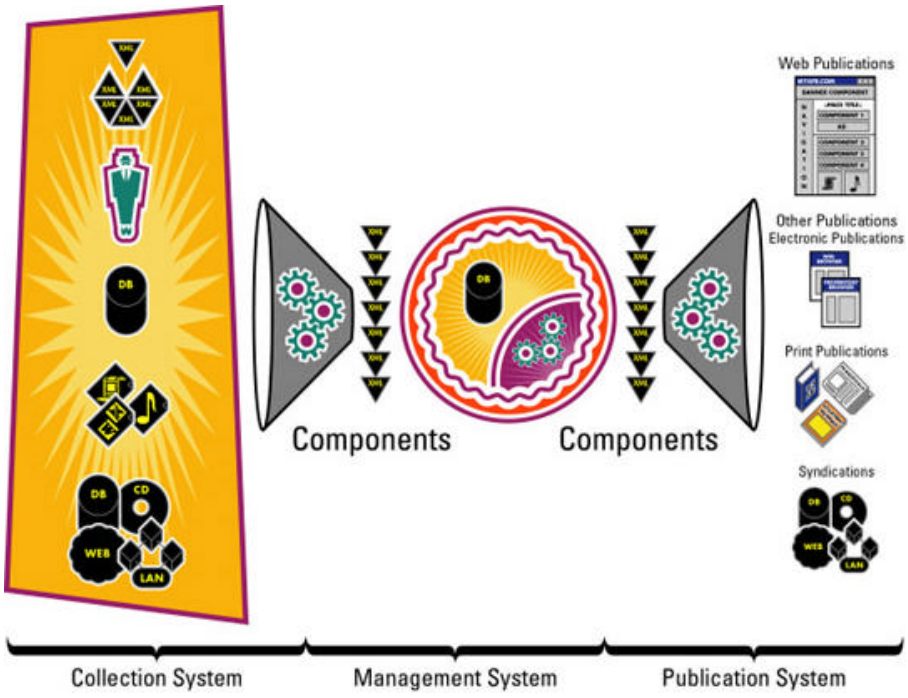
Processing Content

A CM Domain White Paper

By Bob Boiko



This white paper is produced from the Content Management Domain which features the full text of the book "Content Management Bible," by Bob Boiko. Owners of the book may access the CM Domain at www.metatorial.com.

Table of Contents

With a good grasp of markup languages, and especially XML, you can begin to dive into the mechanics of content processing. Loosely speaking, content processing is conversion. To do conversion you need to be able to fully parse (that is, get at) each markup tag of the source files. You must know exactly what markup you want in the target files or database, and you must devise a feasible plan for the transformation.

In this white paper I work through many of the issues that you may need to confront as you a plan and implement a content processing project. I try to stay as nontechnical as possible in the

beginning of the white paperbut end with a lot of programming code for the benefit of those who may need to implement content processing systems.

What Is Content Processing?

The vast majority of content that's today in HTML was once in some other form. My various companies alone have been responsible for converting hundreds of thousands of pages of word processing, desktop publishing, spreadsheet, database, print, and other forms of information into HTML. The vast majority of HTML is converted into XML. XML, believe it or not, will some day be overtaken by the next "best" format. Of course, there's still a lot of hard-copy information in the world that one day people are going to get around to digitizing. The sad fact that the format that you have isn't the format that you need is in no danger of changing any time soon. Add the even sadder fact that the structure that you want isn't the one that you're usually given, and you have all the reason you need to learn about and then master content processing.

Content processing focuses on the content conversion process but overlaps with authoring, acquiring, and aggregating. The point of content processing is to create low-cost and effective systems to do the following:

- Convert the format and structure of authored or acquired content into the format and structure that you need inside your CMS.
- Aggregate as much of the content as possible without the use of a metator.

If you can get a computer to do conversion and aggregation automatically, more power to you. If you find that you need people and process in addition to programs to get the job done, you ought to create a content-processing project that ensures that you get the most and best components for the least effort.

Because conversion is at the center of the content-processing world, I start by detailing the conversion process (see Figure 1).

***Figure 1**: The conversion process, showing the stripping and mapping that's involved*

If you strip content, you remove (and throw away) extraneous information. What exactly is extraneous depends on your particular needs. Here, however, are the sorts of information that are most commonly stripped:

- **From print files:** Front and back matter including titles, publishing information, indexes, end notes, and the like. In addition, headers and footers, including all the numbering and sectioning information that print files include, are generally stripped.
- **From Web files:** Surrounding banners, branding, and navigation and advertisement are generally eliminated.
- **Transitions, introductions, and other print devices:** Anything used to move the reader from one subject to the next. Because the online reader may jump directly into the middle of a discussion, these devices can end up confusing the reader more than they help.
- **The binary file format:** If the destination file format is different from the original. In converting from Word RTF to HTML, for example, the Font and Color tables at the beginning of the Word file are generally discarded.
- **Rendering formatting:** Any that isn't universally supported. The formatting codes that make columns of text in a desktop publishing program, for example, are often discarded because they're not easy to produce in HTML, and columns do not work well on the computer screen.

- **Structural elements:** Any that aren't used in the target publications. If the contents of an annual report, for example, are broken into separate chunks and distributed in various configurations, you have little need to keep the report's table of contents. Similarly, cross-references to parts of the work that aren't used or to sites that no longer exist can be discarded.

What's left after all the extraneous information is removed is the content to be included in the CMS. The structure and format of the remaining content, however, probably still needs to be mapped to the target format and structure.

Mapping is a process of using target markup to represent the formatting and structure of the source. If you're lucky, there's a good correspondence between the source and target markup. If you're unlucky, there are inherent ambiguities or discontinuities that can be resolved only by human judgment. In the section "Understanding the Principles of Mapping Content," later in this white paper I go into detail about the process of mapping source to target structure.

The conversion process could be anywhere from completely automated to completely manual. I've seen large teams of people do nothing all day but cutting and pasting text from one format and structure into another manually. On the other hand, I've seen (and written) programs that plow through megabytes of content in a few seconds, fully parsing and converting it. Most likely, you can automate a lot, but not all, of your conversion processes. At the very least, you need people to check the work of the computer to ensure that it's at the quality level that you expect and that exceptions are handled correctly.

Content processing projects define and implement processes for transforming content from a set of source formats to target formats that you need for a CMS. In pursuit of this goal you may do the following:

- Simply convert file formats.
- Strip and select content chunks and turn them into content components in a database or other storage medium.
- Conform source content to a particular DTD.
- Add missing metadata to content.
- Author missing parts of the content.
- Combine or split content sources.
- Develop a combination of automated and manual processes.
- Hire and train small or large teams to perform the process.
- Do the process yourself indefinitely or outsource it to an outside organization.

If you're lucky, you have only a few of these tasks to accomplish and they can be easily automated. In all likelihood, over time you need to all these tasks for some kind of content that you want to acquire. In addition you end up needing to involve people in at least some of the tasks some of the time.

Content Processing and the CMS

I haven't seen many commercial CMS products that address the core issues of content processing directly. To be sure, you can use the CMS workflow system, in part, to structure your conversion processes, and you may even get some conversion programs along with the product. By and large, however, CMS products expect that you do your processing outside their software and bring content in after it has the format and structure that you want it to finally have.

This isn't to say that content processing is outside the bounds of a CMS. Quite the contrary. In your planning and implementation of a CMS, you must take content processing into account. If

you don't account for processing, you're faced with large costs that seem to come out of nowhere and large bottlenecks in content collection as you try to deal with this potentially large issue casually.

Focusing on the long-term benefits

The basic idea behind content processing (and generally behind any automation) is to spend some effort up front to develop tools and processes so that you can reap much more benefit over the lifetime of the system. In other words, you're looking for a good Return on Investment (ROI) from the tools and processes that you create.

As simple as this idea is to understand, it's amazing how often it doesn't work. I've seen many examples of projects in which too much time and money were spent for the amount of savings that were accrued. From what I've seen, the main reasons for diminishing returns are people who do the following:

- **Rush to automate.** For some strange reason, automation projects are often driven by what can be done, not by what ought to be done. A programmer looking at a sample of the source content begins the conversation by saying what "can" be done. The conversation moves then to how to do it. The crucial unanswered question here, of course, is what *should* be done. In the rush to begin automating, many people forget to ask, "Is this program necessary?"
- **Underestimate the complexity of the task at hand.** It's easy to underestimate the complexity of a conversion process. Especially if you're going from unstructured to structured content, it's hard to overestimate the complexity of the task. *Unstructured* means without structure, variable, inconsistent, unpredictable, and generally hard to parse. Unstructured content lacks all the qualities that programs depend on to do their tasks without judgment or intuition. Novice programmers may think that they can make gold from content lead. The more times that they fail, the better they become at differentiating between the silk-purse content that's worth automating and the sow's-ear content that doesn't yield to any algorithm.
- **Underestimate the necessary amount of human effort.** A good processing tool does the tasks that require no human intervention and facilitates people to do the tasks that they need to do. A bad (and costly) processing tool tries to do too much of the human work and ends up costing more human work later. Suppose, for example, that the files in a directory are of two types that need two different sets of programs applied to them. If the processing tool guesses incorrectly 20 percent of the time, at some break-even volume it's worthwhile for a person to go through the files first to save the people later in the process from needing to find and repair the mistakes that the tool made. The programmer may produce a better ROI by creating a tool that helps a person categorize the files quickly rather than by trying and failing 20 percent of the time to categorize them automatically.
- **Provide poor sampling.** I've seen many processing systems fail because the designers failed to look at a truly representative sample of the content to be processed. A typical scenario is that the processing team gets a sample chosen by the contributor, because it was convenient to find. The sample may or may not be representative of the full load. By the time that the full load is available, it's too late. The tools are already designed to the wrong specification.
- **Fail to balance the cost vs. the benefits of a user interface.** The core code that actually does the processing can get dwarfed easily by the code needed to produce a full user interface. That's why so many CMS products require you to type parameters into configuration files and other unfriendly places. Programmers can easily get carried away making everything easy to do and ultimately configurable. Although it's always a good idea to provide a usable interface, usability must be balanced against cost effectiveness. Suppose, for example, that it takes a programmer an extra week of development to provide a graphical

user interface to a processing tool. Only five people use the tool, however, and you can train them in a few hours to run the tool without an interface.

- **Handle all exceptions programmatically.** The way that many programmers work is as follows: Write a conversion script or program and then try running it. As unanticipated situations arise, they write more code to handle them. In this way, the program can handle more and more of the exceptional conditions that arise in the source content. There's a point of diminishing returns in this process. Not all exceptional situations are worth handling programmatically. Suppose, for example, that a certain situation comes up only once a week and would take a day of programming to handle. To fix the problem may add only 10 minutes per instance to the work of a staff member. It would take more than a year for the time the programmer spent to be recovered (and that's assuming that the programmer is paid the same amount as the processor).
- **Change process in mid stream.** If tool and process creation overlap with tool and process use, expect an expensive process. The same content may need to be processed multiple times; schedules fall by the wayside; special tools need to be created to fix problems caused by the unfinished tools; and frustration rises sometimes to dramatic levels. In my experience, it's only rarely worth the extra effort to begin a content processing task before the process is completed and tested.

My intention isn't to scare you away from process automation but rather to provide you with some tools to make sure that it saves you more than it costs you. If the quantities to be processed are large, you can almost always structure your process so that it rewards your investment handsomely.

Suppose, for example, that you have 1,000 items to process. Without automation, say that it takes half an hour each to process them. That's 500 hours of processing time. Suppose that, with 40 hours of design and programming effort, you can cut the time down to 15 minutes per item (only a modest gain). Even if an hour of technical time costs twice that of a processing hour, you still save the equivalent of 170 processing hours, as the following formula shows:

(1000 items) x (.25 hours/item) + 2 x (40 hours of dev) = 330 hours

If 80 hours of development time can cut the processing time down to five minutes, you save the equivalent of 256 processor hours.

(1000 items) x (.083 hours/item) + 2 x (80 hours of dev) = 243 hours

I leave it as an exercise for you to figure out where the point of diminishing returns is. If you do much content process design, you become close friends with calculations like the ones that I illustrate here. As usual, of course, these calculations are only as good as the assumptions that they're based on, so you're well served by not taking them past their useful precision. There's no use calculating the savings to the nearest hour if your time estimates are good only to within 20 percent.

Here are some general guidelines to follow to ensure that your content processing projects stay on the right side of the cost/justification calculation:

- **Don't automate until you have a reason to do so.** Create a compelling case like the one that I illustrate in the preceding example before launching into any sort of development cycle. Make sure that you can establish manual processing benchmarks (like the figure of a half hour per item that I cite in the preceding example) that you trust. Then make sure that you clearly communicate how much development time you can spare given a particular gain in processing time. You may even turn the dry calculation into a sort of game, in which the process development team itself tries to optimize the development time and processing savings equation. As you calculate development time, don't forget to include the time that it takes to debug, test, and train staff on how to use the automation tools.

- **Start with the core algorithms.** Don't spend any time on user interface or exception handling until you're sure that the core processing code that you have in mind can do the job that you expect of it. Get a bare-bones tool together that only a programmer can run. Prove that it can meet the basic requirements of processing time gains before adding the nice-to-have features. Then, if you run out of budget for development, it affects the least essential parts of the tool first.
- **Run complete samples soon.** As soon as is absolutely possible, run the complete set of content through the automated processes that you've designed. Until you've run a sample 100 percent through the system, you can never be sure of the true gains that the tool may yield.
- **Test with real people.** Include the people who are actually to operate the tool as soon as possible. In the likely situation that you're building tools and process for a small number of people, it behooves you to tailor your product directly to their needs and abilities.
- **Create and review exception logs.** Rather than writing code or process for processing exceptions as they arise, track them for a while first. Tracking them tells you how frequent they are and how much time you can justify spending to handle them. It also forces you to think about the exception for a while before rushing to automate its handling.
- **Go for percentages.** Prioritize your process and tool development by expected gain. Always do the big gains first and the smaller gains only as time allows.

Focusing on the short-term benefits

Some of the most personally rewarding, well-recognized, and successful programming that I've ever done has been in what I'd call *kamikaze* mode. In kamikaze mode, you swoop in from outside the process, zero in on the center of the problem, and create just enough of a tool to relieve a big pressure point. Then you stop. Kamikaze automation provides pinpoint solutions to immediate and large problems.

Suppose, for example, that it's taking 20 minutes per item for the processing staff to find the right part of a set of HTML files to cut and paste into CMS components. You may begin a three-week project to study the source and develop an automated conversion system, or you may simply notice that it takes ten of the 20 minutes just to open each HTML file and find the content within all the JavaScript code. So, in a few hours (or less), you create a simple program that strips out all the JavaScript code from the source files. In three weeks, you may get the processing time down by 75 percent, but in three hours, you can reduce it by 50 percent! In another three hours, you can perhaps cut the task time in half again and reach the same goal (75 percent reduction) in six hours rather than three weeks.

Note

Kamikaze automation isn't appropriate for all situations. You need to look at each situation and see the quick fixes that are possible. In my experience, you always can find lots of small ways to increase efficiency if you really look. If you watch a group of people work, you notice where they spend their time and what small repetitive or mechanical processes they do (and may not even notice).

Here are some principles that you can try to facilitate the kamikaze automations in your group:

- **Stay close:** My best results at automation have been when I was right there in the same room as the people doing processing. There's no better way to learn about what tools are really needed than to listen to the constant complaints of the people on the front lines. There's no better motivator to create relevant tools than to have lunch each day with the people who must use the tools that you create. If the kamikaze automator is a protector and servant of the little guy, she ought to get to know them.

- **Make quick turnarounds:** The tool or process that you create ought to respond immediately to an immediate need. If a problem arises at 10 a.m. and you deliver a fix by 2 p.m., you lose minimum processing efficiency and gain a lot of respect and credibility.
- **Set aggressive incremental deadlines:** Focus on one-day or shorter tasks that can yield measurable gains. If a tool doesn't work out, you've wasted only one day. If it does work, it doesn't need to gain you much to justify a single day. The Kamikaze projects may be coalesced later into a control panel for processors or some other wider tool, but each should stand alone and provide quick, clear value.

Kamikazes are heroes. They address immediate needs, cut through bureaucracy, and work fast. It's fun for the automator, gratifying for the processing team to have someone pay that much attention to them, and it can be quite rewarding for the processing bottom line.

Distilling the essence of a process

Whether you use kamikazes or teams of long-term backroom automation programmers and analysts, the essence of a content process is the same. A good content process is as follows:

- **Comprehensive:** It includes tools (custom or commercial software), methods for using the tools, staff jobs, tasks, and workflows. Too often, a team designs a tool and forgets to figure out how it fits into the team or into the wider processes of which it's a part.
- **Efficient:** The time to process one unit of content (generally one component) is the least that it can realistically be.
- **Effective:** The quality of the content that comes out of the process is as good as it can realistically be.
- **Complete:** Whether all exceptions are handled automatically or manually, all exceptions should be handled. No (or at least a minimum of) content should be allowed to slip through the process without being adequately processed.

Your processes as a whole ought to work together well. There should be a minimum of conflicts in the staffing or workflow of all processes together. There ought to be no content eddy currents, where some content flows in a different direction or gets substantially different treatment from the rest.

Finally, there should be an overall organization to the processes so that software and methods are released to everyone at once and are always current. There ought to be no tweaks needed for a particular machine and only one current version of all significant lists and control files.

Bringing people into processing

If you have any significant amount of content processing to do, you'd better get used to including people in your considerations of how it can be accomplished and stay successful.

Here are some considerations to help you keep humans and humanity in your automated processes:

- **Intelligence vs. instruction:** The better the instruction that you supply, the less independent intelligence you need in the people whom you hire to do the manual parts of the process. Good documentation and instruction more than pay for themselves in increased accuracy and decreased experience or expertise required on the part of your processing staff.
- **Meticulousness:** Look for, expect, and reward a meticulous attitude in your processing staff.
- **Creativity:** Expect and reward a form of creativity that's found most among the meticulous - the ability to see how to become more organized or more efficient. Don't expect that, just because your processing staff may be inexperienced and not very technical, they can't

recognize a bad process if they see one or can't articulate a good solution. In my experience, most of the best improvement suggestions come directly from the processing staff.

- **Clarity and consistency:** It's much better for morale and the quality of the output if you stress consistency over exactness. It's much better for your team to make a mistake in processing in a consistent way than to change what they're doing part way through the process. If the mistake is indeed consistent, you can likely fix it with a quick script. Your staff requires readjustment time and increased quality control after every process change, so it's best to keep these changes to a minimum.
- **Appropriate tools:** Provide your staff with tools that are at - not above and not below - their skill level. A search-and-replace program, for example, is a powerful tool. Incorrectly used, it can wreak havoc. It's likely that some members of your processing staff are ready for such power, and some aren't. If you're smart, you recognize one sort of staffer from the other and provide each with the most powerful tools that she can be counted on to use correctly. I've seen unskilled processors rise quickly from being computer phobics to nascent programmers, because someone trusted them enough to let them write and use Microsoft Word macros.

It's important to understand that the people in your content processes aren't just parts of the machine. In fact, I think another metaphor entirely works better for both them and the effectiveness of the process. Content processing staff members are like airplane pilots. They ought to have a sophisticated and powerful control panel in front of them to move the content payload from its source to its destination. They're the navigators and pilots who use the tools and methods that you provide rigorously but creatively to obtain the best result.

Tracking the master copy

It's exceedingly important in any content process to keep your eye on the master copy of content. The master is the one from which all others derive. For many types of content, the master copy changes over time, as follows:

- Before you receive the content, the master belongs to someone else and isn't your problem.
- As soon as you receive it, you must establish a master copy of each piece of content that's the one that you work on.
- As you process content, the master is the one that you've made changes to. At each stage of processing, the master may change to a new file or database record. Previous versions must lose their master status.
- As you create content components from the source, these components become the master copies, and the CMS handles their management.

The counter points to the master copy are the backup copies. Backups are old masters that you may need to return to. On the one hand, if you delete all backups, you run no chance of mistaking them for the master copies. On the other hand, the more backups that you keep, the easier it is to recover if you find that you've made a processing mistake and ruined the master. All processes must balance these two factors against each other and have neither too few backups for safety nor too many to interfere with a clear flow of control from master to master.

Here are some tips to keep your master files clear:

- **Tokens:** The clearest way to ensure that the file that you're working on is the master is to name it that way. If the root file name is 123.HTM, for example, you can name the master file MASTER_123.htm. This has the distinct advantage of being unambiguous. It has the disadvantage of making you name and rename files throughout your process. If you're using programs to access and move files anyway, this method may not be a lot of extra effort. At the very least, you can establish a regime for directory naming that clearly marks the directory where the master files reside.

- **Permissions:** If you're diligent, you can use the file attributes or directory permissions available in your operating system to manage master files. You can set the properties of backup files to read-only, or even inaccessible, so that they can't be mistaken for the master.
- **Directories:** You can do a lot to preserve master awareness by putting together a clear directory-naming policy. If your process has three main steps, for example, you can create directories called Preprocess, Step1, Step2, Step3, and Done. Content that's ready to be processed goes in the first directory. As each file completes a step, it's put in the directory for the next step. Thus any files in the Done directory can be assumed to be finished with the process and ready to move to whatever process comes next.
- **Workflow steps:** You can match directory naming with workflow steps to reinforce the distinction between copies of the same file. If your workflow steps are author, edit, and review, for example, you can create directories called BeingAuthored, ReadyForEdit, and ReadyForReview. Don't be afraid to be verbose and redundant in your naming. The key to keeping masters is to make sure that everyone always knows exactly what stage every file is in.
- **Physical objects:** In fast processes in which there's a lot of ownership change, passing a physical object from person to person can really help you keep track of who has the master. If you don't have the object, you shouldn't be touching the files. I've seen a printed and marked up version of a text file, for example, used as the marker for who has that file. If the printout is on your desk, you own the master of the corresponding file.

However you decide to track the master, do it consistently. Nothing ensures a faux pas more than an ever-changing set of rules around what the master content is today.

Managing the Processing Project

A content processing project begins with a set of assumptions about the amounts of content to be processed and its current state. From your logical design, you've determined the target state into which you want to bring the content (that is, what components and elements you expect it to yield). The gist of the project after logical design is to define and implement a process that reliably turns the current content into the structure and format you want.

The generalized steps of a content processing project are as follows:

- **Determine assumptions:** Determine your assumptions to create an initial scope, schedule, and budget for the project. Assuming that the project seems feasible, you can finalize your plan later based on a full specification.
- **Create an inventory:** Create a content inventory to show exactly what pieces of content you're expected to process.
- **Create a specification:** Create a content specification, a process plan, and a staffing plan that describes the process completely. The specification should also build the business case for the process specifying the expected gains that the process should yield.
- **Finalize the project plan:** From the specification, create a final project plan that maps the expected costs and risks of the project. Based on this plan, you can make the final decision whether to do the process as it's laid out or not.
- **Begin development:** Begin developing tools and methodology by focusing on the core algorithms and processes. Save the user interface for later.
- **Run a simulation:** If development has progressed sufficiently, run a small random sample of the source content through a working simulation of the processes and review the assumptions in the specification (especially those concerning the business case). Get sponsor signoff on the sample results and revisit estimates as needed.

- **Finish development:** Revise your approach as needed based on the results of your simulation and complete development.
- **Provide staff support:** Create processing manuals and training guides as needed to launch the process.
- **Staff and launch the process:** Clearly define the skill sets, characteristics, and aptitudes needed to staff the conversion project; then hire and train the appropriate staff.
- **Revisit and revise the tools and processes:** You must do this as needed to maintain or increase the quantity and quality of your output from the process.

The process that I'm suggesting emphasizes (almost to a fault) repeated feedback and checkbacks. The critical factor for the success of a content process is that it's viable economically. Thus you can't test and check back frequently enough. On the other hand, the process doesn't need to be a big deal or even overly formal. Just make sure that you're constantly checking your assumptions and that you remain sure that the process is netting you as much gain as possible for its cost.

Taking stock of the content inventory

The content inventory derives partly from the logical design of the acquisition system and partly from a simple cataloging of the files that you're expected to process.

A content inventory helps you create a solid agreement of what's to be processed and what it consists of. Especially if you're receiving a large bulk of content to be processed for the startup of the CMS, a content inventory can help you sort through and figure out how to dispose of the variety of content that you expect to receive. In addition, as you process content on an ongoing basis during the running of the CMS, you can request an inventory that serves as a packing slip, which goes along with each content delivery.

A content inventory is basically a file list. So it ought to list the file name of every file that you're expected to process. In addition, it ought to include the following:

- **File properties:** These should include date and size. The two attributes enable you to identify each file uniquely. If you receive a changed file with the same name later, you know it (or at least you could). Size also gives you a basic idea of the amount of content in each file.
- **Type:** Use the types that you defined per the instructions that I provide in the preceding section.
- **Owner:** This is the person to call if you have a problem with this file or some question about it.

The content inventory can save you. Suppliers have a tendency to dig up files at the last minute that they expect you to simply accept. Never mind that the new files are of a completely different type and require you to redevelop all your processes. If you insist on an inventory early in the process and agree to do development based on it, you can protect yourself from this unfortunate situation. (I've been there, and it's no fun at all.)

You may need to settle for less than a list of every file name that you expect to receive. It may not be possible for the source to come up with such a list. To my mind, the appropriate response to this is, "Give me what you have, and I'll plan based on that. But don't expect content that I don't know about to always get processed." It's a hard-core approach; but believe me, the alternative is worse.

Drafting the processing specification

After you've prepared a content inventory, you're ready to draft a processing specification. A processing specification ought to include the following:

- **A source specification:** This starts where the content inventory leaves off. In other words, for each file type identified in the inventory, the input specification can describe the structure and format of the input, including what character, paragraph, and section formatting is present, how the parts of the source are named, how its navigation is represented, and how any other metadata is represented in the source.
- **A target specification:** This includes the target format and structure, including what components the source gives rise to, what access structures its navigation feeds, and what sort of formatting is allowed to remain in its body elements.
- **A process specification:** This shows how the input is mapped to the output. The process specification should include a definition of all the tools and methods that are needed to do the transformation. It should also include expected task times and production rates.
- **A quality control specification:** This defines the level of quality you expect to reach and the methods that you expect to use to assess the quality of the output.
- **A staffing specification:** This defines the kinds and amounts of people that you need.
- **A business justification:** This shows that the process can do the tasks that it needs to within the budget and staffing constraints of the project. This part of the specification proves that there's adequate return on the investment that you make in the development of the process.

The processing specification should be created by the processing staff and signed off on by the project leaders (or sponsors if necessary). Before going into full production, you should run several random samples of content through the process and review the assumptions in the specification (especially those that have a direct effect on the business justification).

Defining testing methods

Especially in a large and ongoing content process, it's critical that you have some way of measuring the success of the process. Here are some considerations that you may want to use to help define your quality control processes:

- **Random sampling:** Find ways to sample randomly the content at any stage of its processing. As you define automated tools, build sampling into them. It's a simple matter, for example, to have a processing script log the name of every tenth file it processes. Make sure that, if you create methods such as this, you're not getting a biased sample. If files come to you in sets of ten, for example, every tenth file isn't a random sample. Every tenth file may be the first file of every batch that, in turn, may be different from the other files in the batch. All the rest of your quality control depends on reliable sampling, so make sure that you put adequate thought into it.
- **Diminish sampling:** Set your sampling rate to the level that's adequate for each individual staff member. Sample more from new staff members and automated processes and less as you gain confidence in the results. It's not a good idea to even go to no sampling for a person or automated process. Even if the person or process doesn't change, the input may.
- **Immediate feedback:** The longer that you wait to detect a mistake and inform the person who made it, the more times she's likely to make it. Especially with new people, try to review their work as they do it and provide feedback quickly so that they can learn quickly.
- **Immediate escalation of process issues:** Be prepared to promptly escalate and repair any systematic problems that you find with the tools or procedures that you create. A lot of bad output can be generated very quickly; or worse, you may need to stop an important process completely while you wait for a problem to be fixed.
- **Clear expectations:** Provide the staff with clear expectations of the quality and quantity of their work and communicate those expectations in many forms. In addition to stating them,

you should write them, send them in e-mail, and even include them in the user interface that the staff uses. The point here isn't to be oppressive but rather to make sure that the goals you have for quality and quantity are clearly understood and remembered by the staff.

- **Definitive samples:** Especially for new staff, provide strong examples of the kind of output that's correct. The stronger is the mental model that they have of the right way to handle a task, the better they can do at it. The exercise of preparing definitive samples helps the design team as well. It's amazing how many times creating an example forces you to reconsider the process being explained.

- **Cycle staff:** As much as possible, cycle staff through the quality-control positions. It's useful for each person to sit in the evaluator's chair for awhile to get perspective on her own work. Cycling staff between processing tasks also avoids boredom and drops in productivity. Do so with caution, however. Balance the need to keep people within their skill sets and the additional training required against the potential gains in staff enthusiasm and productivity.

Getting to the Core of Content Mechanics

The core of content mechanics is to discern the format and structure conventions in a set of content inputs (generally files or databases) and make good calls about what format and structure you can find and manipulate automatically and what must be dealt with by hand.

There's a level of structure below which you can't create transform programs. I'll never forget going to an English class with my friend Tim. It was a class in transformational grammar, and I was amazed. I had no idea that, behind language, there was so much structure. I got the same feeling while studying foreign languages. Every part of speech has a name and a correct place in the structure of a sentence. You can construct, deconstruct, analyze, and study language as you would any other system - more or less. As the grammar checkers available in modern word processors demonstrate, it's possible for a strictly logical computer to decompose and analyze language - but not very well. There's a logic and a mechanics behind language that you can tap into, but you still need a person (and a smart one) to give the final thumb's up to your text.

Note

Of course, these remarks apply much less to nontextual content. There are few generally accepted equivalents to transformational grammar in images, sound, or moving pictures.

In addition to the grammatical level of text, which you need editors to deal with effectively, there's a coarser grain of structure that you may call the format and explicit-structure level of text that has its own mechanics. These are the same two content attributes I've been discussing throughout this white paper. I add the word *explicit* to structure to contrast it in the present context with the grammatical structure of text that's more implicit. Grammatical structure depends entirely on word choice and punctuation. Explicit structure generally doesn't depend on word choice or punctuation, but rather on extra tagging or other markers that aren't the content, but are metadata.

Consider, for example, this sentence:

Danny is Isabel's son; Debbie and Naomi are her daughters.

You have no trouble getting the meaning out of this sentence because you've internalized the meaning-conveying rules of grammar. This kind of meaning, however, is implicit in the words that I choose and the punctuation that I use. No one has yet figured out (and they very well may never figure out) how to get a computer to reliably decode this kind of structure. In contrast, consider this alternate form of the same sentence:

```
<ISABEL>
   <DANNY type="son"/>
```

```
    <DEBBIE type="daughter"/>
    <NAOMI type="daughter"/>
</ISABEL>
```

This kind of structure is ideal for a computer to deal with. Nothing (or little) is left to interpretation. The computer may not know the true significance of sons and daughters, but it doesn't need to. Usually, it needs only to recognize tags and perform the rule associated with them.

Of course, content rarely comes to you like preceding the sample; and if it does, there's little need for a process to transform it. The most likely situation is that most of the grammatical structure is below the level of the structure that you need to recognize and transform. Few content management uses require you to deal with Isabel and her children. The structure that you need to deal with is much less granular. It's structures such as titles, headings, and management elements or such as creation date and author.

I say little more about the grammatical level of text and focus here on the format and explicit-structure level. Although the grammatical level of mechanics is common to all the texts in a particular language, the format and explicit structure apply to single publications, or, if you're lucky, sets of publications. You can usually expect, for example, that only one set of formatting and structure rules are applied to a single file containing, say, a description of a service offering. With little effort, you can determine what the rules are and transform the service-offering file into a service-offering component by stripping out the unnecessary surrounding information and mapping its parts to elements of the service-offering component class.

If you have a set of service-offering files, you're less likely to have the same level of consistency. Unless the creators were unusually careful in their creation and enforcement of standards, there's some level of variability among the files that requires you to write more sophisticated programs and bring in staff to make judgment calls on each file that varies too far from what standards you can discern.

The best content mechanic is someone who can use the tools available in the content's native environment (Visual Basic in Applications in Microsoft Word, for example), knows the structures of the CMS (an XML database, for example), and knows how to connect the two (by using Web programming with the DOM, for example).

Understanding the principles of mapping content

The best way to capture the mechanics of content is to go right to the core conversion issue. How do you find information of interest in source content and map it to components and elements in the CMS? Your ability to map hinges on the amount of correspondence that you can find between the format and structure of the source and that of the target. There are only a few ways that the source and target can correspond.

Both source and target can correspond as follows:

- **Directly:** There's exactly the same format or structural element in both.
- **Indirectly:** There's some way to infer the target format or structural element from clues in the source automatically.
- **Ambiguously:** There's more than one target element that the source element may give rise to.

If the source and target correspond poorly or not at all, the reason may be as follows:

- **No format or structure:** There's no format or structural element in the target components that corresponds to a source element. A person must add the additional elements to the target.

✍✍ **No markup:** Certain target elements exist in the source but are unmarked or inconsistently marked. Mapping depends on the judgment of a person.

Direct correspondence

In direct correspondence, the structure or formatting codes in the source have a direct translation in the target. The codes for bold, for example, are easy to recognize and convert to and from almost any form. Fortunately, but not surprisingly, most of the common format conventions have direct correspondence in all systems. Here, for example, is a sentence that has a variety of character formatting:

> *The **rain** in Spain falls mainly on the plain.*

And here it is using the arcane Word RTF markup language:

```
The {\b rain} in {\ul Spain} falls {\i mainly} on the {\strike
plain}. \par
```

Here's the same sentence in HTML:

```
The <B>rain</B> in <U>Spain</U> falls <I>mainly</I> on the
<STRIKE>plain</STRIKE>.<P>
```

You can see how simple it would be to translate from the first to the second markup. In fact, most of the tag names are exactly the same. Notice that more than one tag in the source can map to the same tag in the target markup. You may choose to map italic text in the source, for example, to bold in the target because italic doesn't render well on the computer screen.

The conversion becomes even simpler if the source and target markup language share the same syntax as do HTML and XML. Consider this HTML source:

```
The <B>rain</B> in <U>Spain</U> falls <I>mainly</I> on the
<STRIKE>plain</STRIKE>.<P>
```

And this in XML:

```
The <B>rain</B> in <U>Spain</U> falls <I>mainly</I> on the
<STRIKE>plain</STRIKE>.<P>
```

If the creator of the XML tag set chooses the same tag names as those used in HTML (and for stuff such as bold and italic, why wouldn't she?), no conversion is needed at all. In fact, with the exception of the rule in XML that all open tags need to match with close tags - which HTML lacks - there's often very little to do to translate HTML into XML. Moreover, there's now a standard called *XHTML*, which adds close tags and a few other minor conventions to HTML to make it completely compatible with XML.

Indirect correspondence

In an indirect correspondence, the same format or structure exists in the source and target, but it takes more than just a simple substitution to map from one to the other. Consider, for example, the following HTML outline:

```
<H1>My Life</H1>

   <H2>Birth to Age 5</H2>

      <H3>Life as a Baby</H3>

      <H3>Kindergarten</H3>

   <H2>Age 6 and Up</H2>

      <H3>My Best Friend</H3>
```

```
      <H3>My Worst Enemy</H3>
   <H2>My Future</H2>
```

Now consider the same outline rendered in XML, as follows:

```
<SECTION>My Life
   <SECTION>Birth to Age 5
      <SECTION>Life as a Baby</SECTION>
      <SECTION>Kindergarten</SECTION>
   </SECTION>
   <Section>Age 6 and Up
      <Section>My Best Friend</Section>
      <Section>My Worst Enemy</Section>
   </SECTION>
   <SECTION>My Future</SECTION>
</SECTION>
```

Headings are represented in both the HTML and the XML code. In HTML, however, there's a different tag for each heading level (H1, H2, H3, and so on.). In XML, the same tag (<SECTION>) is used over and over, and the heading level is determined by how nested a particular <SECTION> tag is. The heading Kindergarten, for example, is nested within two other sections (Birth to Age 5 and My Life) so, therefore, must be a level-3 heading. Although not terribly difficult, this isn't a direct translation.

In an indirect translation, there's a programmatic way to convert from one markup language to the other, but the programming isn't a simple substitution of one tag name for another. That is, you can get a computer to do the transformation for you, but it may require some sophisticated programming.

Ambiguous correspondence

Sometimes a single tag in the source could be one of many in the target. Consider the following HTML source:

```
<P>Sir Isaac Newton said:
<PRE>"If I have accomplished something great, it is because I am
standing on the shoulders of giants"</PRE>
<P>Our programmers have said:
<PRE>Function Accomplishment(Whose)</PRE>
<PRE>myAccomplishment = Accomplishment(myMentor)</PRE>
<PRE>End Function</PRE>
```

I now want to convert this HTML into the following XML target:

```
<P>Sir Isaac Newton said:</P>
<QUOTE>"If I have accomplished something great, it is because I am
standing on the shoulders of giants"</QUOTE>
<P>Our programmers have said:</P>
```

```
<CODE>Function Accomplishment(Whose)</CODE>
<CODE>myAccomplishment = Accomplishment(myMentor)</CODE>
</CODE>End Function</CODE>
```

In the source, the <PRE> tag (an HTML tag that enables you to retain the formatting of the original text, including white space) was used for both a quotation and for programming code. In the target, I want to differentiate the two. Unfortunately, no computer that I know of can tell the difference between the two. You need a person to help.

Ambiguities of this sort happen most frequently if format tags are used to represent structural elements in the source. Suppose, for example, that I use an Arial 8-point italic font to represent both a level-5 heading and a note. How can a computer tell one from the other? Generally, one looks for other clues to make the best assumption possible. Maybe the notes always begin with the word *Note*. As one who's been fooled by these sorts of assumptions many times, I recommend being very careful. There's no limit to the variation and creativity of an author who's worked within a nonconstrained authoring environment.

By the way, you may also notice in the preceding example that the HTML sample has open paragraph tags (<P>), while the XML has open and close paragraph and line-break tags (<P></P> and
</BR>). For HTML tags that aren't required to be closed (<P>,
, <IMG>, and the like), you must map indirectly to XML.

No correspondence

If you're most fortunate, there's a direct correspondence between the format and structural tagging of your source and target markup languages. If you're least fortunate, there's no correspondence at all. Markup languages vary in their richness and range of coverage. All represent simple constructs, such as bold and italic. Few represent complex constructs such as columnar layout and advanced table formatting. HTML, which began quite impoverished, has improved considerably with the advent of Cascading Style Sheets (CSS), which enable precision layout as well as arbitrary character and paragraph formatting.

But how do you represent a part number in HTML? There's no <PartNumber> tag in HTML, so how can the fact that QS213 is a part number be maintained if you convert to HTML? The answer is that it can't be represented (except through various forms of trickery.) Furthermore, there may be no way to decide that a particular number in the source is a part number or something else.

Noncorrespondence between source and target often takes the following forms:

- **Rich formatting to impoverished formatting:** Exactly converting a richly formatted table in Microsoft Excel, for example, to HTML simply can't work. There's a level of subtlety and formatting richness that you can create in Excel that can't be duplicated in HTML, because HTML doesn't have the corresponding markup to support it. The only solution here is to "dumb down" the source markup until it can be represented in the target system.
- **Structural markup to formatting markup:** As in the example of the part number, if your source environment enables you to create and use arbitrary names but the target environment doesn't, you have a problem. Microsoft Word documents that you come across, for example, may include dozens of paragraph styles that authors and editors use to indicate the structure of the document. If you convert a file so formatted to HTML without CSS, you of necessity lose all these structural names. HTML with CSS does provide a way to preserve structural names but no particularly apt way to work with them.
- **Interactive markup to noninteractive markup:** Much of what appears in HTML pages, for example, occurs not because of the markup on the page but because of the programming script behind the markup. An expanding and collapsing table of contents, for example, is

often created by JavaScript code. If you convert such a table of contents from HTML to XML, you lose the expand/collapse functionality.

No or poor markup

If some element that you expect to be present and tagged in the target is present but unmarked in the source, you have a problem. Consider the following HTML source:

```
<P>My Best Friend
<P>At age 6, I met Timothy, a blond, square
jawed 3rd-generation member of the Fighting Irish.
```

Suppose that you want to convert this HTML to the following XML:

```
<TITLE>My Best Friend</TITLE>
<BODY>
At age 6, I met Timothy, a blond, square
jawed, 3rd generation member of the Fighting Irish.
</BODY>
```

I'm sure that you can distinguish that the first line of the HTML is the title, and the second line is the body, but an unaided computer-based conversion program couldn't (at least, not reliably). In this example, you want markup in the target that doesn't exist in the source. The way that the title element is tagged in the source simply doesn't provide enough information to distinguish it as a title at all. You must provide a measure of human judgment to ensure a clean conversion.

Most wouldn't bother trying to convert the title automatically the way that it's represented in the preceding example. But how about the following representation?

```
<P><B>My Best Friend</B>
```

Now, you can detect some logic in the source markup. You can say, "If it's in its own paragraph and is bold and doesn't end with a period, it must be a title." Maybe so, or maybe not. Do you know for sure that these characteristics mark a title uniquely and unambiguously? Without some extensive sleuthing through a large sample of what you want to convert, I'd say that you're on thin ice. The element that you want to isolate is poorly marked in the source, and you must include some measure of human judgment to ensure a clean conversion.

It's worth noting here that the capability of advanced programs to discern subtleties of language and make reasonable assumption is on the rise. Each new generation of conversion products gets better at its capability to make something in the target out of nearly nothing in the source.

A good example of these increasing capabilities is name recognition. Although you and I can recognize a person's name in a block of text quite easily, a computer can't. Simple rules such as capitalization simply aren't enough. Rather, you use much more subtle clues such as whether you've ever seen a name like that or whether it make sense that a name goes at this point in the sentence. Until fairly recently, computers were incapable of tagging names reliably. Today, however, they can. Recognition programs can discern clues intriguingly close to those that educated people use.

No existence

Of course, there's one final situation that can't be handled by a conversion system at all. If your target needs to have an element in it that's simply not present in the source, no amount of clever conversion scripting can create it for you. Rather, you need to create the element as part of a wider aggregation process.

Converting content from Microsoft Word to XML

As a short, practical guide to some of the issues of conversion, I provide an example and analysis of a program that converts an Microsoft Word file to a particular XML DTD. The program happens to be the one that I created when, early on in the creation of this white paper, I decided to move out of my Word-authoring environment and into an XML environment to begin to create more of a system. It converts files with a particular Microsoft Word format into XML that obeys my CM Domain DTD. The program is written in Microsoft Visual Basic for Applications (VBA) which is the programming language built into Microsoft Word.

The CM Domain system that I created to manage the content of this white paper and its peer publications is not an enterprise CMS. It is a loose affiliation of tools and programming code that let's me keep my creation, administration, and publishing organized and efficient. While my CM Domain system is much smaller than the kind of system you are likely to need to implement, it is just the right size to illustrate some of the key concepts of collection, management, and publishing. I draw examples from my CM Domain system in this section as well as in the chapters on building collection, management, and publishing systems.

Note

My intent in presenting this example is to talk not only to would-be content process programmers, but also to a more general audience who wants to understand the process of conversion more than the mechanics. If you're a nonprogrammer, you may not understand the code that I present; but you should understand most of the commentary that I provide.

Note

This code also combines the Basic programming language with the Word programming objects. The result is a program that looks half-Basic (If, Then, and so on) and half-Word (AutoFormat, Replacement.Text, and so on). All the documentation for both halves of the language are very possibly on your nearest hard drive. Try choosing the Tools » Macro » Visual Basic Editor command in Microsoft Word. Then choose Help from the Visual Basic Editor that appears.

At the highest level, the code that I describe looks as follows:

```
Sub WordToXml()
   CharsToXML
   ParasToXML
End Sub
```

I break the Word-to-XML program into two parts. The first program adjusts the character formatting in the Word file so that it's compatible with XML. The second program parses through the Word file and tries to map format and structural elements from Word to the appropriate XML forms.

Special characters to XML

The CharsToXML program, as shown in Listing 30-1, is almost entirely Word object code. In fact, I recorded most of the program by using Word's Macro Recorder function.

Note

For nonprogrammers, it may also help to know that the lines preceded by an apostrophe are comments that I've added that describe what the code that follows does.

```
Sub CharsToXML()
   'Turn off Word's AutoFormat feature
```

```
With Options
   .AutoFormatAsYouTypeReplaceQuotes = False
   .AutoFormatAsYouTypeReplaceSymbols = False
   .AutoFormatAsYouTypeReplaceOrdinals = False
   .AutoFormatAsYouTypeReplaceFractions = False
   .AutoFormatAsYouTypeReplacePlainTextEmphasis = False
   .AutoFormatAsYouTypeReplaceHyperlinks = False
End With
'Replace single quotes
Selection.Find.ClearFormatting
Selection.Find.Replacement.ClearFormatting
With Selection.Find
   .Text = "'"
   .Replacement.Text = "'"
   .Forward = True
   .Wrap = wdFindContinue
   .Format = False
   .MatchCase = False
   .MatchWholeWord = False
   .MatchWildcards = False
   .MatchSoundsLike = False
   .MatchAllWordForms = False
End With
'Replace double quotes
Selection.Find.Execute Replace:=wdReplaceAll
With Selection.Find
   .Text = """"
   .Replacement.Text = """"
End With
Selection.Find.Execute Replace:=wdReplaceAll
'Replace em dash
Selection.Find.Execute Replace:=wdReplaceAll
With Selection.Find
   .Text = "^+"
   .Replacement.Text = "-"
End With
```

```
        Selection.Find.Execute Replace:=wdReplaceAll
        'Replace en dash
        Selection.Find.Execute Replace:=wdReplaceAll
        With Selection.Find
           .Text = "^="
           .Replacement.Text = "-"
        End With
        Selection.Find.Execute Replace:=wdReplaceAll
    End Sub
```

At the highest level, what this program does is turn off a lot of Word's automatic formatting options and change "special" characters (such as curly quotes and em dashes) into a form more easily stored in an XML file and translated into other formats, such as HTML. Of course, as I later recreate Word files, I must translate these symbols back; but for the CMS repository, I want them in as general a form as possible.

The particular code segments work as follows:

- **Turn off Word's AutoFormat feature:** This disables the feature in Word that automatically applies formatting codes to text. I turn it off so that it doesn't keep applying formats that aren't universally recognized in my target publishing formats.
- **Replace single quotes:** This performs a Find and Replace function on all single-quote special characters and replaces them with generic single quotes. Notice the peculiarity of Word in that, in recording a macro such as this, you put the same character in both the find and replace parameters. There are myriad quirky things such as this in every environment that you just must find and learn (or as I do, find, forget, and find again!).
- **Replace double quotes:** It works the same way that the single-quote code does. Luckily, Word VBA doesn't make you retype all the parameters of the replace operation each time that you call it.
- **Replace em dash:** The Word representation of an em dash (-) is ^+. Word can't deal with special characters in their text boxes so it, too, needs to represent them as special text codes.
- **Replace en dash:** Notice that, for simplicity, I get rid of en dashes by translating them into the same characters as the em dashes.

You can create something very close to the CharsToXML program in about two minutes by using Word. Simply choose Tools » Macro » Record New Macro and perform the actions that I outline by using the commands on Word's various menus. You're likely to transcend Word's macro recorder, but it's a great way to learn and quickly generate VBA code.

Paragraphs to XML

The ParasToXML program, as shown in Listing 16-2, has quite a bit more logic and fewer Word object calls. It runs through the file one paragraph at a time and tries to map Word styles to some component or access structure in the repository. It's fairly simple in its approach, but still accomplishes a lot of work.

```
    Sub ParasToXML()
        Dim sXml, sFront, sEnd, sBadStyles, sFilePath As String
        Dim sStyle, sText As String
```

```
   Dim nParaCount, IDnID, nCurrPara, nNumBull, bConceptOpen As
Integer
   'Get the paragraph count for the document
   nParaCount = ActiveDocument.Paragraphs.Count
   'Set the XML front and end matter
   sFront = "<?xml version=""1.0""?>" & Chr$(13) & "<!DOCTYPE
Cmdomain PUBLIC ""Cmdomain"" ""cmdomain.dtd"">" & Chr$(13) &
"<CMDOMAIN>"
   sEnd = Chr$(13) & Chr$(13) & "</CMDOMAIN>"
   'Get the section and concept ID to start at
   IDnID = InputBox("start IDs at")
   'Step through the paragraphs
   For nCurrPara = 1 To nParaCount
      sStyle = LCase(ActiveDocument.Paragraphs(nCurrPara).Style)
      sText = Selection.Text
      nPrevLevel = DoPara(sStyle, sText, IDnID, sBadStyles, sXml,
nNumBull, nPrevLevel, bConceptOpen)
      Selection.MoveDown Unit:=wdParagraph, Count:=1
      Selection.MoveDown Unit:=wdParagraph, Count:=1,
Extend:=wdExtend
   Next nCurrPara
   'Add the final closing tags
   sInsertText = ""
   If bConceptOpen Then
      sInsertText = "</CONCEPT>" & Chr$(13) & sInsertText
   End If
   For m = 1 To nPrevLevel
      sInsertText = "</SECTION>" & Chr$(13) & sInsertText
   Next m
   'Write the XML to a file
   sFilePath = "C:\CMDomain\"
   sFilePath = sFilePath & "ConvertedWordFile" & nID & ".xml"
   Open sFilePath For Output As #1
   Print #1, sFront & sXml & sInsertText & sEnd
   Close #1
   'Alert user to unhandled styles
   MsgBox "Here are the Uncaptured style(s) " & sBadStyles
End Sub
```

Simply explained, the program steps through each paragraph of the file and passes it to a function called DoPara. DoPara decides what kind of XML to produce from each paragraph. After all the paragraphs are processed, ParasToXML writes the resulting XML to a disk file. The result is an XML file that's a translated version of the source file. It's up to a process that comes later to merge the XML file into the larger repository.

The various code blocks perform the following functions:

- **Get the paragraph count for the file:** This line retrieves the number of paragraphs in the file from the appropriate Word object.
- **Set the XML front and end matter:** The sFront variable holds the open XML tag and the XML <Doctype> tag that are needed for the file to be validated against the target DTD (cmdomain.dtd).
- **Get the section and concept ID to start at:** To avoid duplication between the tag IDs that this program assigns and the ones that are already in the repository, the program prompts the user for a start ID. Without too much extra effort, I could have queried the repository for the next available ID. This simpler way requires the user to know an ID range that's not yet been used.
- **Step through the paragraphs:** These lines simply step the word cursor forward one paragraph at a time through the file. At each paragraph, the program collects the paragraph's style and its text and passes it along with the current ID, a list of bad styles, and the XML so far created to the function DoParas. DoParas adds to the XML and possibly to the list of bad styles and returns the number of levels deep that the last section tag is so that it can be closed correctly after the last paragraph is processed. *Bad styles* are those that appear in the Word file but aren't accounted for by the conversion program. I probably should have used Word's newer capability to collect paragraph information without actually changing the position of the cursor. My way works just fine, even if it may be slower than the newer method.
- **Add the final closing tags:** This section of code closes off the last <CONCEPT> tag and all the <SECTION> tags that need to be closed after the last paragraph is processed. This is an example of indirect correspondence. There's nothing directly in the Word file that indicates that the <SECTION> tags ought to be closed. But you can infer this fact correctly (including how many closing tags to insert) based on the nesting level (nPrevLevel) of the last <SECTION> opening tag created. This lack of symmetry between Word and XML is a pain, but not a terminal one.
- **Write the XML to a file:** After all the paragraphs of the source file are processed, the program opens a text file in a particular directory on the local hard drive (sFilePath) and writes the XML code into it. The directory in this case is hard-coded into the program. A slicker program would retrieve it from somewhere or even embed this program in a user interface for selecting the path that you want. But that's all frill that you can add later. The file name is derived from the last ID that was assigned to the tags in the source. If the ID supplied at the beginning of the program is unique, the file name is unique too.
- **Alert user to unhanded styles:** If a paragraph styles finds no match in DoParas, it's added to the bad styles variable (sBadStyles), which is displayed at the end of the program. This is a fail safe to ensure that the user knows that there are styles in the source file that couldn't be accounted for.

Although interesting enough to look at, this program is no more than a shell for the real workhorse, DoParas(), which I show you next.

Structure to XML

The DoParas() function does the real work of the conversion system. It implements the processing specification for this source and target. To understand it, you first need to understand the source and target specifications. To cut right to the chase, I combine the source and target specifications into the following list of format elements. These are all Word styles that need to be recognized in the source and mapped to the target formats, as follows:

- **Headings:** All heading styles (Heading 1 through Heading *n*) need to be found and mapped to <SECTION> tags. The level of the heading isn't represented by a number (as it is in the Word styles) but rather by the number of <SECTION> tags within which it's embedded. The <SECTION> tags that are generated by this code represent one branch of the overall repository hierarchy. When, later, this file is added to the CMS repository, it's nested within whichever <SECTION> tags are already there. Thus the heading level within this file isn't the heading level that this content has after it's inside the larger repository. The <SECTION> structures aren't components, but rather the hierarchy within which the components in this CMS are listed. In the language of correspondence, you'd say that heading tags in the source correspond indirectly to <SECTION> tags in the target.
- **Summary:** If sections aren't components in this CMS, what are? In this example, there's only one component class that's created. It's called a *Concept class*. In the Word file, Concept components are marked by a paragraph with the style Summary. The beginning of a Concept component is marked by a Summary paragraph. The end of the Concept component is marked by either the next Summary paragraph, the next Heading-styled paragraph, or by the end of the file. As you may expect, the rules I've laid out dictate that Concept components can't span sections of the hierarchy. In the language of correspondence, you'd say that Summary tags in the source correspond indirectly to <CONCEPT> tags in the target.
- **Normal and Body Text Indent:** If a paragraph has either of these styles (Normal or Body Text Indent), it's simply converted to an XML <P> element. In the language of correspondence, you can say that Normal and Body Text Indent tags in the source have a direct correspondence to <P> tags in the target.
- **Bull and Bullet:** If a paragraph has either of these styles, it must be mapped to an <LI> element within a <UL> element. The transition is tricky, because the first bull or bullet style in a sequence should be preceded by a <UL> tag. The last one should be followed by a </UL> tag and the ones in the middle should not be preceded or followed by any tags. In the language of correspondence, you'd say that Bull and Bullet tags in the source have an indirect correspondence to <UL> and <LI> tags in the target.
- **Pre-styled paragraphs:** These are mapped directly to <PRE> elements in the target components. You may have noticed that the XML elements I chose look strangely like HTML. That's no accident, of course. I decided to use HTML tag names for readability and simplicity. This doesn't mean that I'm limited to HTML in the publication output. Because the tags are within an XML structure, my publication templates can translate them effectively into any other form (almost) that I may want. Of course, translating them to an HTML format is easiest.
- **Intro-styled paragraphs:** These are mapped directly to <INTRO> elements in Concept components. <INTRO> is an XML, not an HTML, tag.

So, in this system, the hierarchy-access structure of the content is represented by the headings; all other content is mapped into a set of Concept components. The only allowed content within a heading is a Concept (marked by a paragraph with the Summary style). All other paragraphs give rise to particular elements within a Concept component. A Concept component can have any number of <P>, <UL>, <PRE>, and <INTRO> elements arranged in any order.

Given these rules, a DoParas() function such as the one shown Listing 16-3 can do the necessary conversion.

```
Function DoPara(sStyle, sText, nID, sBadStyles, sXml, nNumBull,
nPrevLevel, bConceptOpen)
   'Add close tag to bullet list after last one
   If sStyle <> "Bull" Or sStyle <> "Bullet" Then
      If nNumBull <> 0 Then
         sXml = sXml & Chr(13) & "</UL>"
         nNumBull = 0
      End If
   End If
   'Prepare for heading styles
   sPrefix = LCase(Left(sStyle, 7))
   If sPrefix = "Heading" Then
      nCurrLevel = Trim(Mid(sStyle, 8))
      sStyle = "Heading"
   End If
   'Exclude Word paragraph characters
   If Right(sText, 1) = vbCr And sText <> vbCr Then
      sText = Left(Selection.Text, Len(Selection.Text) - 1)
   End If
   Select Case sStyle
      'Process Section syles
      Case "heading"
         If bConceptOpen Then
            sXml = sXml & Chr$(13) & "</CONCEPT>"
            bConceptOpen = False
         End If
         sInsertText = ""
         nlevels = nCurrLevel - nPrevLevel
         Select Case nlevels
            Case 0
               'We are at the same level, close prev section
               sInsertText = Chr$(13) & "</SECTION>"
            Case Is > 0
               'We are at a lower level so no close's
            Case Is < 0
```

```
                'We are at a higher level so no close's
                For m = nCurrLevel To nPrevLevel
                   sInsertText = Chr$(13) & "</SECTION>" &
sInsertText
                Next m
             Case Else
          End Select
          sXml = sXml & sInsertText & Chr$(13) & "<SECTION ID =""S"
& nID & """><TITLE>" & sText & "</TITLE>"
          nPrevLevel = nCurrLevel
          nID = nID + 1
       'Process Concept component syles
       Case "summary"
          If bConceptOpen Then
             sXml = sXml & Chr$(13) & "</CONCEPT>"
             bConceptOpen = False
          End If
          sXml = sXml & Chr$(13) & "<CONCEPT ID =""C" & nID & """
><SUMMARY>" & sText & "</SUMMARY>"
          nID = nID + 1
          bConceptOpen = True
       'Process p element syles
       Case "normal"
          sXml = sXml & Chr$(13) & "<P>" & sText & "</P>"
       'Process ul element syles
       Case "bull", "bullet"
          Select Case nNumBull
             Case 0
                sXml = sXml & Chr$(13) & "<UL><LI>" & sText &
"</li>"
             Case Else
                sXml = sXml & Chr$(13) & "<LI>" & sText & "</LI>"
          End Select
          nNumBull = nNumBull + 1
       'Process Pre and Intro element styles
       Case "Pre"
          sXml = sXml & Chr$(13) & "<PRE>" & sText & "</PRE>"
       Case "intro"
```

```
            sXml = sXml & Chr$(13) & "<INTRO><P>" & sText &
"</P></INTRO>"
         Case Else
            If InStr(sBadStyles, sStyle) = 0 Then
               sBadStyles = sBadStyles & Chr$(13) & sStyle
            End If
      End Select
      DoPara = nPrevLevel
   End Function
```

The code chunks in this sample perform the following tasks:

- **Add close tag to bullet list after last one.** This code block detects whether the previous paragraph was the last in a sequence of <UL> element styles (bull or bullet Word styles). If it was the last, it inserts the closing tag </UL> into the XML output.
- **Prepare for heading styles.** If the current style is a heading variety, this code sets the variable nCurrLevel to its heading level (as indicated by the last letter of its style name) and resets the sStyle variable so that, regardless of the level of the heading, it's trapped later under the style name Heading.
- **Exclude Word paragraph characters.** Word paragraph markers (which VB has a constant for called vbCr) need to be excluded from the text (sText) that's turned into XML.
- **Process section styles.** This block of code produces <SECTION> tags in the XML output. The target element has an ID attribute and a <TITLE> subelement. The whole element looks as follows: <SECTION ID="S123"><TITLE>Section Title</TITLE></SECTION>. The program creates the ID by prepending an S to the current value of the nID variable (after which the program increments an ID so that it remains unique). The <TITLE> element is filled from the sText, which is the entire text of the Heading-styled paragraph. If there's a <CONCEPT> element open, a corresponding close tag is added. In addition, if the current heading's at a level higher than the previous heading (whose level is stored in nPrevLevel), the program closes the right number of previous <SECTION> tags.
- **Process concept component styles.** If the program encounters a style called Summary, it creates a new Concept component in the output XML. If a <CONCEPT> tag is already open (as indicated by the flag bConceptOpen), the code closes the old <CONCEPT> tag before opening a new one. The Concept component lives within a <CONCEPT> element that has an ID attribute and a <SUMMARY> subelement. The program creates the ID by prepending a C to the current value of the nID variable (after which the program increments nID so that it remains unique). The <SUMMARY> element is filled from sText, which is the entire text of the Summary-styled paragraph.
- **Process P element styles.** If the program encounters a style called Normal (Word's default style), it simply transforms it into a <P> XML element.
- **Process UL element styles.** The styles Bull or Bullet are transformed into <LI> elements. If this is the first <LI> encountered, the program puts in the opening <OL> tag.
- **Process Pre and Intro element styles.** The Pre and Intro styles in Word translate straight across to XML <PRE> and <INTRO> elements.

The price of correspondence

The tags with direct correspondence in Listing 16-3 represent no trouble. As long as you can find the target structure, a simple line of code is all that you need to do the transformation. On the other hand, notice how much of the code in the example that I provide is dedicated to indirect correspondence. The fact that XML requires end tags while Word doesn't is the source of the majority of the complexity in this script. You can generally infer where a tag ought to end, but you need to do a lot of accounting to keep track. In this simple example, the inferences cause just enough complication to make my point. It wouldn't take much to complicate the Word file to the point where the inferences become a major cause of complexity.

Much of the complexity in conversion programs comes from these sorts of structural mismatches. Thus you'd do well to study the structural differences between your source and target markup language before beginning to code. In an indirect correspondence, complexity doesn't come from not knowing what source structure to map to what target structure. Complexity comes from the amount of logic that it requires to do the mapping that you know you need to do.

Poor or ambiguous markup causes even more complexity than indirect correspondence. For the sake of brevity, I didn't illustrate them in this example, but they're common. Poor markup means that the structure you're seeking in the source is marked inconsistently. Depending on exactly how poorly marked the structure is, you can generate a lot of code by trying to find all the possible representations of a poorly marked source element. In a case of ambiguity, no amount of code can tell which target structure to create. In ambiguous markup, you need a human in the process to say which target tag an ambiguous source tag should be mapped to. Of course, not all cases are purely ambiguous, so you can spend a lot time finding ingenious ways to figure out what target tag to create from the subtle clues that may surround a source tag.

From example to reality

The example that I provide is working code (only slightly simplified) that I've used to transform source Word files into XML. I chose it because it was simple enough to explain quickly. Here are just a few of the reasons why the code that you write may be somewhat more complex:

- **Management elements:** Except for the ID attributes, there are no management elements located in the source or created in the target components. I really only found and translated the body elements. In general, you need more code to find and translate or infer the values of management elements (such as Create Date, Author, and so on) in the target components.
- **Multiple components:** Notice that the files contain only one type of component. Although this isn't an uncommon situation, you often have files in which you must find a variety of components in the source and turn them into target components.
- **Formatting:** The only formatting that I cover in the example is special characters. In fact there's an ocean of formatting that may occur, especially in a program such as Word that complicates your code considerably. Among the worst formatting to deal with is table formatting, in which you have so many different intertwined options that you're likely to give up before you arrive at a complete conversion routine that covers all possible tables.
- **Styles:** My example covers only a few styles in the source files. Most Word files have many, many styles that you'd need to sort out and eliminate. In addition, my program assumes that all structure in the source files is represented by Word paragraph styles (rather than people simulating styles by applying character formatting to the Normal style), a dubious assumption in most word processing files that you get to convert.
- **Validity:** To be sure, the biggest simplification that I made in the example is that the Word file is valid with respect to its own style sheet. If I unleashed this program against a file that wasn't already well structured in the way that I assumed, it would likely fail. The only

validation that the program provides is to alert me to any styles that were unhandled. This is only one of many ways that the file could fail to be in the form that I expect.

Summary

Content processing is the term that I think best summarizes the work that must be done to the information that you collect to ready it for the CMS. In content processing, the following occurs:

- You mostly convert content, but in doing so, you get involved also in acquisition and aggregation.
- You find the places in your system where the up-front cost of automation can be recovered clearly by gains in productivity or quality.
- You develop a content inventory and processing specification to work through the details of the transformation from source to target.
- You look for the correspondence between the structures of the source and those of the target components that you're trying to create automatically. May all your correspondences be direct.