Thinking with Style

Writing Software Books

Herein lays a journey past
the Phoenicians, the Greeks, the Romans,
past Gutenberg, Manutius, and Gill,
into the 21st century where the typesetter's soul
guides the computer programmer.

Daniel Lanovaz

Los Gatos, California

May 3, 2004

Thinking with Style

his article describes how ideas long used in the art of printing, word processing, and desktop publishing enhance computer programming: writing *software books*, compiling software books into programs, and archiving software books. Styles guide book compilation. Style templates dictate the book's consistent look-and-feel. Revision tracking, commentary, and embedded objects such as images, diagrams, and video enrich the software book's expressive power. We must balance learning to discover and create with learning to communicate and educate. It is time to dispose of your archaic text editors and start to author software novels.

Introduction

> ***style*** *(noun)*
> *3. way of writing or performing: the way in which something is written or performed as distinct from the content of the writing or performance.*
>
> ***Encarta Dictionary: English (North America)***

Everyone has his or her own style. Everyone has his or her own taste. A rare breed possesses a combination of elegant style and refined taste, attributes that are largely subjective and difficult to measure. Yet to meet one of these people, their rarity becomes immediately evident.

A decade of my life, now long past, was spent inside the fascinating world of Smalltalk and Lisp systems whose writing instruments were based on graphical bit-mapped display technology, fonts, and style metrics similar to many of today's word processing systems. At their foundation, however, the source code editors in these rich systems degenerated into a text editor not much different from *vi*[1] or *Emacs*[2]. At least Lisp systems introduced syntax-directed editing to manage all those parentheses.

No matter how evolved the graphical display system, no matter how evolved the input system, no matter how powerful the computing platform, computer programming has, unfortunately, remained largely

---

[1] Born from the hands of Bill Joy, *vi* is part of a class of electronic document editors known as *text editors*. Heralded for its simplicity and ubiquity, *vi* is still in wide use today.

[2] Another famous text editor is *Emacs*. First conceived by Richard Stallman, *Emacs* made popular the programmable editor, multiple windows and buffers, and the ability to turn a text editor into a multifaceted tool. Many versions of Emacs were produced over the years, including Multics Emacs, Gosling Emacs, GNU Emacs, Epoch, Lucid Emacs, and XEmacs.

Copyright © 2004 Daniel Lanovaz

unchanged. At its heart, the author manipulates sequences of characters and lines, not media rich pages within books.

This article introduces *software books* and describes five principles that guide their creation:

1. **The story is more important than the source code** — source code is supplementary text supporting the structured and media rich software book.
2. **Writing reduces spaghetti thinking** — just as structured programming reduced spaghetti code, writing software books reduces spaghetti thinking[1]. Writing down clear and concise thoughts is not only very difficult, but also very important in the dissemination of ideas.
3. **Use style templates to enforce stylistic consistency** — styles and style preprocessing allow rich and expressive rendering of the software story and supporting source code ("what you see is not what you compile"). A style compiler generates programs from software books. Styles guide not only the on-screen look, but the transformation ("*style preprocessing*") of the book into executable code.
4. **Use a standard representation to store and manipulate software books** — forty years ago the choice was s-expressions. Today the choice is the Extensible Markup Language (XML). I guess we have progressed over forty years, trading parentheses supporting a sound mathematical design for angle brackets supporting an ad-hoc design.
5. **Publish your software books for others to learn and enjoy**, or protect your software books as you protect other important intellectual property; the choice is yours.

I describe a software book authoring system based on Microsoft Word 2003. Word 2003—one of the most widely used "What You See Is What You Get (WYSIWYG)" documentation tools— was chosen because of its genealogy[2], its compelling feature set, its extensibility and programmability, and its ability to render documents in XML. Word 2003 is, in short, a powerful tool.

This article, however, is *not* just about using Word as a source code editor. I introduce software books and describe their construction using five guiding principles, but I ultimately want readers to realize these principles using an array of tools, from open source systems such as

---

[1] I use the phrase "*spaghetti thinking*" or "*spaghetti thought*" to draw parallels with the phrase "*spaghetti code*"; disorganized and unstructured thought that results in confused, illogical, and difficult to understand thought processes and/or conclusions.

[2] Microsoft Word is arguably a direct descendent of the people, ideas, and work done at Xerox PARC in the '70s and '80s.

 Copyright © 2004 Daniel Lanovaz

OpenOffice[1], to commercial publishing systems such as Adobe® FrameMaker[2], to custom designed software book editors[3].

Writing Software Books

This article is itself a software book. You can compile this book into an executable program[4]. You will notice, however, several features that are different from typical source code:

1. This software book contains sections, pages, and paragraphs.
2. Pages contain a header, footer, margins, and a body.
3. Styles guide the look of objects on the page.
4. Pages contain other embedded objects such as pictures, drawings (see below), and even embedded media objects such as sound recordings and video.
5. Pages may contain revision and editorial mark-up.
6. Pages contain cross references to other parts of the book, and may contain cross references to other software books.
7. A "What You See Is What You Get" (WSYWIG) editor (Word 2003) created the software book.
8. Books may optionally be protected using digital rights management (DRM).
9. The book is stored as XML.

The use of styles to guide the structure and display of pages is certainly not original. A page in a software book is a sequence of content objects with styles applied to those objects. I use styles to separate source code from supporting software book objects. A style sheet details how the editor renders content on the screen. The style sheet also guides how to compile the book. Figure 1 depicts how styles are like layers of onion paper, with the source code at the lower layer, the style sheet applied in

---

[1] http://www.openoffice.org

[2] http://www.adobe.com/products/framemaker/main.html

[3] Software books as an open-source project.

[4] This software book may be the longest "hello, world" program ever written, based on the venerable and oft cited example from "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie.

the middle layer, and the actual graphical representation of objects in the top layer.

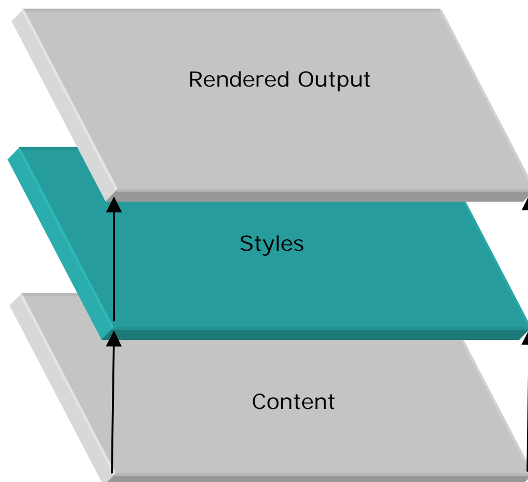


Figure 1 Styles

Styles—contained with a template—are the foundation of a software book. Styles differentiate objects on the page from the supporting source code. It is the source code styles whose associated content is compiled into the final executable. In the section “Compiling Software Books” on page 4, I describe the use of styles to compile books into executable programs.

Writing software books is as much a way of organizing and expressing ideas as it is writing the instructions that allow a computer to execute those ideas. Donald Knuth wrote many years ago about the need for “Literate Programming.”[1] Stephen Wolfram created Mathematica® Notebooks[2] that are a combination of book and mathematics engine that organize and express ideas with mathematical foundation. Finally, Charles Petzold, a freelance writer who specializes in Windows programming, aptly described the relationship between writing a book and writing source code commentary:

> The C# compiler has a terrific feature that lets you write comments with XML tags. However, I’ve chosen not to make use of this feature. The programs in this book tend to have few comments anyway because the code is described in the text that surrounds the program.[3]

---

[1] Knuth, Donald E. **Literate Programming**. Distributed for the Center for the Study of Language and Information. xvi, 368 p. 1992 Series: (CSLI-LN) Center for the Study of Language and Information - Lecture Notes.

[2] http://www.wolfram.com/products/mathematica/benefits/notebook.html

[3] Petzold, Charles, Programming Microsoft Windows with C#, Microsoft Press, 2002, page xxi.

 Copyright © 2004 Daniel Lanovaz

Software Book Structure

A software book's structure is similar to most books, comprising a title page, a table of contents, chapters, appendices, and an index. A software book may be a collection of separate books compiled into a master book. Figure 2 depicts the overall structure of a software book.

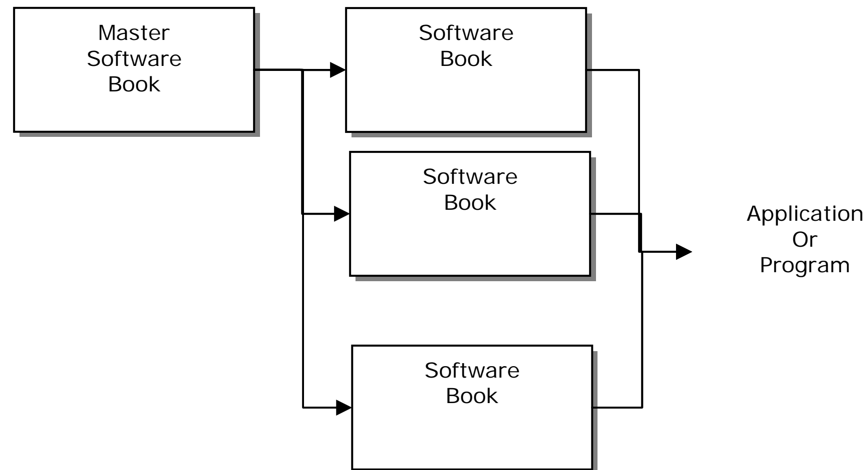


Figure 2 Master Software Book Structure

An individual software book is a set of pages, with each page divided into five sections (Figure 3).

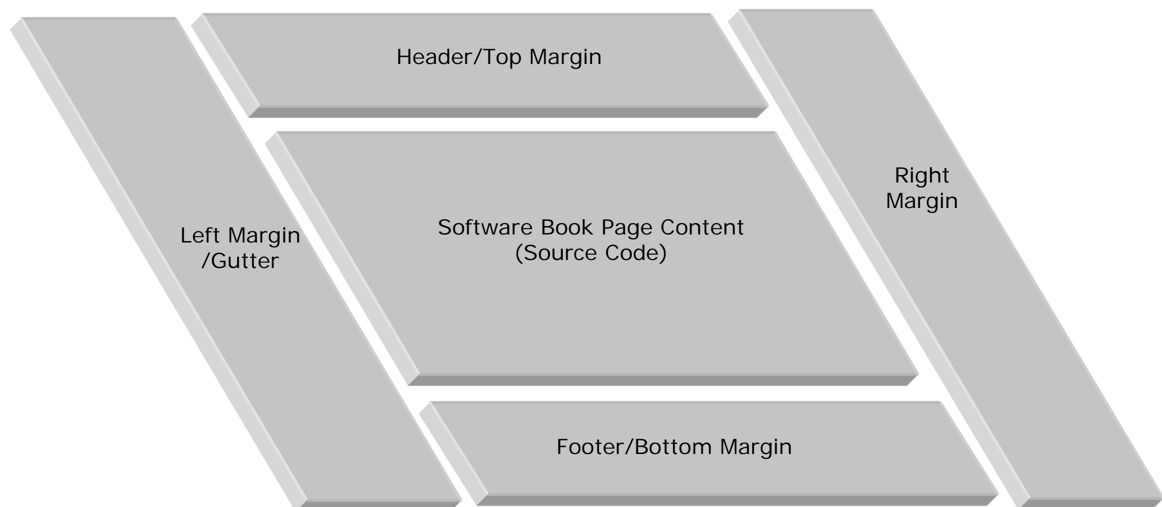


Figure 3 Page Regions

As with any word-processing document, page regions contain any object, from formatted text to embedded drawings to audio and video.

A "*software book compiler*" translates the master software book and subordinate software books into the target object. The target object varies depending on the programming language and desired output. Example target objects are executables, dynamic-link libraries, assemblies, shared

Copyright © 2004 Daniel Lanovaz

objects, or documentation. Yes, software books may simply compile into another book.

A software book is created using an editor capable of managing a style template, embedding objects within a page such as images, drawings, and other media objects, and managing the structure of pages, sections, chapters, and master books comprising a collection of software books (Figure 4).

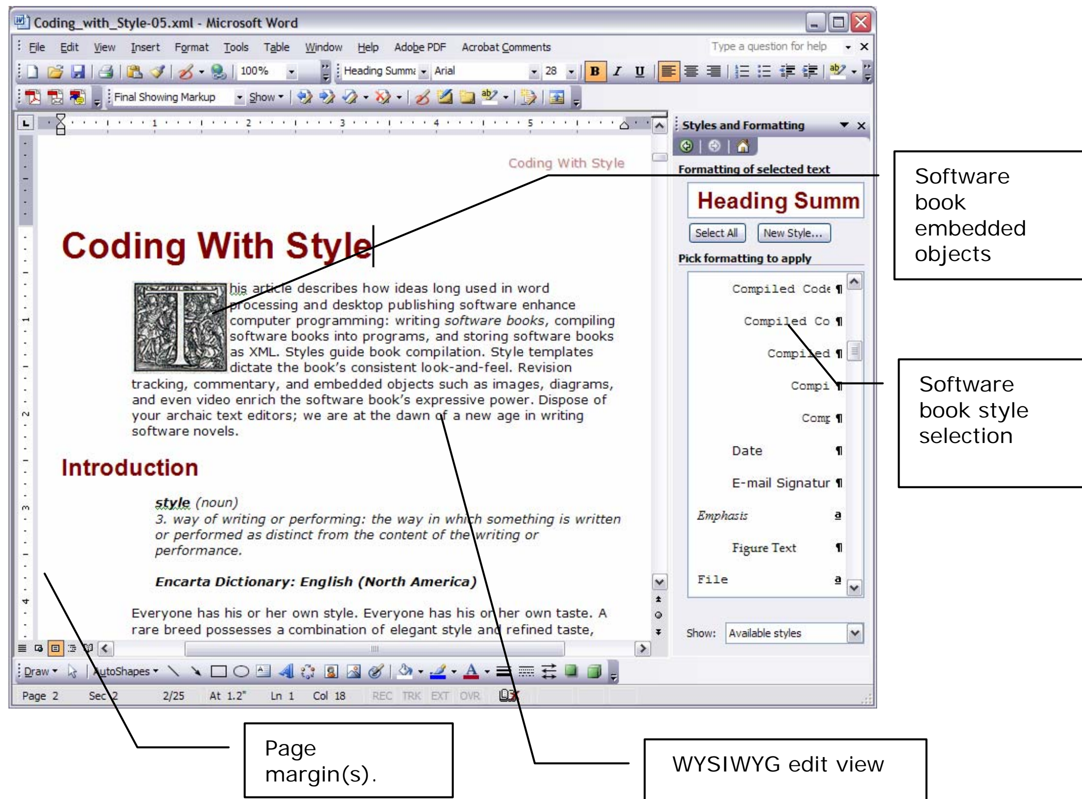


Figure 4 Software Book Editing

Software Book Features

In this section, I describe features that prove invaluable in writing software books. Authors have used these features for many years, and software developers should use these features during the design, construction, editing, and reviewing of their software books.

Revision Control

Arguably, one of the most useful features in many word-processing applications is revision control: the ability for the editor to not only

manage change between versions of a document, but also display those changes to the reader in a concise and graphical manner.

Revision control as used in word-processing is a powerful tool for the software developer. Imagine you are close to release. You have locked your software books into your software book control system. You fixed a defect, but you want to ensure adequate review of the fix before allowing it into the system. You check out the appropriate software book for edit, make the necessary changes, and prepare the book for peer review.

Figure 5 shows a portion of a software book modified with revision control enabled. The software book's revision control tracks insertions, deletions, format changes, and comments for multiple authors. Change bars appear along the left margin.

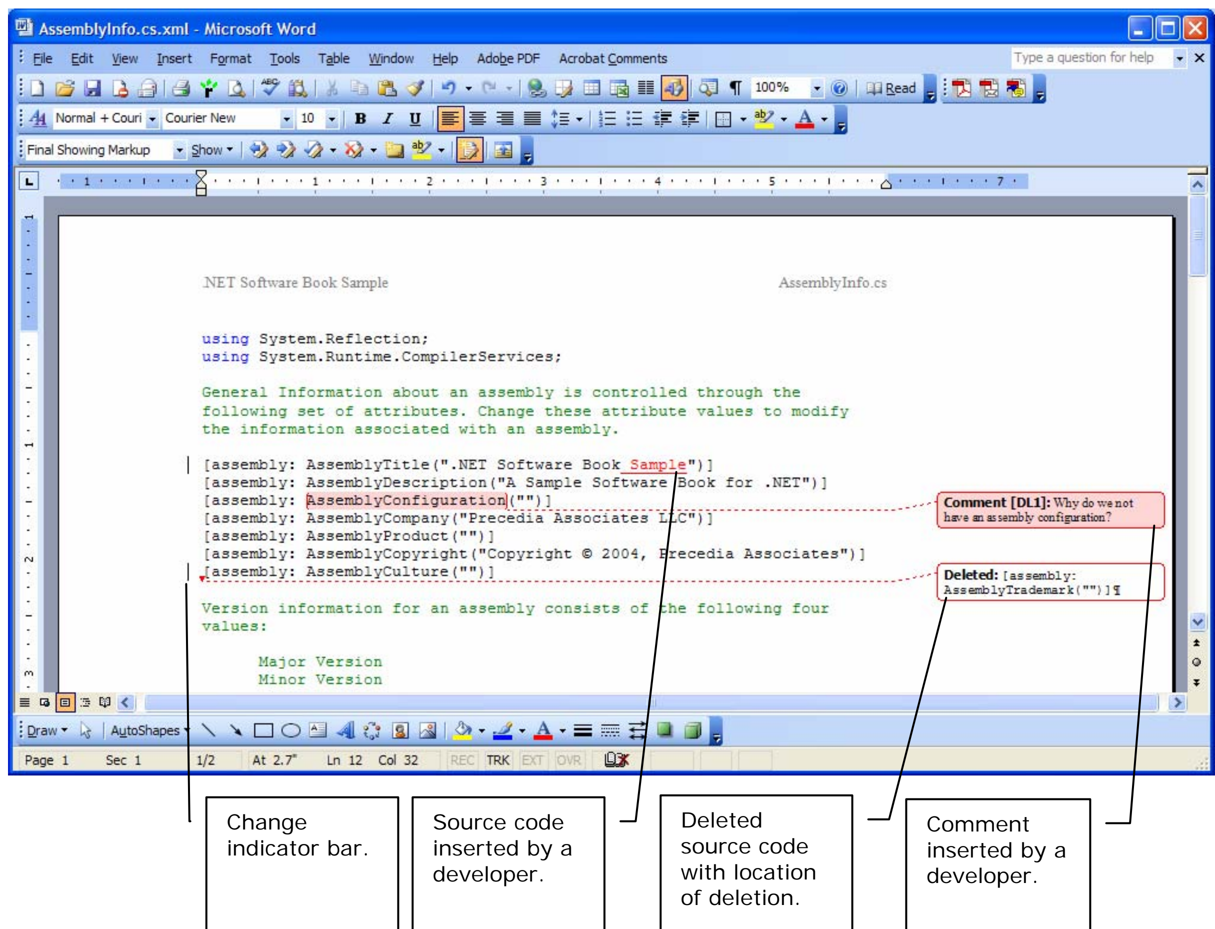


Figure 5 Source code revision control

The revision control system implemented by Word is very powerful, and in contrast to many traditional source code change management tools, provides an intuitive and rich mechanism to visually recognize changes. No more scanning Unix "diff" output looking for context.

 Copyright © 2004 Daniel Lanovaz

Footnotes

There are occasions when one would like to place additional information into source code, but not distract from the intent and flow of the code. Footnotes are perfect for this. The author can attach additional commentary, references, or supporting information, directly into an expression without affecting the compile-time and run-time semantics of the expression.

As described in the section "Compiling Software Books", footnotes are one of the many objects on a page discarded by the software book compiler.

Figure 6 is an example of a footnote inserted into a snippet of C# source code.

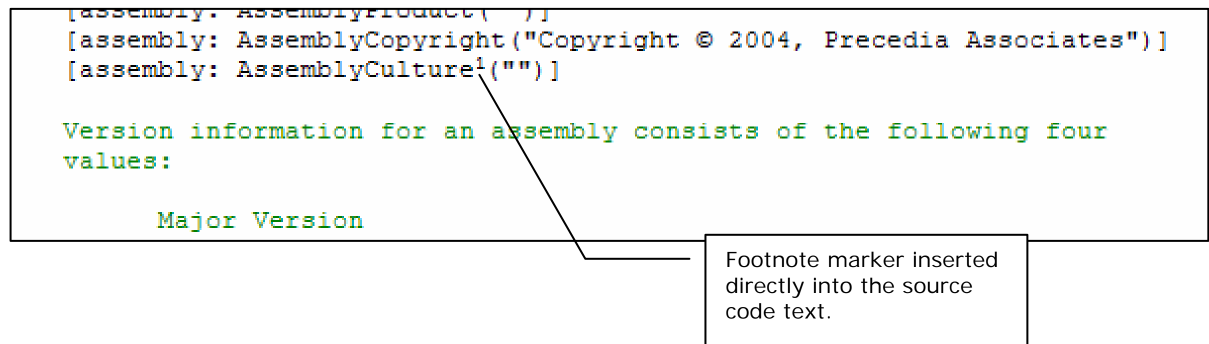


Figure 6 Footnote market

Figure 7 is the footnote text that appears at the bottom of the page.

Figure 7 Footnote text

Headers and Footers

Software book authors may want to ensure header and footer information appears on each page of the book, typically important information required when printing hardcopies of the book. Authors are free to place any text within the header and footer. For example, copyright notices, page numbers, or other identifying information such as the book name or project name. Figure 8 is prototypical software book footer.

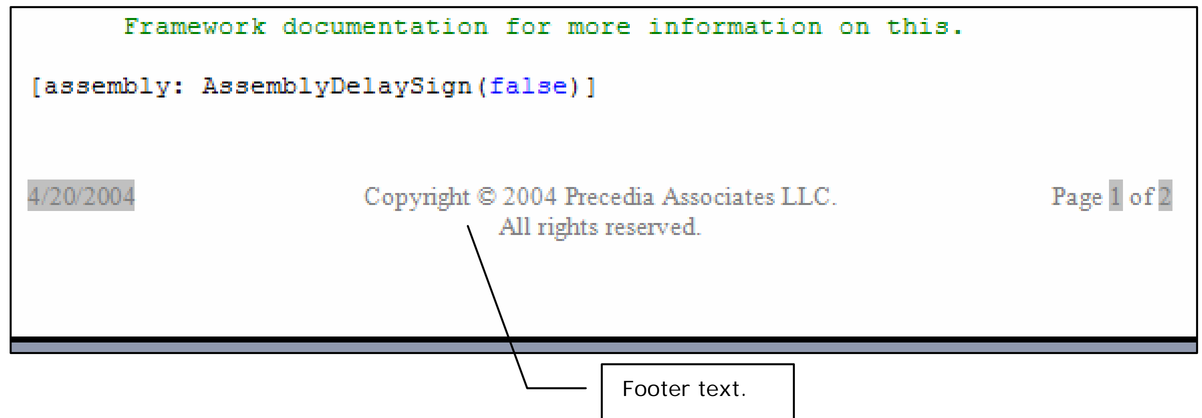


Figure 8 Software book page footer

Authors can directly edit the content of headers and footers (Figure 9), inserting custom fields such as the current date or current page number.

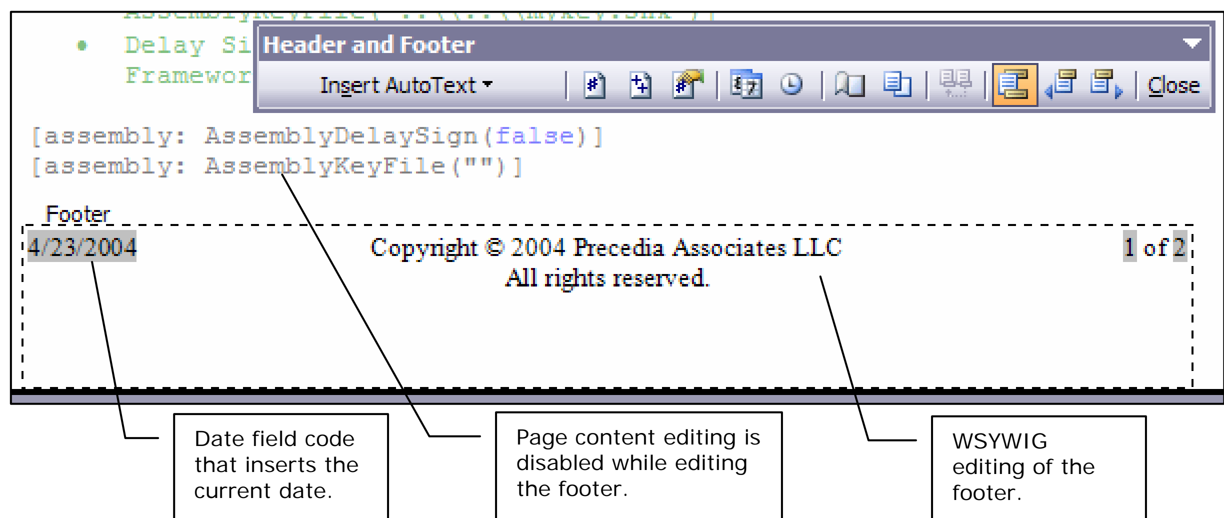


Figure 9 Page Footer

Embedded Drawings

Over the years, I have found it extremely useful to conceptualize structure and algorithms in the form of diagrams. Unfortunately, diagrams that I use to explain a portion of software come in two forms: 1) detailed

pictures created in word-processing software as part of requirements, specification, or design documentation or 2) diagrams embedded within source code.

Diagrams created within formal documents reside too far from the source code to which they pertain. Developers must make a mental mapping between a portion of the specification and the associated source code file. With software books, the requirements, specification, and design documents are the source code, compiled to form the resulting executable. In this way, software books group explanatory diagrams closer to the source code to which they apply.

When diagrams are embedded within traditional text-based source code, all too often they are created by using available ASCII characters (e.g. #, *, |, /, -, etc.). Figure 10 is an example of such a diagram, a snippet of C++ source code extracted from the Windows header file winerror.h. Note how the original author uses various ASCII characters to draw the enclosing boxes.

```
//
//  Values are 32 bit values layed out as follows:
//
//   3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
//  +---+-+-+-----------------------+-------------------------------+
//  |Sev|C|R|     Facility          |               Code            |
//  +---+-+-+-----------------------+-------------------------------+
//
```

Figure 10 Header File Diagram

There are four problems with embedding drawings like this within traditional text-based source code:

1. The diagrams are rudimentary.
2. The diagrams take too much time to create.
3. The diagrams are difficult to maintain.
4. The diagrams must be embedded within source code comments.

A software book takes a different approach by allowing professional diagrams to be drawn in situ using sophisticated drawing tools. More importantly, the developer can utilize drawing tools tailored to creating these rich diagrams. An example of this is the Microsoft Visio drawing package or the drawing tool included in Word. The software book compiler omits the drawing during compilation. In addition, drawings appear at any coordinate within the source code page, located in positions that best augment and explain the intent of the source code.

Figure 11 is the same code snippet shown in Figure 10 but rendered within a software book. You should notice four things that differ between the software book version of the header file and the original header file:

1. Comment delimiter characters do not appear, replaced by text with the comment style applied.

Copyright © 2004 Daniel Lanovaz

2. The drawing tool is active showing a selected text box, demonstrating the active nature of the embedded drawing.

3. A table provides structure to the source comments.

4. The page header and footer contain relevant text.

winerror.h.xml - Microsoft Word

Windows Header Files — winerror.h

Error code definitions for the Win32 API functions.

```
#ifndef _WINERROR_
#define _WINERROR_
```

Values are 32 bit values layed out as follows:

Sev | C | R | Facility | Code

Bits 16-27, Bit 28, Bit 29, Bits 30-31, Bits 0-15

where

Field	Purpose
Sev	00 – Success 01 – Informational 10 – Warning 11 – Error
C	Customer code flag
R	Reserved bit
Facility	Facility code
Code	Facility's status code

Define the facility codes

```
#define FACILITY_WINDOWS        8
#define FACILITY_STORAGE        3
```

Language comment delimiters are not required because of comment styles.

Activated diagram editor with selected text box.

Tables provide improved comment structure.

Figure 11 Embedded drawings

The current generation of Tablet PCs enable the author to draw diagrams freehand (Figure 12).

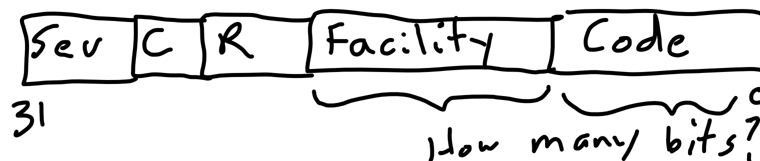


Figure 12 Hand drawn diagram

Copyright © 2004 Daniel Lanovaz

Embedded Objects

Creating media-rich compound document architectures is not a new idea. From Xerox, Taligent, Apple, and Microsoft, the ability to embed arbitrary objects within a compound document is a powerful and useful technique for organizing related information.

Microsoft arguably produced the most commercially successful compound document architecture with its Object Linking and Embedding (OLE) technology. Based upon Microsoft's Component Object Model (COM) framework, applications that support OLE, including this version of software books, can integrate third-party data and applications directly into the user-interface of the containing application.

Software books have a new dimension of expression and utility by providing the capability to embed within themselves application data such as Excel spreadsheets, Visio diagrams, audio commentary, and a plethora of third-party data and applications.

Imagine shipping a copy of your software book with an embedded full-motion video where you explain certain aspects of your system to the reader, or like I have done, embed an audio message to the reader.

Double-click the speaker icon to listen to the embedded audio message. Audio is just one of many examples of objects embeddable within a software book.

Embedded Execution

Imagine a software book where objects on the page are the execution of the story described in that book. Smalltalk systems such as Squeak[1] have used this technique for years, where one environment is both program creator and program executor and the distinction between active and inactive objects is much finer than in Microsoft's compound document architecture. Microsoft calls their technology in-place activation, where an object within a container application becomes active based on a selection gesture.

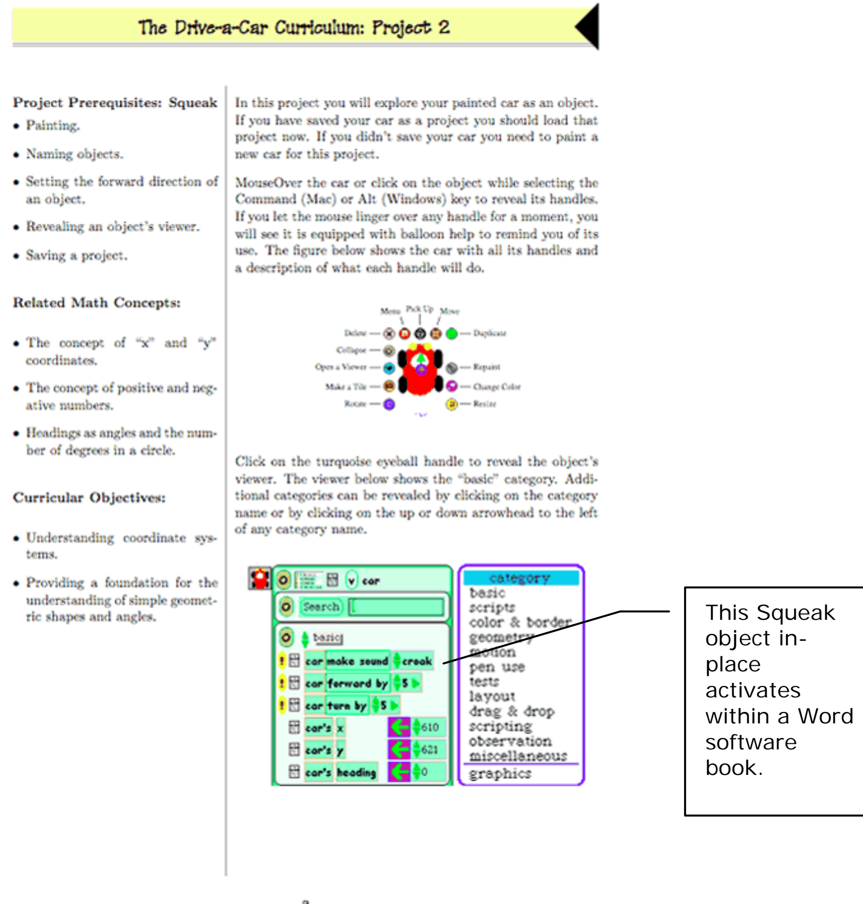
Squeak is a teaching tool. Figure 13 shows one page from a book entitled "Powerful Ideas in the Classroom"[2]. In a software book, this page would

[1] Squeak is a descendant of Smalltalk-80. More details are located at http://www.squeakland.org.

[2] Allen-Conn, B.J. and Rose, Kim, "Powerful Ideas in the Classroom," Viewpoints Research Institute, Inc., 2003

 Copyright © 2004 Daniel Lanovaz

look the same, but the embedded objects would "come to life," or in-place activate, when touched. Other sections and pages in the book would describe the code used to create these objects and be part of the book itself.



The Drive-a-Car Curriculum: Project 2

Project Prerequisites: Squeak

- Painting.
- Naming objects.
- Setting the forward direction of an object.
- Revealing an object's viewer.
- Saving a project.

Related Math Concepts:

- The concept of "x" and "y" coordinates.
- The concept of positive and negative numbers.
- Headings as angles and the number of degrees in a circle.

Curricular Objectives:

- Understanding coordinate systems.
- Providing a foundation for the understanding of simple geometric shapes and angles.

In this project you will explore your painted car as an object. If you have saved your car as a project you should load that project now. If you didn't save your car you need to paint a new car for this project.

MouseOver the car or click on the object while selecting the Command (Mac) or Alt (Windows) key to reveal its handles. If you let the mouse linger over any handle for a moment, you will see it is equipped with balloon help to remind you of its use. The figure below shows the car with all its handles and a description of what each handle will do.

Click on the turquoise eyeball handle to reveal the object's viewer. The viewer below shows the "basic" category. Additional categories can be revealed by clicking on the category name or by clicking on the up or down arrowhead to the left of any category name.

3

Figure 13 Page from a Squeak software book[1]

Writing software books in Squeak, however, requires either a Squeak virtual machine to in-place activate within Word, or a book authoring system within Squeak.

Three Dimensional Books

> *Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into*

---

[1] This image reproduced with permission from the author(s).

Copyright © 2004 Daniel Lanovaz

> *the book her sister was reading, but it had no pictures or conversations in it, `and what is the use of a book,' thought Alice `without pictures or conversation?'*[1]

From the age of six, my son pressed me to help write his first video game. He is fascinated by sporting games from Electronic Arts, absorbing the three-dimensional life-like world like a sponge, and masterfully manipulating the Xbox controller's six buttons, three joysticks, and two triggers, while at the same time synthesizing real-time movement of ten basketball players; five under his control. Now eight years old, he is already learning how to touch type and, interestingly, loves to type (in Word) short books of what he has learned.

A three dimensional world is an interesting place to create and store three-dimensional books. Pull the book off the shelf, open it up, and walk into it. Systems such as Croquet[2] may prove useful in creating three-dimensional worlds and the three-dimensional books authored by people, including my son, which both contains the world they create and describes that world for others to read, learn, and enjoy[3].

Commentary

Anyone who has written software is probably guilty of not writing enough documentation, writing useless documentation, or spending arguably too much time struggling with writing structured documentation.

Software developers have probably seen at one point in their career the well documented source code file. You know what I mean. The software team that decides on a standard file structure including header documentation and function prologue templates, complete with "Function:", "Parameters:", and "Return Value:" fields to be filled in by the programmer.

Someone always decides that a long series of '*' or '=' characters should be used as separators, and paragraphs of text should have line lengths no longer than eighty characters (or some arbitrary line limit).

I wish someone would perform a study to calculate the amount of valuable time wasted by developers trying desperately to make their comments look reasonable, splitting paragraph lines when new text is added, tabbing and spacing to align the programming language comment delimiters, or

---

[1] Carroll, Lewis, "Alice's Adventures in Wonderland," The millennium Fulcrum Edition 2.7a, © 1991 Duncan Research.

[2] http://www.opencroquet.org

[3] The Croquet document located at http://glab.cs.uni-magdeburg.de/~croquet/downloads/Croquet0.1.pdf should itself be a software book.

Copyright © 2004 Daniel Lanovaz

waiting as they hold the '*' key down to complete that next divider line. Unfortunately, I am as guilty as the next at wasting time performing this mindless task.

The industry did not stop there. Along came the Java programming language. Now developers spend their time not only tabbing, spacing, and comment delimiting, but a new series of key strokes—in the form of javadoc tags—add meta-data to the comments.

It did not stop there. Microsoft agreed that meta-data was required within comments, but decided to make them XML elements. Now developers are busy tabbing, spacing, comment delimiting, and typing hundreds of angle brackets to give meaning to their comments.

At least the Smalltalk and Lisp systems of thirty years ago had the sense to introduce paragraph editing that would remove the need for the programmer to format sentence length.

There must be a way to insulate the developer from the tedium of programming language comment and document structure syntax, yet still make available meta-data required to process the source code?

There is, in fact, a better way. Styles and templates provide a mechanism to layer meta-data on top of the source code. Not only does it allow a tool such as Word to alleviate the tedium of writing comments, but also allows software tools to process and reason about the source code. Here is how it works.

Every object in a software book has an associated style. A "*style preprocessor*" takes each object in the software book and based on its style decides how to render the output. Just like the C language preprocessor transforms one set of tokens into another set of tokens, the style preprocessor transforms a set of stylized content into another set of [optionally stylized] content.

A simple example is a programming language comment. All programming languages have them, but each language tends to introduce a different syntax to deal with them. Smalltalk uses double quotes. Java and C++ use '/*', '*/', or '//' character sequences. Perl uses the pound character. Visual Basic uses the single quote character.

Copyright © 2004 Daniel Lanovaz

```
Java
   /** A comment with <b>markup</b>
    * @param argument Is used for something.
    */
C#
   /// <summary>A comment with <b>markup</b></summary>
   /// <param name="argument">Is used for
   /// something.</param>
Smalltalk
   "A comment with markup. The parameter, argument, is used
   for something."
Objective-C[1]
   /*!
      @abstract A comment with markup.
      @param argument Is used for something.
    */
C++[2]
   // A comment with no markup.
   // Parameter:
   //        argument - is used for something.
```

A software book removes the need to deal with programming language specific syntax by applying styles to the text. The style preprocessor transforms the stylized text into language-specific tokens consumed by the language compiler. It is not necessary to implement a linear preprocessing and compilation sequence. Development environments, including Smalltalk and Lisp systems, have shown how incremental compilation and reflective facilities optimize this sequence.

The same comment written in a software book would look to the reader as follows (depending on the authors chosen style):

```
A comment with markup.
   argument — Is used for something.
```

The comment style applies to all the text. The word "markup" is an embolden style applied after the comment style. The word "argument" has a parameter style applied to it. Figure 14 shows the style structure.

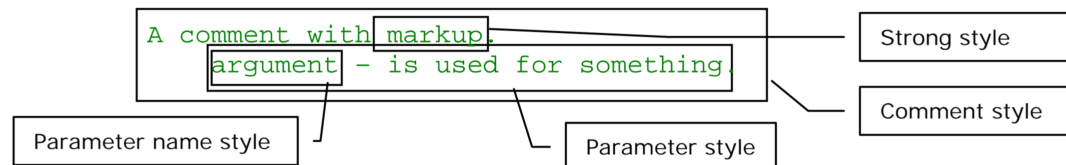


Figure 14 Comment Styles

[1] Apple HeaderDoc tags used in Objective-C, C, and C++ source code.

[2] Systems such as Doxygen are used to add structured comments to C++ source code.

The style preprocessor for a given language knows how to take the text of a comment style and render it in a programming language specific manner. The developer does not need to concern themselves with line lengths, paragraph editing, or inserting tabs, spacing, and comment delimiters within the text. Not only that, but the resulting source code is arguably much more readable than when language-specific comments and interspersed throughout.

Different style preprocessors can take the same software book and produce different output. A language style processor generates programming language output suitable for the associated language compiler. A document style preprocessor would take the software book and generate Application Programming Interface (API) documentation.

Ink Comments and Annotations

The current generation of Tablet PCs is one-step closer to realizing the vision of digital paper and personal digital assistants. Microsoft provides support for ink within Word 2003, enabling authors and editors to pen comments and corections within software books.

-> Corrections

Spelling and Grammar Checking

When it comes to errant spelling and grammar, I am one of the worst offenders. Every piece of software I own that supports spell checking has the feature enabled. Software book editors must provide spelling and grammar checking. Word has a simple and intuitive feature to highlight spelling and grammar errors by underscoring the offending text with a small squiggly line.

It is amusing that a Windows header file (winerror.h), probably more than a decade old and viewed/edited by hundreds, if not thousands, of developers, contains both grammar and spelling mistakes (Figure 15), recognized immediately when placed into a software book.

 Copyright © 2004 Daniel Lanovaz

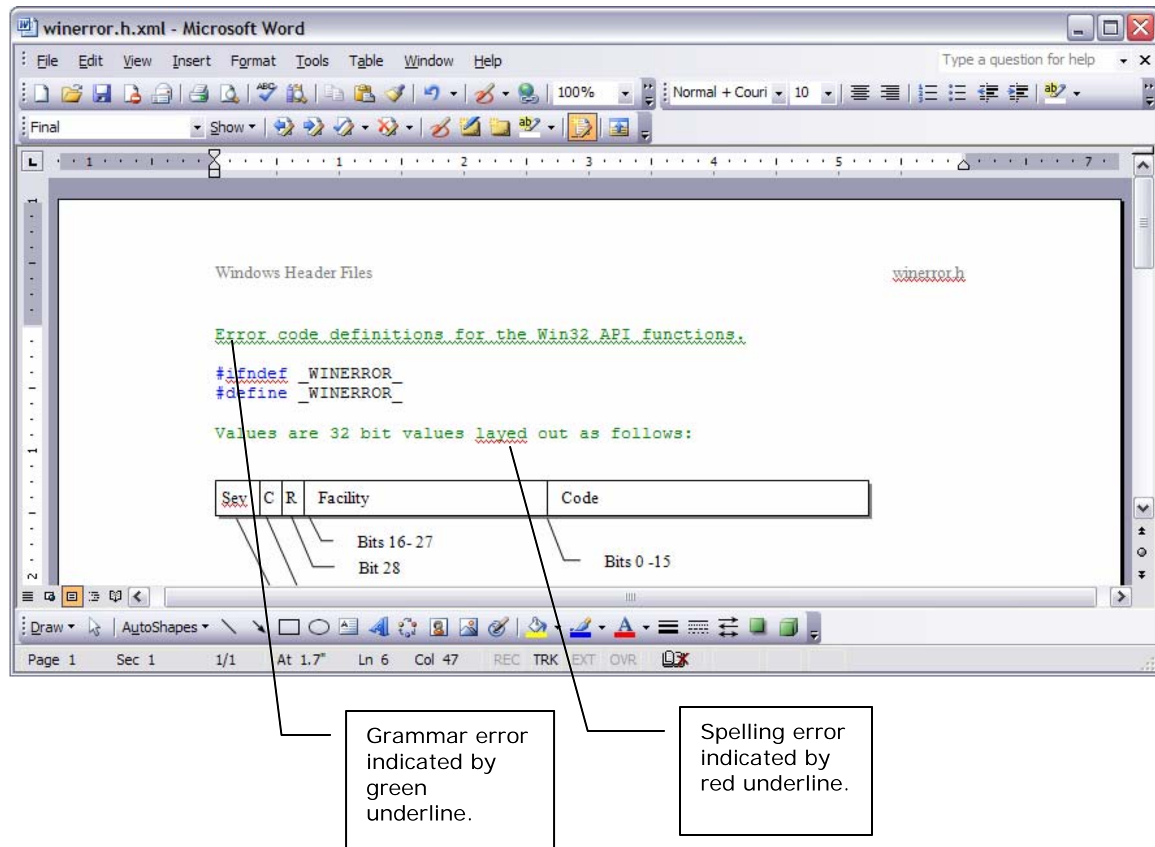


Figure 15 Spelling and Grammar Checking

Should I use tabs or should I use spaces?

A common feud in the software community regarding text-based source code files revolves around the question, "Should I use tabs or should I use spaces?"

Hotly debated with no clear winner, my answer is simple: use neither, use styles.

Compiling Software Books

Software books are stored on disk using the Extensible Markup Language (XML). In my Word 2003 implementation, software books are stored in Word XML using the Word XML Schema (WordprocessingML)[1].

---

[1] XML features are only available in Microsoft Office Professional Edition 2003 and stand-alone Microsoft Office Word 2003. For information of XML in Microsoft Office see http://www.microsoft.com/office/xml/default.mspx.

Word saves and loads XML documents just like traditional Word (.doc) files. All WordprocessingML files use the '.xml' file extension.

I created the Word template SoftwareBook.dot (the template used for this book) containing the styles required to format a software book. In addition, a set of Extensible Stylesheet Language (XSL) files transform a software book into the native source code piped into a programming language compiler. A "*software book compiler*" compiles, or translates, software books into the native programming language before feeding the output into the language compiler[1]. The translation process removes all style content that does not represent legal programming language text. This includes objects on a page such as headers, footers, footnotes, embedded drawings, embedded objects, and any text whose style does not represent programming language content. The following pseudo-code compiles a software book.

```
Foreach object in Book
   If (object.Style isKindOf CompiledCode)
         EmitForCompilation(object.Content)
```

The file extension ".<source>.xml" represents the XML version of the source code file. The following non-exhaustive table contains exemplary file extensions.

Extension	**Purpose**
.h.xml	C/C++ language header file
.c.xml	C language source file
.cpp.xml	C++ language source file
.cs.xml	C# language source file
.java.xml	Java language source file
.pl.xml	Perl language source file
.st.xml	Smalltalk language source file
.js.xml	JavaScript language source file
Makefile.xml	GNU Make
makefile.mak.xml	Windows make file
.swb.xml	Software book (programming language specified within the book)

Figure 16 depicts the general flow of compiling a software book into a target object.

[1] As noted earlier, software book environments that provide incremental compilation and linking, such as Smalltalk, would implement this preprocessing and compilation step in a manner much different than file-based compilation environments used by other language compilers (C, C++, C#, etc.).

 Copyright © 2004 Daniel Lanovaz

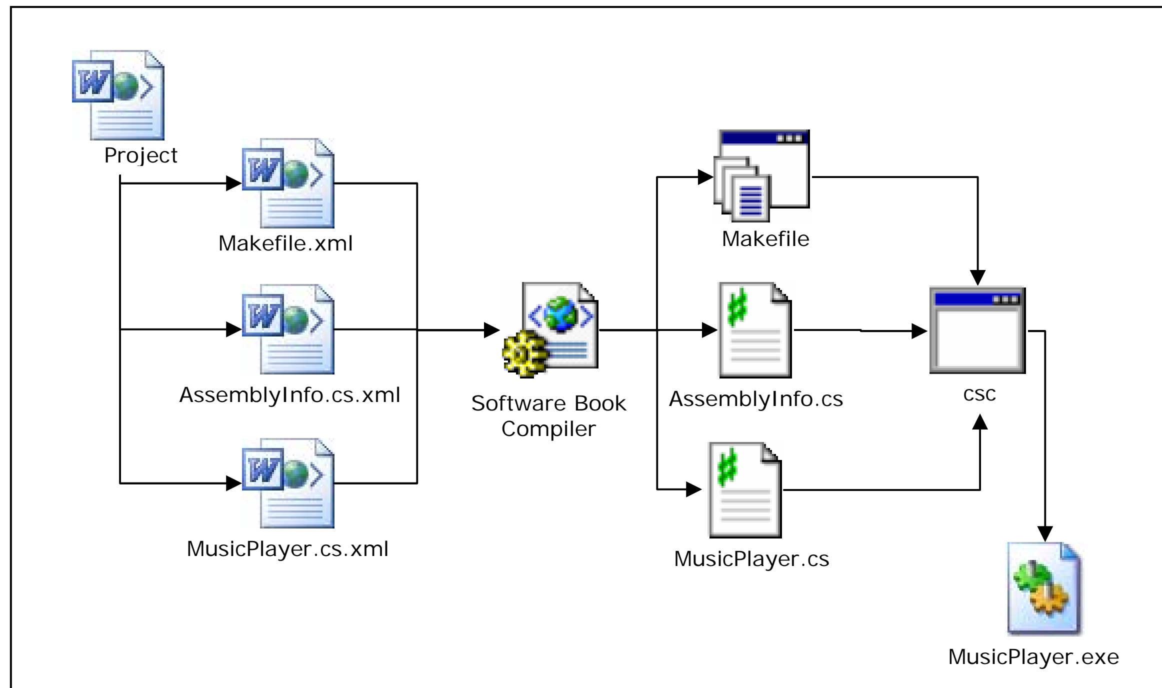


Figure 16 Software Book Compilation

Style Preprocessing

Style preprocessing is the transformation of software book content based on the content's style. Style rules dictate the format of the output.

Take for example a C# software book that contains source code interspersed with introductory comments, diagrams, and images. A C# style preprocessor walks through the software book and discards content whose style does not derive from the "Compiled Code" style, removing embedded diagrams and images, footnotes, endnotes, embedded review comments, and any embedded media objects. The C# style preprocessor may decide to transform the "Comment" style into syntactically valid C# comments, or it may decide to discard source comments. Finally, the C# style processor would emit all content based on the "Compiled Code" style in a form suitable for compilation by a conformant C# compiler.

If a special-purpose software book editor and compiler is created, this translation step is unnecessary as the software book compiler would process the XML elements directly rather than text output.

The one page software book in Figure 17 demonstrates how:

1. The developer writes source code comments without regard to the programming language's comment delimiters. Comments are continuous paragraphs, alleviating the need to worry about line length.

 Copyright © 2004 Daniel Lanovaz

2. Graphics are embedded within the source code to add useful supporting information.
3. Headings provide structure to the software book.
4. Headers and footers supply identifying information on hardcopy.

Heading style

Source code style

Embedded drawing.

Comment style

Source code style

Figure 17 Hello-World Software Book

The C# software book compiles the page in Figure 17 into the following:

```
using System;
public class HelloWorld {
   public static void main()
   {
         Console.WriteLine("hello, world");
   }
}
```

A C# compiler (csc) compiles this C# conformant program to produce the final executable (HelloWorld.exe).

Appendix B contains a version of this sample program, and is actual the source code for this software book.

Storing Software Books

A software book is stored on disk as well-formed XML. Programming language source that is typically stored in ASCII or Unicode form as a raw sequence of characters is now stored as a set of XML elements that conform to a well-defined XML schema. This enables the processing of complex software books not only by the native software book editors, but also by third-party programs.

Let us take a closer look at a very simple program written in C# (Figure 18) for Microsoft's .NET Framework.

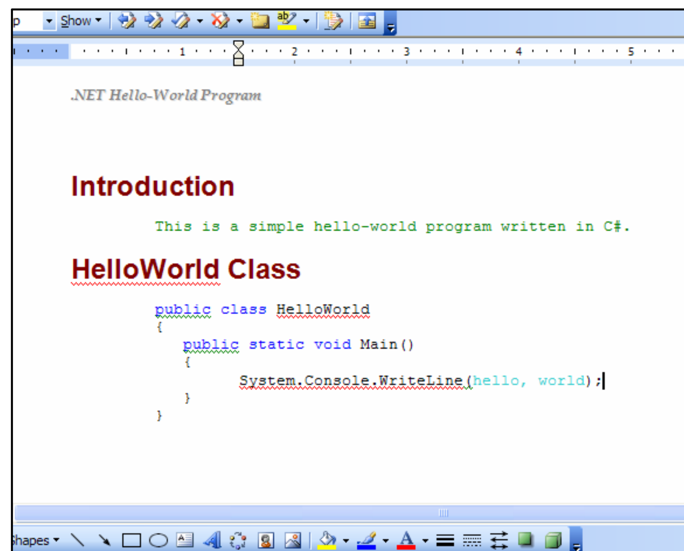


Figure 18 Simple Hello-World Program

This C# program does not look syntactically correct. A C# compiler certainly could not compile the header, ".NET Hello-World Program". Likewise, the green comment "This is a simple hello-world program written in C#" does not contain C# comment delimiters. Finally, the "hello, world" string as an argument to the WriteLine method does not include valid string delimiters.

 Copyright © 2004 Daniel Lanovaz

Each segment of characters, however, has applied to it a specific style that will allow the software book compiler to translate this text into syntactically valid C# code.

The stylized text is stored as XML elements with enough supporting information to understand the applied style, enabling a translation program to interpret the language constructs.

For example, the comment string is represented by the following XML element:

```
<w:p>
   <w:pPr>
         <w:pStyle w:val="Comment" />
   </w:pPr>
   <w:r>
         <w:t>This is a simple hello-world program written in
         C#.</w:t>
   </w:r>
</w:p>
```

In this example the "Comment" style is applied to a string of text. The <w:pStyle> element identifies the style name that should be applied to the subsequent element. The <w:t> element is the actual text string to which the comment style is applied. With those two pieces of information it is possible to write a translation program, or to write an XSL transformation, that takes the <w:t> element and appends comment delimiters before copying the given text.

As with the comment string, the "hello, world" parameter to the WriteLine method is represented using the following XML elements:

```
<w:p>
   <w:pPr>
         <w:pStyle w:val="CodeIndent2" />
   </w:pPr>
   <w:r>
         <w:t>System.Console.WriteLine(</w:t>
   </w:r>
   <w:r>
         <w:rPr>
               <w:rStyle w:val="StringChar" />
         </w:rPr>
         <w:t>hello, world</w:t>
   </w:r>
   <w:r>
         <w:t>);</w:t>
   </w:r>
</w:p>
```

The C# expression "Console.WriteLine(hello, world);" is broken into three main XML elements. The first element's style is "CodeIndent2" that indents the "Console.WriteLine(" statement two levels (no need for tabs or

spaces; just use an indenting style). The second element's style is "StringChar" and modifies the string "hello, world." The compiler transforms this element into a syntactically legal C# string constant by inserting string delimiters[1]. The third element completes the original C# expression by appending the closing parenthesis and semi-colon.

The style template (SoftwareBook.dot) contains additional information about each style such as the font color, font size, paragraph attributes, and other display-related information. This is why the source code comments above are displayed in green and the string argument in blue.

Style Hierarchy

Styles in a software book template have a hierarchical structure. The top-level style I have called "Compiled Code." All styles that modify source code derive from the "Compiled Code" style. The top-level code style is used by the style preprocessor during the book compilation process to extract relevant objects from the software book and discard those objects that could not be compiled by a traditional language compiler (such as a C# compiler).

Styles are stored in the style template. Styles are represented as XML elements. The following is an example style hierarchy as represented by Word. The following is a partial definition of the top-level "CompiledCode" style.

```
<w:style w:type="paragraph" w:styleId="CompiledCode">
   <w:name w:val="Compiled Code" />
   <w:basedOn w:val="Normal" />
   <w:link w:val="CodeChar" />
   <w:rsid w:val="00510BC4" />
   <w:pPr>
         <w:pStyle w:val="CompiledCode" />
         <w:autoSpaceDE w:val="off" />
         <w:autoSpaceDN w:val="off" />
         <w:adjustRightInd w:val="off" />
   </w:pPr>
```

A code style that indents text derives from the top-level code style.

[1] Note that the current system does not perform this syntactic sugar transformation, and describe this hypothetical string style preprocessing only for illustrative purposes.

 Copyright © 2004 Daniel Lanovaz

```
<w:style w:type="paragraph" w:styleId="CompiledCodeIndent1">
<w:name w:val="Compiled Code Indent 1" />
<w:basedOn w:val="CompiledCode" />
<w:rsid w:val="008313B9" />
<w:pPr>
   <w:pStyle w:val="CoompiledCodeIndent1" />
   <w:ind w:left="360" />
</w:pPr>
```

Note how the "CompiledCodeIndent1" style has a <w:basedOn> element that indicates this style is based on, or inherits, the style attributes of the "CompiledCode" style. The <w:ind> element indicates the indent level for this paragraph style.

The above XML elements are based on the WordprocessingML schema. It is possible to create an alternative language-specific schema where XML elements encode statement parse nodes. This allows software to manipulate structure of the software book in a more refined manner than the current scheme permits. For example, the expression "i = 42" may be represented as:

```
<assignment>
   <lvalue>
         <variable>i</variable>
   </lvalue>
   <rvalue>
         <number>34</number>
   </rvalue>
</assignment>
```

Appendix A contains a sample code style hierarchy.

Publishing Software Books

Writing software books is evolutionary, not revolutionary. Authors (developers) are free to mix a software book with traditional source code files, compiling the software books into intermediate files consumed by traditional compilers[1]. GNU automake tools can be used to perform the software book preprocessing step, converting the software book into a set of .h, .c, or .cpp files, depending on the project.

Software books can be checked into traditional source code control systems such as CVS, Perforce, or Visual SourceSafe. Managing revision control in these traditional source control systems is, however, problematic because these systems manage differences at the XML level and do not provide the graphically intuitive change tracking I described in the section "Revision Control" on page 7.

[1] Example compilers are gcc, javac, csc, perl, Smalltalk development environments, etc.

Digital Rights Management

Digital Rights Management (DRM) is technology that associates rights to use content directly with the content as opposed to more traditional access controls systems (file permissions, for example) that acts as gatekeepers to the content.

In rare situations where a software book contains important intellectual property such as trade secrets, new patent-pending algorithms, or business sensitive information, the software books themselves can be protected by DRM. Since the document itself is protected, there is no need to rely on system-level protections such as access control lists in the source code control system or file system. Authors simply restrict access to a software book based on their need-to-know requirements.

Office 2003 Professional supports Windows Rights Management[1] (WRM), a system to protect Office documents using Microsoft's DRM system.

You use WRM to restrict who has access to a document and what can be done with the document, such as print, view, edit, or even copy information from the document.

Outside of selling software books, I acknowledge the rarity of the situation where a software book needs DRM protection; I include a description of this capability because I believe that software books may contain intellectual property that is as important as other DRM protected content. In addition, the notion that the source code itself is protected, in the form of a software book, as opposed to access control rules implemented by file systems or source code control systems, adds a new dimension to source code security.

Conclusion

In this article, I describe how software developers need to rethink and retool how they write software. The story is more important than the source code. Writing reduces spaghetti thinking. The use of word-processing tools such as Microsoft Word and mark-up languages such as XML are an evolutionary step along the road to writing better, more readable, and more maintainable software. While learning to discover and create, we need to learn to communicate and educate.

"Talk of nothing but business, and dispatch that business quickly."

Manutius Aldus (Aldo Manuzio), Italian Printer and Scholar, 1447-1515.

[1] http://www.microsoft.com/windowsserver2003/technologies/rightsmgmt/default.mspx

 Copyright © 2004 Daniel Lanovaz

Appendix A

Style Hierarchy

The software book template contains the following code-related style hierarchy. The template also contains styles related to the formatting of the software book as opposed to the source code.

Character Styles

```
Comment
Keyword
Variable
Constant
   String
Method
Function
Statement
```

Paragraph Styles

```
Compiled Code
   Compiled Code Indent 1
         Compiled Code Indent 2
               Compiled Code Indent 3
                     Compiled Code Indent 4
```

 Copyright © 2004 Daniel Lanovaz

Appendix B

Introduction

Endowed by my author with anthropomorphic powers, I am a simple yet informative program that demonstrates the ideas behind my author's "Coding with Style" software book. I demonstrate some of my author's basic concepts, including using styles to embed source code within a software book, and the use of embedded objects such as diagrams to enhance the readability and maintainability of software books.

Figure 19 shows how I am compiled into an executable program. An XML representation of myself is scanned for embedded "Compiled Code" styles. When content with such a style is encountered, the content is transformed using a set of "style preprocessing" rules to generate source code recognizable by a traditional C# compiler. Of course, I am compatible with any programming language.

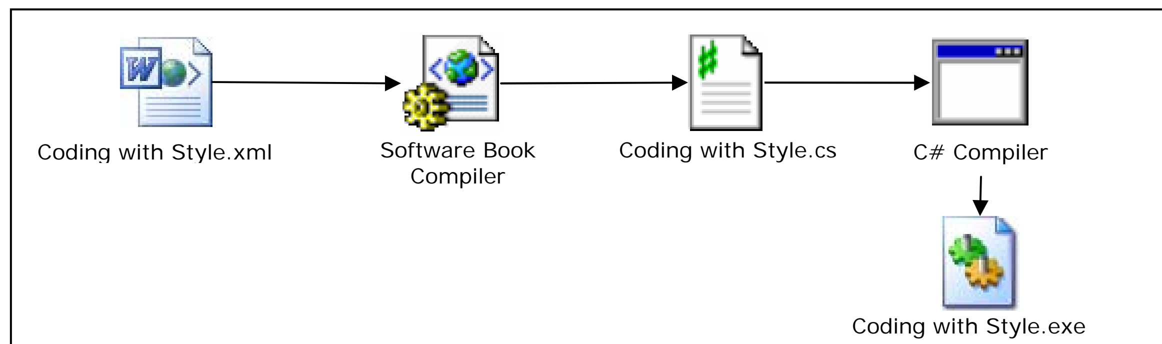


Figure 19 Hello-World Sample

Imports

```
using System;
```

HelloWorld Class

```
public class HelloWorld
{
   public static void main()
   {
         Console.WriteLine("hello, world.");
   }
}
```

 Copyright © 2004 Daniel Lanovaz