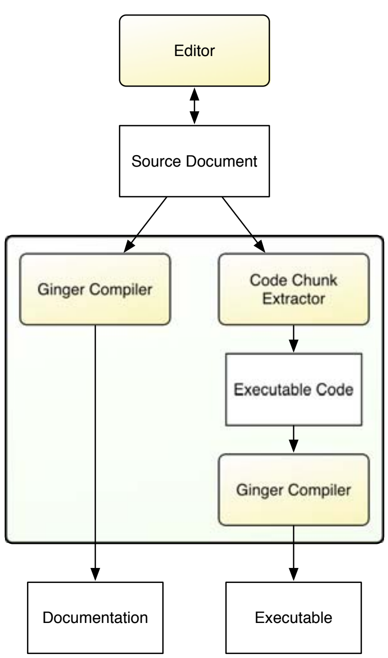


Figure 1. Source documents are transformed into documentation or code by manipulating how G-expressions are evaluated.

2.6 Untangling programs and documentation

The code and documentation chunks described in the previous two sections connect to each other forming a *web* of connections and content that ultimately defines both the program and the description of the program. As with other literate programming systems, Ginger must weave and tangle this literate web to extract the usable program and documentation. While the process illustrated in Figure 1 is similar to other literate systems its implementation is quite different. All of the subprocesses in the middle box happen within the Ginger compiler; each transition save the last to either documentation or executable works on in-memory G-expressions. G-expressions are manipulated such that their evaluation forms one or more programs or documentation. End users need only know that the Ginger compiler can target executables or documentation and each can be generated with a single invocation of the `ginger` compiler command.

3. Implementation Details

Ginger's simplifying assumptions that unify a single syntax used for code, documentation and literate glue also simplify the actual implementation. Since G-expressions implement every aspect of the literate program, we can simply manipulate these hierarchical data structures to generate a set of G-expressions that generate code or a set of G-expressions that generate documentation.

Ginger's `read` function plays triple duty; parsing literate documentation, code and chunks in a single step. Evaluating the resulting tree directly yields to the literate result. To extract the actual program we add each chunk definition to a dictionary keyed with the chunk's name. We then begin substituting chunk references in top level nodes with their respective definitions. We continue this substitution process until no more substitutions can be made. The resulting tree can then either be evaluated or compiled.

While our current work focuses on Ginger as the base programming language, the base language can be any G-expression based language. In other work we have experimented with the development of non-functional and non-homoiconic languages based on G-expressions. The literate programming system described here could be used almost transparently with such languages.

4. Challenges and Future Work

In this paper we have describe the literate programming system used in the Ginger language. While literate programming in Ginger shares many commonalities with other literate programming systems it unifies the literate programming experience with a single language, which is based on a powerful homoiconic G-expressions syntax. Literate programs written in Ginger use a single parser that constructs a G-expression based tree that can be trivially transformed such that evaluation generates either human readable documentation or computer executable code. While this has the effect of simplifying the programming experience for users it also makes powerful inspection and manipulation of both documentation and code possible.

Both the Ginger language and information about this evolving work are available at `http://ging3r.org`.

References

[1] K. Beck. A theory of programming. *Dr. Dobb's Journal*, Nov. 2007.

[2] A. Binstock. Interview with Donald Knuth. *InformIT*, Apr. 2008.

[3] M.-J. Dominus. POD is not literate programming. `http://www.perl.com/pub/a/tchrist/litprog.html`, Mar. 2000.

[4] R. Fjeldstad and W. Hamlen. Application program maintenance study: Report to our respondents. *Tutorial on Software Maintenance*, 1982.

[5] P. Grubb and A. A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific Publishing Company, Sept. 2003.

[6] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on the history of programming languages*. ACM, 2007.

[7] A. L. Johnson and B. C. Johnson. Literate programming using noweb. *The Linux Journal*, Oct. 1997.

[8] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[9] D. E. Knuth. *Literate programming*. Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[10] M. W. Mengel and P. Briggs. Nuweb home page. `http://nuweb.sourceforge.net/`.

[11] G. Milde. PyLit home page. `http://pylit.berlios.de/`.

[12] J. D. Palmer. Ginger: Implementing a new lisp family syntax. In *Proceedings of the 47th Annual Southeast Regional Conference*. ACM, Mar. 2009.

[13] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[14] V. Pieterse, D. G. Kourie, and A. Boake. Literate programming to enhance agile methods. In J. Eckstein and H. Baumeister, editors, *XP*, volume 3092 of *Lecture Notes in Computer Science*, pages 250–253. Springer, 2004.

[15] N. Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[16] J. Raskin. Comments are more important than code. *ACM Queue*, 3(2):64–ff, 2005.

[17] E. Ream. Leo home page. `http://webpages.charter.net/edreamleo/front.html`.

[18] J. Sametinger. Object-oriented documentation. *ACM SIGDOC Asterisk Journal of Computer Documentation*, 18(1):3–14, 1994.

[19] J. Sametinger and A. Stritzinger. A documentation scheme for object-oriented software systems. *ACM SIGPLAN OOPS Messenger*, 4(3):6–17, 1993.

[20] D. D. Smith. *Designing Maintainable Software*. Springer, May 1999.

[21] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, September 1984. Special Issue on Software Reusability.

[22] R. Williams. FunnelWeb home page. `http://www.ross.net/funnelweb/`.