APPLYING MODEL-VIEW-CONTROLLER (MVC) IN DESIGN AND DEVELOPMENT OF INFORMATION SYSTEMS

An example of smart assistive script breakdown in an e-Business Application

Andreas Holzinger, Karl Heinz Struggl
Institute of Information Systems and Computer Media (IICM), TU Graz
a.holzinger@tugraz.at, kstruggl@student.tugraz.at

Matjaz Debevc
Faculty of Electrical Engineering and Computer Science, University of Maribor
matjaz.debevc@uni-mb.si

Keywords: Information systems, software design patterns, Model-view-controller MVC, script breakdown, film production

Abstract: Information systems are supporting professionals in all areas of e-Business. In this paper we concentrate on our experiences in the design and development of information systems for the use in film production processes. Professionals working in this area are neither computer experts, nor interested in spending much time for information systems. Consequently, to provide a useful, useable and enjoyable application the system must be extremely suited to the requirements and demands of those professionals. One of the most important tasks at the beginning of a film production is to break down the movie script into its elements and aspects, and create a solid estimate of production costs based on the resulting breakdown data. Several film production software applications provide interfaces to support this task. However, most attempts suffer from numerous usability deficiencies. As a result, many film producers still use script printouts and textmarkers to highlight script elements, and transfer the data manually into their film management software. This paper presents a novel approach for unobtrusive and efficient script breakdown using a new way of breaking down text into its relevant elements. We demonstrate how the implementation of this interface benefits from employing the Model-View-Controller (MVC) as underlying software design paradigm in terms of both software development confidence and user satisfaction.

1. INTRODUCTION

The process behind the production of a motion picture, be it a documentary, an action movie or an animation film, is typically segmented into the four phases of development, pre-production, production (or principal photography) and post-production. Analogous to other fields where project management is applied, the first phases of the film production management (FPM) process not only define and affect all efforts and results involved in the whole production, but also essentially decide whether the project will eventually be green-lighted (Clevé, 2005). The producer's main responsibility in these first defining phases is to formulate the project (in terms of finding a screenplay script, acquiring licenses and possibly receiving commitment by important staff) and to obtain funding for the actual realization of the project. For the latter task, it is vital that the producer create robust cost estimations. Contrary to later phases, where actual budgets are available (Singleton, 1996), these first cost estimations need to be based on cost defining elements in the script, which are identified and quantified by conducting script breakdown (Singleton, 1991). This paper will first present the results of an observation on script breakdown interfaces of major film production management software suites and motivate the need for an

unobtrusive and efficient script breakdown interface. Following this, the model-view-controller (MVC) pattern and its relevance to usability by software design will be discussed. The proposal of auto-advancing shortkey (AAS) tagging will then follow and lead to a discussion of how MVC was adopted in the system design phase and supported by the development environment. An observation of lessons learned and benefits of the provided solution will conclude this work.

2. SCRIPT BREAKDOWN IN THE FILM PRODUCTION PROCESS

Script breakdown is the act of analyzing the script of a film production (e.g. a movie screenplay), identifying its defining, relevant elements, and grouping them into categories (Clevé, 2005). For example, all speaking roles in a script are identified and assigned the "Actors" category.

The resulting breakdown information is, as noted, used to create first cost estimations. This is usually done by consulting so-called labor rate books and price lists provided by guilds and manufacturers, and applying them to the elements of the breakdown. For example, if in a given scene there are five extra cast members noted (e.g. to represent passengers waiting for a train), a labor rate book would be consulted that lists typical costs for such silent performers according to any possible special requirements, such as age or shooting conditions. Thus, it is essential to build such cost estimations on correct script breakdown data.

Furthermore, so-called breakdown sheets (Singleton, 1991) are created for every scene, providing a tabular summary of all contained elements, along with other production notes about the time of day the scene takes place, and similar. These breakdown sheets are of the utmost importance for the scheduling (time-planning) of the project. Also in later phases, they are considered the main reference document by all personnel involved in the preparation and shooting of scenes.

Historically, script breakdown has been done by printing out the script and highlighting elements with textmarkers that are color-coded according to their respective category (Singleton, 1991). Obvious drawbacks of this solution are the time required to conduct the breakdown and to transfer the data into breakdown sheets, scheduling plans, etc., as well as its error-proneness.

In recent years, film production software applications started adopting the idea of script breakdown and providing interactive interfaces for it. However, due to deficiencies in the solutions provided by these applications, many producers still resort to script printouts and textmarkers, and transferring and updating data manually. Analysis of the script breakdown interfaces of five major film production and management software suites (CeltX, Final Draft and Final Draft Tagger, Cinergy MPPS, Movie Magic Scheduling, Movie Magic Screenwriter) provided the following results:

1. Most often supported was fully manual tagging, i.e. tagging by manually selecting or entering text and category, either with or without the use of a separate tagging dialog window (all suites).
2. Tagging selected text into categories by context menu or category-buttons was supported by three suites.
3. One suite allowed to assign categories to the selected text by shortcut keys (e.g. "a" for actors).
4. Two suites provided a breakdown sheet-preview that could be manipulated interactively.

Two conclusions were drawn from these findings. First, most observed suites did not rely on one particular tagging mechanism, but provided up to three different interfaces. While this fact in itself does not disqualify any of the solutions, it shows that software design must be flexible and able to support different workflows with various involved interface elements independently of the underlying data model. In the light of this observation, it must be stated that some issues concerning this requirement were discovered. This surfaced e.g. in the use of separate (unnecessary) dialog windows in three suites, or the fact that solutions using two or more windows only allowed the use of one window at a time, while the others were disabled. The system architecture presented later in this paper will employ MVC and point out how these problems can effectively be avoided by software design.

Second, all provided interfaces required the user to carefully select (or worse, manually input) text in order to tag an element with a category. This will obviously become difficult with typically dozens of pages long film scripts and therefore presents a likely reason for user frustration. The tagging interface proposed in this work will provide smart pre-selection of text in order to mitigate this problem.

3. METHODS AND MATERIALS

3.1 Auto-advancing shortcut key tagging (AAS)

The idea of AAS is based on the findings presented in section 2. It combines the two main principles of auto-advancing using smart text pre-selection heuristics, and the use of shortcut keys for assigning categories.

The pre-selection heuristics are responsible for finding the range of text that is most likely to be selected by the user, and, as the name implies, pre-selecting it, effectively mitigating the problem stated before. The heuristics themselves should be defined and implemented at one single point and therefore usable for various kinds of workflows. These include e.g. triggering after user has tagged an element (the program automatically selects the next likely element, i.e. auto-advancing) as well as normal navigation.

The rules implemented by the heuristics have to be carefully designed and defined to suit the actual use case. In the context of script breakdown, one important rule is that the user will never select single characters or parts of words for tagging. Therefore, the smallest granularity for selection is a word, i.e. upon auto-advancing (e.g. after tagging), the heuristics will select the next word after the current selection.

In order to provide more meaningful pre-selections, the heuristics should consider previously tagged elements, which may consist of multiple words, and include language-independent recognition of word boundaries.

The latter is a non-trivial task due to e.g. context-dependent meanings of punctuation (as shown in) and language-specific ways of defining word boundaries.

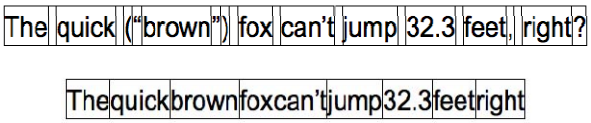


Figure 3.1: Segmentation of an English sentence by word boundaries. Original sentence at the top, extracted version at the bottom. [Image extracted from (Unicode, 2009).]

Additionally, more complex language-specific processing can be included for better results, e.g. using specialized stopword-filters or word dictionaries. Assume, for example, the following text fragment as part of a movie script:

```
the Tin Man carries an axe
```

Assume also that the word `the` is currently selected by the user. Upon advancing, the pre-selection heuristics will select the word `Tin` for the user. Now, assume the same text fragment, but with the `Tin Man` already tagged, e.g. as speaking role:

```
the Tin Man carries an axe
```

In this case, the pre-selection heuristics would recognize the words `Tin Man` as compound token and select both for the user. Next, assume the user advanced once more:

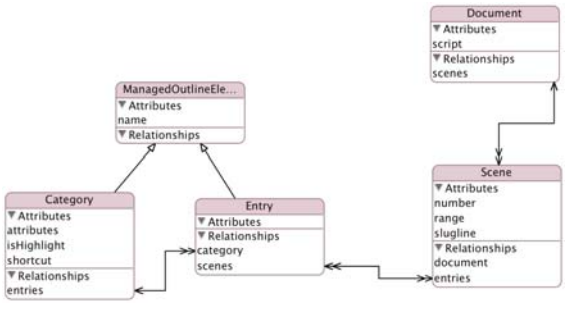
```
the Tin Man carries an axe
```

In this case, stopword-filtering can be applied to skip over the word `an` as it is reasonably safe to assume it will never represent an important element in an English script.

Shortcut key tagging of elements into categories on the other hand is a more user-centric design task. It requires a simple and flexible interface for managing categories and assigning them shortcut keys (and possibly formatting information, such as colors and underline- or highlighting-styles).

Both the pre-selection and the shortcut key tagging mechanisms inherently require the software to implement workflows that interact with several interface elements and controllers to reflect the user’s actions in the software and in its data model.

3.2 Development Environment and Tools



The prototype described in the following sections was implemented on and for Apple Macintosh machines running Mac OS X 10.5 or above.

Design and implementation were done using the Xcode development environment, which is freely provided by Apple for any Mac OS X owner or member of the Apple Developer’s Connection (ADC).

The underlying data model was built using Xcode's data model builder. All object persistence is managed by its Core Data module. Interface design was done using Interface Builder.

Classes and frameworks of the Mac OS X-native API Cocoa were used and partly subclassed and extended to implement the actual functionality and user interaction. It is to note that Xcode and the higher-level Cocoa API frameworks themselves employ MVC and support and advocate its use as design pattern.

3.3 Prototype Software

The prototype was developed with MVC as main underlying design pattern. MVC is an architectural software design pattern and engineering concept described by Trygve Reenskaug (Reenskaug & Skaar, 1989) (see Figure 3.2).

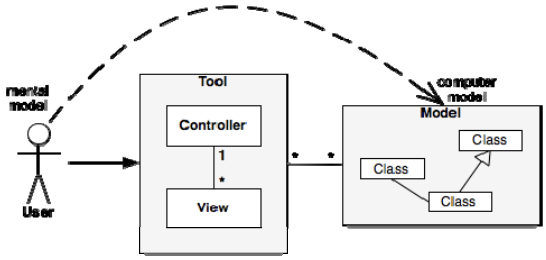


Figure 3.2: The general idea of the Model-View-Controller (MVC) paradigm, altered according to Reenskaug (Reenskaug, 2008)

3.3.1 Model

Analysis of the typical structure and the characteristics of movie scripts (as e.g. provided in (Clevé, 2005)) yielded the data model depicted in Figure 3.3.3.

Figure 3.3: The prototype's data model as abstracted from real world entities.

Category and *Entry* share a common super-class that provides persistence and usage as outline items in the interface, as well as attributes and methods both subclasses need to. Category additionally administers a relationship to its elements (i.e. Entry objects representing text added to this category) and on how they are formatted in the script view. Entry on the other hand knows which Category it belongs to, and in which Scenes its specific text is tagged.

Scene objects are used to optionally limit the scope of tagging to scene boundaries. Also, they store references to all entries within them for the purpose of creating structure breakdown sheets and similar. Scenes are recognized and pre-processed automatically when a script is loaded into the breakdown prototype. The *Document* class is used for storing project information in order to enable saving and loading of projects including their scripts and all tagged elements.

It is to note that the entities defined in the data model above are represented in code by their respective model classes, e.g. *Category* is implemented in a *Category* class in the *Model* directory and implements properties and methods pertaining only to itself. Inter-class workflows or interface functionality is implemented by controllers.

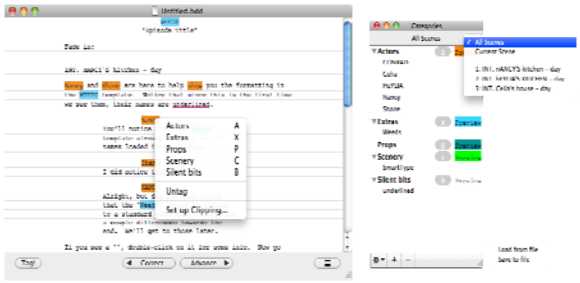
3.3.2 View

As noted, interface design was done using Xcode's Interface Builder. All controls and interface elements shown on the screen are implemented by so-called view-classes which are responsible for drawing themselves in order to create the user interface, as well as holding references to any controllers they may incorporate. E.g., a text view not only draws text (i.e. view code), but also provides text manipulation, which is handled by the views' text controller.

Where required, view and controller classes were subclassed to incorporate new or override existing behavior. E.g., the text view class was extended to handle shortcut keys and trigger pre-selection correctly after the according button presses.

Figure 3.4: Breakdown prototype's main window and category management window including context menus and scene selection.

Figure 3.4 gives an impression of the prototype's script view and category management interfaces. The main window supports tagging by AAS and



alternatively by the use of a tagging context menu. The category management window features a scene selection menu, an outline view of all categories and their respective elements, and controls for manipulation and saving/loading category configurations.

3.3.3 Controller

Two controllers in the breakdown prototype are responsible for implementing workflows and providing other general functionality across interface elements and involving several data model classes. Figure 3. shows how responsibilities between the two controllers are organized, including workflows and management of data model objects.

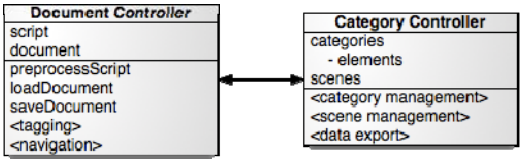


Figure 3.5: Responsibilities of the prototype's controllers.

The *Document Controller* is the core unit of the breakdown prototype as it implements saving and loading of projects and manages *Document* instances. It also encapsulates pre-processing of the script (for scene scanning) and tagging of elements using AAS (including pre-selection), the context menu or the *Tag*-button.

The *Category Controller* provides an interface to the *Document Controller* for tagging and handles user initiated category management itself. It also features a scene selection menu which is automatically generated from the scene data gathered by the *Document Controller*'s script pre-processing and can be used to bind tagging to a scene's scope, as well as to jump to the beginning of the scene in the script view.

Both the scene menu and the actual tagging process therefore present valid examples of how the system architecture effortlessly allows workflows to span over different controllers, interfaces and data model entities.

4. LESSONS LEARNED

It was noted before that employing MVC offers benefits for software systems dealing with a range of data model objects that the user can manipulate using an interactive interface. In order to give the user an impression of direct manipulation, as often discussed in software usability engineering, the prototype was implemented in a way that no user interaction or interface elements (e.g. tagging dialogs) are required that are not inherently part of the respective workflow (contrary to several of the observed film production software suites).

For example, tagging an element using a shortcut key first invokes an event on the text view, informing it about the user's keystroke. The subclassed view knows to forwards the event to its delegate (i.e. the controller class handling such user input, in this case the document controller) for handling. The controller then invokes the AAS algorithm which itself uses a native library for word boundaries scanning. For every tagged element, the category controller is called with the required information to create an element object.

In short, this example shows that no artificial interface class or any inclusion of workflow logic in model classes is required. The document controller interprets and handles input as notified by the script view and invokes the category controller for creating the appropriate model class objects. The same applies e.g. for the script pre-processing step where the document controller scans for scenes and invokes the category controller to create scene elements.

Without the use of MVC, it is likely the system design would need to incorporate artificial pseudo-controllers to handle such workflows. In the system architecture presented herein, there are only two controllers responsible for a cohesive group of views and interactions using a clearly defined interface to implement complex workflows.

Another important aspect is reusability. For example, the algorithm for tagging a particular text is also used for un-tagging the text (i.e. removing from the category and un-formatting all occurrences in the text) and for re-formatting all of a category's occurrences if the category formatting itself has been changed. This is supported by the system design automatically since no additional user interaction (e.g. tagging dialog) is required. For the user, this means that un-tagging an element can be done by either selecting the element in the script (using AAS this is done semi-automatically) and hitting the backspace key, or by deleting the element from the categories outline view.

5. CONCLUSION

Work on the prototype has shown, as discussed, that MVC strongly advocates a system design that efficiently reflects workflows, business logic and therefore user interaction and its expected results in a way that is both intuitive for the user and plausible for the system engineer.

It can be said that MVC helped in minimizing the compromises between system usability and ease of development. In other words, most of the features and usability considerations of the prototype could have been taken into account and realized with other software development paradigms, but likely with the introduction of tradeoffs towards either system design or usability (as, again, was seen in several of the observed film production software suites).

Recommendable future work includes research of other, possibly similar software development paradigms for interactive applications and evaluation of how well they would suite the prototype at hand. For example, the Document-Context-Interaction (DCI) architecture recently proposed by Trygve Reenskaug suggests a possibility for improvement over MVC by introducing roles to integrate generically defined workflows and business logic into the software design model (Reenskaug, 2008), (Reenskaug & Coplien, 2009).

6. REFERENCES

Bass, L. & John, B. E. (2003) Linking usability to software architecture patterns through general scenarios. *Journal of Systems and Software*, 66, 3, 187-197.

Booch, G. (1994) *Object-Oriented Analysis and Design with Applications.* Redwood City (CA), Benjamin/Cummings.

Clevé, B. (2005) *Film Production Management.* Burlington, USA, Oxford, UK, Focal Press.

Curry, E. & Grace, P. (2008) Flexible self-management using the model-view-controller pattern. *IEEE Software*, 25, 3, 84-90.

Dahl, O.-J. & Nygaard, K. (1966) SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9, 9, 671-678.

Holzinger, A. (2005) Usability Engineering for Software Developers. *Communications of the ACM*, 48, 1, 71-74.

Juristo, N., Moreno, A. M. & Sanchez-Segura, M.-I. (2007) Analysing the impact of usability on software design. *Journal of Systems and Software*, 80, 9, 1506-1516.

Kay, A. C. (1993) The early history of Smalltalk. *The second ACM SIGPLAN conference on History of programming languages.* Cambridge, Massachusetts, United States, ACM.

Kristaly, D. M. & Moraru, S. A. (2006). *Java technologies for model-view-controller architecture*. 10th International Conference on Optimization of Electrical and Electronic Equipment (OPTIM 2006), Brasov (Romania), Transilvania Univ Press-Brasov, 175-178.

Leff, A. & Rayfield, J. T. (2001). *Web-application development using the Model/View/Controller design pattern*. 5th IEEE International Enterprise Distributed Object Computing Conference, Seattle, Wa, Ieee Computer Soc, 118-127.

Lethbridge, T. C. (2000) What Knowledge Is Important to a Software Professional? *IEEE Computer*, 33, 5, 44-50.

McLaughlin, B., Pollice, G. & West, D. (2006) *Head First Object-Oriented Analysis and Design.* Sebastopol (CA), O'Reilly.

Oestereich, B. (1999) *Developing Software with UML: Object-Oriented Analysis And Design In Practice.* Harlow (UK), Addison Wesley.

Reenskaug, T. (2008), The Common Sense of Object Orientated Programming. Online available: http://heim.ifi.uio.no/~trygver/themes/babyide, last access: 2010-02-20

Reenskaug, T. & Coplien, J. O. (2009), The DCI Architecture: A New Vision of Object-Oriented Programming. Online available: http://www.artima.com/articles/dci_vision.html, last access: 2010-02-10

Reenskaug, T. & Skaar, A. L. (1989) An environment for literate Smalltalk programming. *Conference proceedings on Object-oriented programming systems, languages and applications.* New Orleans, Louisiana, United States, ACM.

Seffah, A. & Metzker, E. (2004) The obstacles and myths of usability and software engineering. *Communications of the ACM* 47, 12, 71-76.

Singleton, R. S. (1991) *Film Scheduling, or, How Long Will It Take To Shoot Your Movie?* New York, Lone Eagle Publishing.

Singleton, R. S. (1996) *Film Budgeting, or, How Much Will It Cost To Shoot Your Movie?* New York, Lone Eagle Publishing.

Unicode (2009) Unicode Standard Annex 29. *Unicode Text Segmentation.*