MUJ
MACROMEDIA USER JOURNAL™
http://www.muj.com
July 1998
Number 72

Building Adaptable Templates for Large Projects

Phillip Kerman

How do you make a template that saves the author's time while remaining fresh and unique-looking to the user? If you use templates, every screen in your project can look unique—without having to program each screen individually. Templates can range from completely automatic (where the author might not even need Director or Authorware to make changes) to something resembling a "style guide," where it only helps the project look consistent. This article will discuss the steps to designing a template that fits your project.

THE dream: Imagine you've finished the first section of your project. The client delivers content for the remaining five sections. You scan the toolbar in Director for the "Finish Project" button … one click, and your project is done! Don't start looking for that button, because it doesn't exist. By designing a set of templates for your project, however, you can automate the process significantly.

The real world: The scenario I experienced during a recent project was a little different—but it was similar in that all the programming was finished months before the content. Through a process of designing and building templates, we ended up with a code file that allowed the content developers to finish the project. They simply followed a few rules, named cast files and castmembers appropriately (see **Figure 1**), changed a few numbers in an "ini" file (see **Figure 2**), and they could finish the project without any help from me!

The conventions for this project were easily communicated—but they did require that the content developers have some Director knowledge. Why didn't we make templates that were so automatic that anyone could modify them? We could have. This would have had a greater value, but also a greater cost. This balance is only one of the many considerations when designing a template.

Model or template?

Though it's just a matter of semantics, I'd like to define these two terms. *Models* are the way the *user* experiences content. *Templates* are the way the *author* implements content. A model is such a good example of an idea that the same model can be used every time you need to communicate that idea; for instance, a "return on investment" model—spend money now, get the payback over time. Models in multimedia can be a simple "slider" (where anyone knows what to do) or a "multiple-choice" model. We should capitalize on good models—use the ones that work. This article is about *templates* and how they can be designed to save the author's time. Even though the concentration will be on the author, it's important not to forget the user when developing templates.

Types of templates

Templates can be categorized on two scales:

- How dynamic or hard-wired is it?
- How much technical knowledge is required to implement the content?

Figure 3 plots these two considerations and shows there are four basic combinations. Before we look at each type, notice that the higher or further to the right, the greater the effort and cost—but the value also increases. Also notice that templates at the bottom extreme are better described as "style guides," and the top extreme are "engines." The templates at the left are "helpers," whereas to the right, you begin to actually recreate software "applications."

It should be noted that templates don't always fall neatly into these four categories, but for the purpose of discussion, we can break them down into the following four types:

1. Style guide for author
2. Engine for author
3. Runtime style guide
4. Runtime engine

Type 1: Style guide for author

The least sophisticated type of template is both hard-wired and requires the author to make modifications. It's really nothing more than a style guide that an author copies and pastes, then customizes to the content. I consider this a template because it ensures a consistent look to your project.

This type of template can take any form with which authors are comfortable. It usually involves a "master" starter file—maybe locked on a server, or saved as an *Authorware model* (don't get confused by the word "model"; these are "templates" according to my definition).

Since it's duplicated in every instance, the disadvantages *are* significant.

- **It's not easy to make global changes late in production.** Each instance is hard-wired and would therefore require re-modification.
- **File size is large.** When an Authorware file has lots of *icons* or a Director file has lots of *members* and *frames*, performance diminishes—not to mention the increased chance of corruption and lack of modularity.

Although these disadvantages are significant, it should be mentioned that one advantage of such a

Figure 1. (Above) Production people simply followed a naming convention for the castmembers.

Figure 2. (Right) This text "ini file" contained all the details for each page in the project—and could be edited from outside the source Director file.

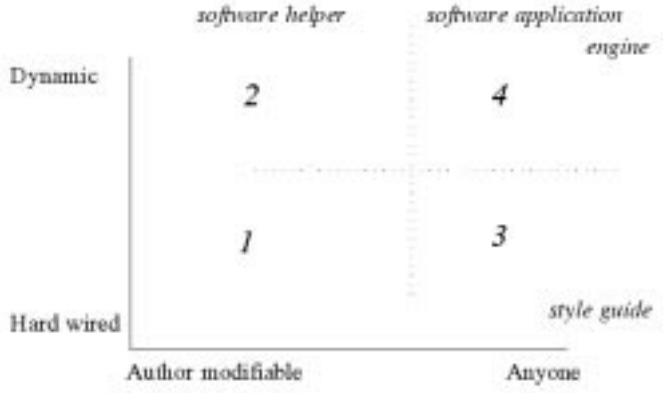

Figure 3. The four types of templates are categorized by two scales—how dynamic the template is and who modifies the template.

Technical Editors Ken Durso and Darrel Plant;
Founding Editors Tony Bovè and Cheryl Rhodes;
Publisher Robert Williford;
Vice President/General Manager Connie Austin;
Managing Editor Heidi Frost; **Copy Editor** Farion Grove

Direct all editorial, advertising, or subscription-related questions to Pinnacle Publishing, Inc.:
1-800-788-1900 or 770-565-1763
Fax: 770-565-8232
Pinnacle Publishing, Inc.
PO Box 72255
Marietta, GA 30007-2255
E-mail: **muj@pinpub.com**
Pinnacle Web Site: **http://www.pinpub.com**
Macromedia technical support: 415-252-9080

Macromedia User Journal (ISSN 1065-3929) is published monthly (12 times per year) by Pinnacle Publishing, Inc., 1503 Johnson Ferry Road, Suite 100, Marietta, GA 30062.

POSTMASTER: Send address changes to *Macromedia User Journal*, PO Box 72255, Marietta, GA 30007-2255.

template is that it's better than nothing. Also, it requires very little upfront investment—so this would be appropriate for a small job. Finally, the same look and feel can be imposed throughout a job (even if several authors are working on the project).

Tips for Authorware

- **Use embedded text.** Expressions inside the curly "{ }" brackets will alleviate the authors from editing display icons (and possibly moving them). Be careful to plan where the text fields word-wrap.
- **Lock the displays.** Set Modify / Icon / Calculation to read "Movable:=FALSE" so the authors can't move the display during testing. Select Modify / Icon / Properties Layout tab and set Positioning to On Screen so if the authors move a display (by accident), it re-displays in the correct location.

Tips for Director

- **Create all graphics at full screen and trim the excess.** This technique will allow all graphics to be "centered"—when dragged to the *score* (not the *stage*). Be aware when you import graphics that Director "shrinks" down to the non-white borders of your image, so be sure to have something that's off white in the corners (you could put registration dots in the corners).

Example

The example I've chosen was a simple multiple-choice template where we used embedded text so the authors couldn't move the text. Notice that in Figure 4, the on-screen text consists of expressions that "parse" the value of "AnswerString".

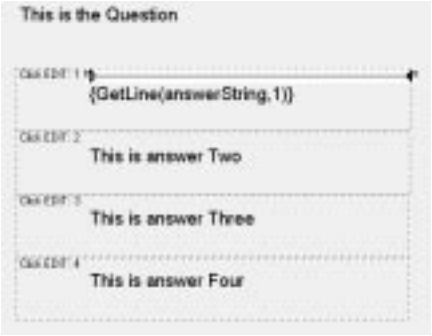

Figure 4. On-screen text is based on the value of the variable "answerString".

The authors simply defined the "AnswerString" in a calculation icon (see Figure 5). One feature we added was to have the text dim out after the user chose the wrong answer—notice that in Figure 6, "Portland" is dimmed out. There *were* two separate *display icons* (one black, one gray), but the authors didn't need to type the text twice, of course. However, this demonstrates why you should be mindful of how text word-wraps, because if the dimmed version of "Portland" word-wrapped, it wouldn't cover the original text properly.

Type 2: Engine for author

The next level of sophistication is to make "helpers" for the author. In the previous type of template, there was a lot of copying and pasting going on … this should set off alarms in efficiency-conscious authors. It's better to build an *engine*, something so dynamic that it can create multiple instances of itself, or automatically make copies. As this template is supposed to help the author, let's think of what kinds of things would help the author. The two potentially problematic issues that immediately come to mind are placement of graphics on-screen and typing data (and possibly typos) into our project. We can automate both of these.

This "engine for the author" needn't be so solid that no one can break it; rather, there can be a set of rules that each author must follow (and these rules can require knowledge of the authoring tool). This template *can* do its work at runtime, but it can also just be an author-time "helper" (for example, if reading in data, processing it, and then displaying it is too time-consuming for the user to endure). Our template, in this case, could generate "hard-wired" scripts automatically that would relieve the author as well as eliminate the potential for typos.

This type of template is valuable because it automates much of the author's work. Although there's a fair amount of development upfront, it's still less than it would be if you were creating a template so durable anyone could implement it. One disadvantage, though, is that the code is "locked down" after production—you have to re-open the source files to make any changes to the *way* the template functions.

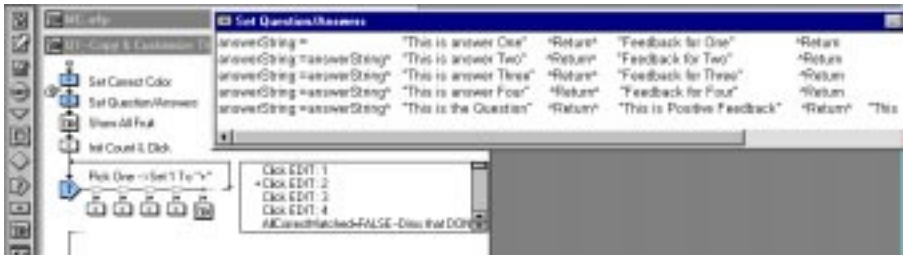

Figure 5. Everything on-screen is defined in calculation icons, not display icons.

Figure 6. Text on-screen that the author never actually touched.

Tips for Authorware

- **Use expressions that reference *"IconTitle".*** Before you put any graphics in a display, title the icon "any name EDIT:2" and set the icon's *layout* tab *properties* to "*On screen*" or "*In Area*". Then put an expression (not an explicit number) in the "*initial*" field. The expression can make reference to the icon's title, like "GetNumber(GetNumber(1,IconTitle),xLocList)", which, translated, means take the first number in our icon's title ("2," in this case) and from our custom variable "xLocList" (of all the x-coordinate locations) take the second number you find. Use this display icon for every graphic, and the author will only have to position the image *once*! If you want the display to position itself somewhere else, just edit the first number in the icon's title … or globally change the value of "xLocList".

Tips for Director

- **Use *"beginSprite"* for initial settings of sprites.** For instance, to automatically center a graphic, use the following:

```
on beginSprite me
    set the locH of sprite the spriteNum of me to _
       (.5 * (the stageRight-the stageLeft))
end
```

- **Use slow author-time index building routines.** But for the runtime version, "lock down" your code after the last save so it runs fast. For example, in a recent project, we were plotting data from many life insurance policies. Ideally, we could (at runtime) read the data from text files and plot the values. However, there was a lot of data, and each graph required significant processing (such as scaling all the data depending on the range of values). Instead, we wrote a "processing script" to read in the data, process it, and save it to *field members* (48 graphs took three minutes to process!). At runtime, we simply dumped the data from the *fields* into *lists*, and the graphing was lightning fast.

Type 3: Runtime style guide

This template is simply an organized way for non-authors to consistently implement content. There must be controls in place to prevent simple mistakes from sabotaging a project—either error checks that prevent the piece from running, or proofing scripts that confirm everything is in place.

To make this template so anyone (with or without authoring skills) can use it, we'll want to keep all the content (or data) external to the code file. This can mean developing simple bitmapped images or "ini" files to reside adjacent to your delivered application. Of course, if you plan to use bitmapped images, the content people must have the skill to edit or rename image files.

This design is quite valuable because changes to the content can be made without the expert authors (or their expert charges). Also, these templates are great for projects that get localized to other cultures or languages because the translators can implement the localized content. You should recognize that although the content people might not need authoring skills, they will need whatever skills you require in your design (maybe editing bitmaps, for instance). The point is, these skills are probably easier to find than, say, Director expertise.

One disadvantage is that there's a fair amount of up-front design cost—and if the project doesn't require it (or the content people don't take advantage of it), then the investment is wasted. Also, more and more sophistication usually adds an inordinate amount of work (these templates are usually kept fairly simple to avoid extra work).

Tips for Authorware

- **Using the "ReadExtFile" function is great, but it can be a bit touchy if you use special characters as delimiters.** Everything will work fine if the content people follow your rules—but one extra character in the wrong place and your "ini" file could fall apart. I'd highly recommend "baReadIni" from the "BuddyAPI Xtra" (note that this is a third-party Xtra). This way, you can require that the content people simply follow "rules of ini files" or use an "ini" file editor.
- Remember that when using an externally linked image (with a variable like "=curImage"), the display icon will need to be updated after you change the "curImage" variable. This is done in one of two ways: Either set the *display icon's* properties to "*Update Displayed Variables*" (which negatively affects performance slightly), or use a *calculation icon* with the following two lines:

```
EraseIcon(IconID@"Display Icon")
DisplayIcon(IconID@"Display Icon")
```

Remember, though, that the display's erasing characteristic will now be based on wherever this *calculation icon* is placed.

Tips for Director

- **Using the FILEIO Xtra is fine** (though maybe not the easiest thing to use) … but the same warnings apply as with Authorware's "ReadExtFile" function. Use a third-party Xtra and your life will be easier.
- **If you have several members "Linked to External Files," but they don't appear on stage at the same time, consider using *one* member and setting "the filename of member" as needed.** This can significantly improve performance because every one of those external links will be established (and take up memory) when your file opens.

Example

This involves a tutorial for the Intel Video Phone that completed in less than one week. When we finished, however, it was sent out to be translated into 17 languages. Needless to say, we designed most of the screens without text—less text, less translation, right? On the screens *with* text, we linked to an external bitmap file. There were only a few screens: "intro.bmp", "exit.bmp", and "help.bmp". The content people simply created full-screen (640x480) images and placed them in the "BMPS" folder next to the projector that we delivered. The same method was used for the few audio files (see Figure 7). The only place something could go wrong was in the image creation. The artists prevented those problems by creating some FreeHand templates, but that's another story.

Type 4: Runtime engine

The idea here is to build a template that can accept a few initial variables and transform itself to reflect the content it's accepting. For example, you could send a template information about the size and location of a highlight box. Then at runtime, the template could display the highlight in the correct place—never requiring an author to physically draw or position the graphic. A characteristic of this template type is that only one instance of the code-template exists in the final product—but there are many instances of *data* (that is, the variables that are different every time you use the template).

Tips for Authorware

- Hang your template off a *framework icon* with a *Nearby Exit Framework/Return navigation icon* under the template. Use a *navigation Call and Return* to get the user *to* your template—just be sure to set the initial variables before navigating.
- Put your template in a *map icon* that's hanging off a *scope perpetual conditional response* (*Erase* set to "*Before Next Entry*" and *Branch* set to "*Return*"). The conditional expression can reflect a variable that you set when you want the user to jump to the template (like "quizTime=1").

Figure 7. Storing externally linked files.

Tips for Director

- Put your template in a *movie in a window* (MIAW). Use "*tell*" to send variables between the MIAW and your main movie ("*the stage*").
- Keep the code in an *external cast* and lock or *protect* that file. This is more to ensure consistency than to prevent authors from "stealing" your code.

Designing a real-world template

These four types of templates represent the four extremes of the dynamic / hard-wired and author / anyone-modifiable scales. However, every template created doesn't fall neatly into one of these four types. Often, the templates you design will have characteristics from several different types. When you do create a template for your own production, there's a series of steps worth following.

You shouldn't simply pick one of the four general "types" of templates—that would be a case of fitting the job to the tools. Besides, picking one doesn't actually help you *design*. The process really involves a hard look at the content. We want a template that uses a proven communication or learning model—but one that can be used repeatedly without becoming stale or redundant. On the one hand, you can rely on the users' previous knowledge (making it easier for them to learn), and on the other hand, you run the risk of creating something that's boring to the user because it's repetitive.

Sequence to designing a template

- **Imagine the ultimate application, with each screen unique.** Like Stephen Covey's *The 7 Habits of Highly Effective People*, the first step to designing a template is to "begin with the end in mind."
- **Categorize the content.** How is it the same, and how is it different?
- **Define models, trying to incorporate as many variations as you can within each model.** Now, design your models trying to incorporate as many of the categories you've determined—don't forget the user, but try to streamline the content. It's also important not to make too many different models that are really the same—for instance, you don't need a "multiple choice with three answers," "multiple choice with four answers," and so on. Rather, try to make a "multiple choice with X answers." Also, you can sometimes find similarities within a model's variations—like if you had a "multiple choice with one right" and a "multiple choice with multiple right," there's really only one type—the one with multiple right. The one with just one right is really a "multiple" of one. Try to combine these variations within one "*super model.*"
- **Create a few prototypes based on representative content.** Make what's called a "full path review." Don't finish every module, section, and page—rather,

complete one module, one section, and a few pages. Remember to pick representative content—take more paths if that's necessary to reach all the content.

- **Analyze results and repeat as necessary**. The idea is to keep repeating the cycle until you're close to the ideal … you'll never *really* get there, but the only way to get closer is to complete the whole cycle (design, build, analyze).
- **Develop a vocabulary for all team members.** This can be arbitrary—it doesn't matter if you call them "sections" or "lessons," but everyone should follow the same convention.
- **Build a table for content.** This should be designed by the people importing the content—but since it's *for* the people writing the content, the priority should be on making it easy and clear for them.
- **Add dummy content to a representative template.**
- **Write "proofing scripts" (especially important for dynamic templates).** Instead of quality assuring every little bit of your project by hand (a time-consuming process fraught with potential errors), consider writing a script that will *do* the error checks before delivery.
- **Test at logical milestones to protect yourself from heading too far down the wrong path.** The "logical" point might be hard to determine, but simply try to prevent expensive rework.

Remembering why we're here

Good templates pay for their own design cost in savings later. The larger your project, the larger the savings. Consider this investment when spending time building a template—a template that's worth more is worth spending time on. Also, it's easy to get hung up making something totally for the benefit of the developer—that is, yourself. As good developers, we must remember to acknowledge our "leader"—the user. ▲

Phillip Kerman is an internationally recognized expert on creating multimedia for training and entertainment. Specializing in Macromedia tools for six years, Phillip has produced rapid prototypes, adaptable templates for large projects, and software that enables easy localization. Expanding on his work as a developer, Phillip currently teaches courses and makes presentations around the world. Within the past 12 months, he's been invited to present and teach in Australia, Iceland, San Francisco, and Toronto, as well as his hometown of Portland, OR. Feel free to contact him at phillip@teleport.com or http://www.teleport.com/~phillip/.