2016

Building an application for the writing process

BOSTON UNIVERSITY

METROPOLITAN COLLEGE

Thesis

BUILDING AN APPLICATION FOR THE WRITING PROCESS

by

AMOD LELE

B.A. (Hons), McGill University, 1997
M.S., Cornell University, 2001
A.M., Harvard University, 2002
Ph.D., Harvard University, 2007

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2016

Approved by

First Reader _____

Robert Schudy, Ph.D.
Associate Professor of Computer Science

Second Reader _____

Eric J. Braude, Ph.D.
Associate Professor of Computer Science

BUILDING AN APPLICATION FOR THE WRITING PROCESS

AMOD LELE

Boston University Metropolitan College, 2016

Major Professor: Robert Schudy, Ph.D., Associate Professor of Computer Science

ABSTRACT

The idea that writing is a process and not a product is now generally accepted in writing education, but discussions of digital scholarly communication often neglect the idea, in theory and in practice. This thesis report introduces a Mac OS X software package to support the early stages of the writing process, called Brouillon. Brouillon's features include: the concatenation of discrete note files into notebooks; notes appearing in multiple notebooks; note intake from mobile devices via Dropbox; and an open standard file format. The report also provides a model of the organization of products of the writing process, with a focus on Brouillon's most unusual feature, multi-notebook notes. It discusses difficulties in implementation and identifies possibilities for future improvement.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION: THE WRITING PROCESS

In recent years, scholars have paid increased attention to the value of new digital tools for academic communication in both scholarship and pedagogy (e.g., Borgman 2007; Association of College & Research Libraries 2003). A relatively neglected—though pedagogically crucial—aspect of communication is the *writing process* itself. In the 1970s Donald Murray (2003) popularized the idea, now commonplace in writing instruction (e.g. Ede 2003) that writing needs to be thought of as a process, not merely a final written product. This literature on the writing process, however, has rarely reached the scholarship on digital communication. Nor are adequate technology tools available to handle the writing process.

Consider for example Christine Borgman's (2007, 47–74) discussion of the process by which a scholarly work eventually reaches publication. Borgman begins from an old but well known chart illustrating this process, originally published by Garvey and Griffith (1964). The 1964 chart, reproduced here as Figure 1, depicts the process of scholarly communication in a way that moves directly from "starts work" to "work reaches report stage", as if one could simply sit down and start writing a major scholarly work from the first word—an assumption still made by many undergraduates, to the work's detriment. Borgman (2007, 49) notes how the technological changes of the following decades have changed the kinds of communication that Garvey and Griffith examine, but does not critique their glossing over all the portions of writing from start to report: this part of the process of scholarly work is still neglected. Technology use in writing reflects the kind of thinking depicted in the chart: many scholarly writers still rely

on Microsoft Word for the entire writing process, a thirty-year-old application that, despite many updates, is still designed on the assumption that all one's writing will end up on the printed page.

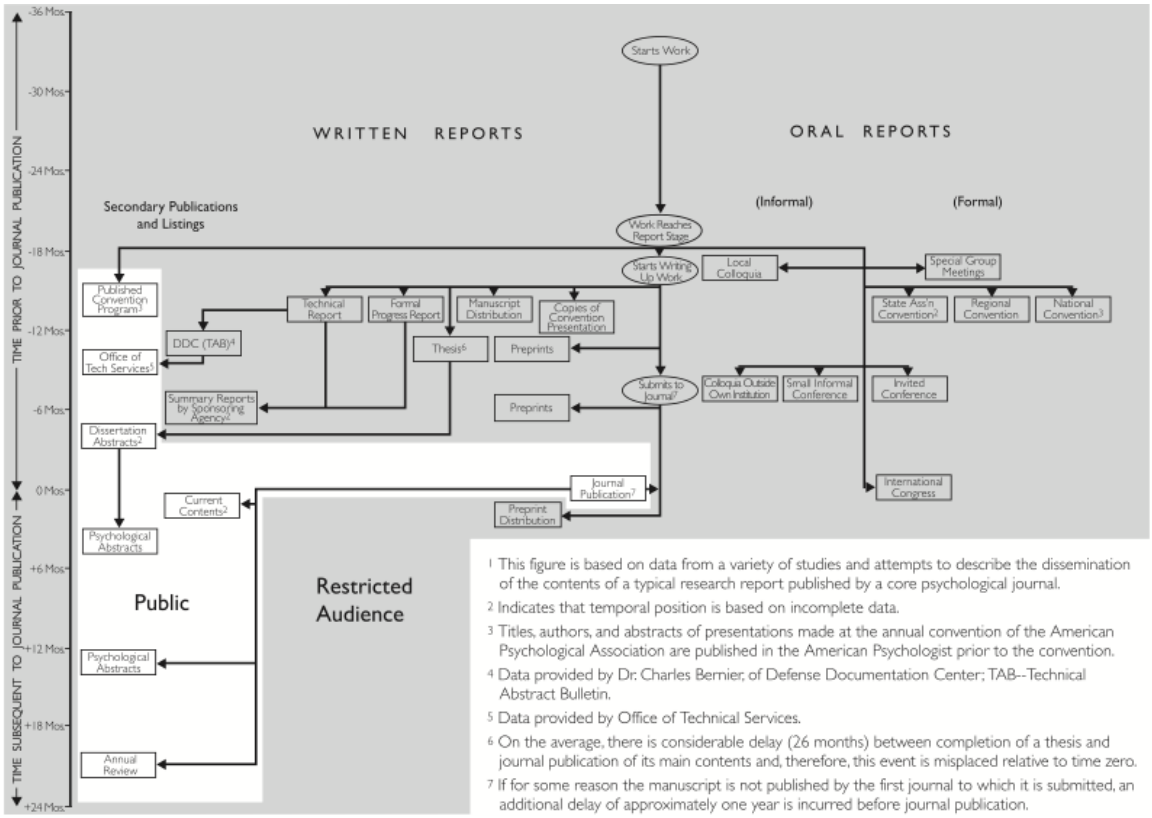

Figure 1: "The Dissemination of Scientific Information in Psychology", from Garvey and Griffith 1964

What is the process by which one moves from start to report? A key element to the writing process is early-stage writing, writing that will not necessarily make it into any final draft. Murray (2003) coined the term *prewriting* to describe steps like brainstorming, freewriting and notetaking that can precede the writing of even a first draft. Such steps are central to student-centred approaches such as that of Peter Elbow (1973). However, later work (Flower and Hayes 1981a) pointed out that the term

"prewriting" is misleading because it gives the impression of discrete stages, such that these reflective elements of the writing process take place entirely before any writing of the final product; in fact, they are also likely to happen during revising, after a first draft is written.

The idea of "prewriting" suggests a model of discrete stages similar to the now oft-maligned "waterfall" model of software engineering. As Sommerville (2009, 33) notes, an incremental model "is better than a waterfall approach for most business, e-commerce, and personal systems" because it "reflects the way that we solve problems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake." The research of Flower and Hayes (1981a, 1981b) shows a similar point applying to writing: the "prewriting steps" can happen after as well as before a draft is written. So while the concept to which "prewriting" refers is helpful, the term itself is likely less helpful than a term like "early-stage writing".

Early-stage writing includes at least four major aspects. The first of these is *brainstorming*, including freewriting (Elbow 1973, 3–13): unstructured writing for the purposes of idea generation, with no intent that any of it will necessarily be included in the final product. Joan Bolker (1998) advises that regular brainstorming both prevents writer's block (because one can always write *something*) and clarifies ideas. Speaking anecdotally from her experience as a dissertation writing counselor, she adds, "I don't think I've ever worked with a student who stuck with freewriting for whom this didn't happen." (Bolker 1998, 43) More rigorously, Glynn et al. (1982) performed experiments

finding that college students generated ideas and arguments generated more effectively when they restricted their focus to the content of the arguments without attention to structure.

The second element is *formal notetaking*: taking notes in class, and notes on books that one reads. The extensive survey conducted by Moore et al. (2016) found that as recently as 2010, students still used pencil and paper for this process more than software. But software tools can offer considerable advantages. In addition to their greater portability, they offer the ability to search one's past notes quickly as paper notes cannot. Many current tools such as Evernote hamper this latter ability because they are not integrated with the basic filesystem, so one must search in at least two places for one's previous notes.

The third element of the writing process served by this application is *casual notetaking*: taking down ideas as they come. Bolker (1998, 47) encourages writers to "write down every idea you have". For this reason, nineteenth-century intellectuals often carried a *cahier de brouillon*, a paper notebook to take down thoughts as they occurred.

Finally, there is the movement from the first three elements to a draft. From her consulting with students stuck at various stages of the writing process, Bolker (1998) came to encourage the writing of "zero drafts": the collection of notes together into a "rich soup" which can be turned into more structured writing.

These various early products of the writing process, moreover, can have further value when shared beyond the writer. The Force11 manifesto on research communication (Force11 2011) has pointed out the value of sharing raw data as well as finished products

in the sciences. In the humanities, the notes of the writing process effectively *are* the raw data, and sharing them provides a similar access to the thought processes that underlie the finished products. This has been true even in the pre-digital era; manuscripts such as Karl Marx's *Economic and Philosophic Manuscripts of 1844* or Friedrich Nietzsche's Nachlaß have proved invaluable to later scholars even though they were not published during the writers' lifetimes.

BROUILLON: AN APPLICATION FOR EARLY-STAGE WRITING

Traditional word processors are focused on a print model, divided into pages as if the work will inevitably make it to print. Such a model has significant disadvantages in the 21st-century environment where much scholarly writing is *multimodal*: as Gould (2014) notes, much scholarly work today is "born digital", aimed for venues like online journals and blogs that are not confined to print. It also is not designed for early-stage writing, which, if done digitally, is unlikely to make it to print. Yet the 2010 survey of Moore et al. (Moore et al. 2016) noted that college students use word-processing software even in the composition of email and text messages, likely for drafting. A born-digital tool designed specifically for the writing process would be a boon to students and other writers. Some available tools exist for this purpose, most notably Evernote, Tinderbox (see Bernstein 2003) and Scrivener, but their limitations will become clear in the discussion.

The present thesis project consists primarily of an application for the writing process, entitled Brouillon after the 19th-century intellectuals' *cahiers de brouillon* and the general French word for "rough draft". The application is built for a desktop or laptop platform, since most students use laptops rather than tablets for work (Hart 2015), and specifically for Mac OS X, since the Mac is widely used in higher education (Elmer-De Witt 2010). When released, Brouillon will be a fully open-source project that anyone is welcome to build on. Brouillon's current stage of development is only proof of concept, for reasons that will be discussed later in this thesis report. However, even at this stage Brouillon has four main features that go beyond existing software for the writing process:

1. Concatenation of discrete note files into notebooks. In a word processor, one has two choices of how to store the notes that one has taken on separate occasions. One can either make each note its own separate file, which does not allow the writer's sequence of thoughts to be viewed in its order and continuity, or one can put all the notes into a larger file representing the notebook. The latter approach impedes search—if one wants to search for an old note one enter the search term *twice*, once in the filesystem and once in the word processor.

By contrast, Brouillon puts together notebooks *made up of* separate note files kept together. This allows an easy search seamlessly integrated with OS X's native Spotlight search functionality: one can enter a search term and quickly find the single Brouillon note that contained it, rather than having to then look for it again within a larger file. One can also find those files containing the search term created with another tool, since the files are together on the filesystem. This latter point stands in contrast to other tools like Evernote, which lock up one's data in a proprietary non-integrated system.

Concatenation also facilitates and enables the next two key features to be discussed: multi-notebook notes and Dropbox mobile intake. It makes multi-notebook notes possible by giving them an existence separate from the notebook that contains them, and makes intake possible because one can work on individual notes on a mobile device and have them put together on a laptop. Those who have tried to work with large notebook files in a mobile text editor such as JotterPad will know the limitations: it is very difficult to scroll through such a long file to find the data one wants to read, or the place to enter it.

2. Multi-notebook notes. Brouillon's most innovative feature is the ability to categorize the same note in multiple notebooks. (The model that makes this possible is discussed in detail below.) In this way, each note can be attached to the multiple projects it is relevant to. For example, a graduate student could use the app to take notes in class, concatenated sequentially by default so that the student can read the course's notes in order—but tag an individual note from that course so that it would also appear in her notes for dissertation preparation, comprehensive exams, article publication or all of the above.

Such an approach allows students and scholars to put together different sets of thoughts as appropriate to different contexts. Such reuse and remixing is natural to doctoral students and scholars, who might reuse ideas from a course in a dissertation or ideas from a dissertation in a journal article, but it is also valuable to undergraduates. Perkins and Salomon (1988) cite research observing that students frequently do not apply knowledge from one context to another context. When one can identify a given note as relevant in a context different from the current one, it becomes much easier to go back and find that note when one is thinking about that different context later on. The National Survey of Student Engagement (2007, 41) found that among the most helpful undergraduate learning experiences were practices such as capstone projects that "provide students with opportunities to synthesize, integrate, and apply their knowledge." Yet existing tools for the writing process, such as Tinderbox and Scrivener, tend to be wedded to a model that takes the individual project file as its unit, making it difficult for writing work to cross projects in this way.

The closest comparable feature within existing tools is hyperlinking between documents (Evernote features this, for example). Hyperlinking is a valuable feature and ideally it will be added in a future version of Brouillon in addition; it is a distinct feature from multi-notebook notes. Multi-notebook notes allow a user to retrace his or her thoughts in the order they occurred; to do this by following a hyperlink requires an additional step each time, which can become cumbersome.

3. Note intake from mobile devices via Dropbox. In the current era, no tool can handle casual notetaking unless it works easily with mobile devices, especially smartphones—the modern pocket's equivalent of the *cahier de brouillon*. Moore et al. (Moore et al. 2016) found that students in 2010 used their cell phones even to write academic papers and take notes from readings and lectures, and they recommend integrating such devices into composing strategies.

Brouillon handles note intake by identifying each notebook with a home folder, which can be on Dropbox as easily as on a purely local hard drive. This approach allows users to compose notes in any Dropbox-enabled mobile Markdown or plaintext client they wish, on any mobile platform, and have the note automatically sync to the Dropbox folder on the user's hard drive. When Brouillon opens, the note is automatically added to the notebook corresponding to that folder.

Dropbox is the preferred solution for mobile synchronization because it integrates seamlessly with the Mac (or other) filesystem. It is also free on an unlimited number of devices, unlike Evernote, which now charges its users if they wish to use more than two. Dropbox's pricing model is to charge once a large amount of data is stored. This model is

ideal for a text-based program like Brouillon, since the amount of storage space required for text is minimal; it would be a challenge to produce enough text files to fill the free Dropbox storage.

4. Open standard file format. If one's unpublished early writing is to be shared, it should be in a format with some longevity. Flexibility and interoperability enable systems to adapt to future innovations and survive disruption (Galanis et al. 2014). This is what the Association of College and Research Libraries (2003) refers to as the *preservation* element of scholarly communication. An open and standard file format is vital for this reason. This is a limitation of existing systems like Tinderbox and Scrivener, whose formats (.tbx and .scriv) are difficult to read without the tools themselves; it is doubly a problem with Evernote, which locks one's notes up on a proprietary platform and requires an exporting process to even access them outside the application at all.

The intention of Brouillon is that its files be formatted in the open MultiMarkdown, an extension of the basic Markdown format (Gruber 2004), designed to be as human-readable as possible. Markdown and MultiMarkdown codes mimic the formatting of old plain-text email (e.g. expressing italics with _underscores_), when ASCII symbols were the only symbols available. There was not time in the scope of the thesis project to implement MultiMarkdown parsing, so while Brouillon can read (Multi)Markdown files, its current effective native format is plaintext.

Still, while pure plaintext loses Markdown's ability to display rich formatting, it shares Markdown's durability. (Multi)Markdown and plaintext are just as readable now as they would have been forty years ago, unlike nearly any other markup format

available. (The present author's hard drive contains a number of thirty-year-old files he composed in his childhood, some of which are in WordStar format and some in ASCII plaintext. The reader can guess which of these are easier to read now.) It therefore stands to reason that, while one can never predict what will and will not be readable in the future, this format is far likelier to be readable many decades from now than other, more complex or proprietary, formats.

The one element of Brouillon's current format that would not have been readable thirty years ago is that its encoding is UTF-8. This standard, released in 1993, allows the system to handles a full complement of non-Roman-script files, important for work in the digital humanities. UTF-8 is now widespread enough to suggest longevity, since over 85% of websites now use it as of this writing. The author has tested and confirmed that the system handles the Devanagari ligatures important to Indian languages, as Word does not.

MODELLING THE PRODUCTS OF EARLY-STAGE WRITING

Brouillon's multi-notebook note feature is intended to provide a new way of organizing information in the early stages of the writing process, one that does not merely mimic other forms of organization. For this reason it is helpful to express the design's essential features through a conceptual model. The model shown here models the data of early-stage writing—the *products* of notetaking, freewriting and the like—as they would be organized in a useful electronic tool. As a model, it is a simplified representation of early-stage writing that could be of use to anyone making a similar application. (Since Brouillon will be open-source, this is more than a hypothetical possibility.) It models one form of *personal information management* (PIM).

Figure 2 and Figure 3 present two different entity-relationship models of early-stage writing data, one specific to an app like Brouillon and one more generalized. Figure 2 presents notebooks and notes as the key entities, related by containment, with their significant attributes—the distinctive feature of a note being that it corresponds to a searchable file in the filesystem. Theoretically, one could have larger notebook entities containing the notes. Figure 3 generalizes this relationship into a recursive model: notebooks and notes are both content units that contain other content units.

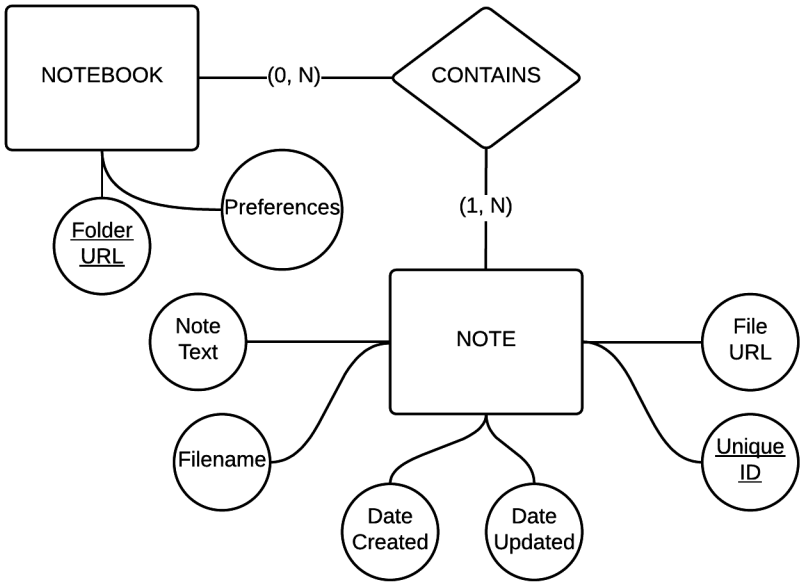

Figure 2: Model of early-stage writing data in Brouillon

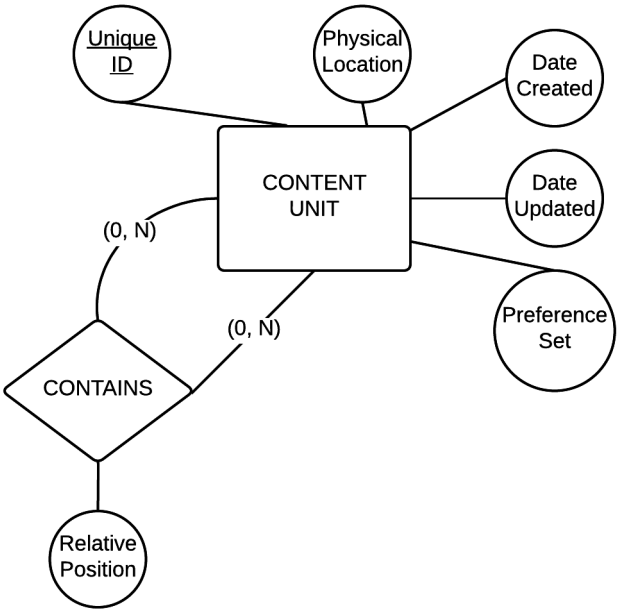

Figure 3: Generalized recursive model of early-stage writing data

Both models conceptualize their key relationship as *containment*: notebooks containing notes, content units containing other content units. The relationship might not be understood as containment in the strictest sense because it is a many-to-many relationship: the multi-notebook note features is such that more than one notebook can contain a note (as well as more than one note belonging to a notebook). The model nevertheless describes the relationship as containment because it is hierarchical or taxonomic, as opposed to the "flat" or "bushy" organization of a tag-based system (see Civan et al. 2008). The recursive model (figure 3) explicitly contains a multi-levelled hierarchy of containment; in the more specific model (figure 2) as implemented in Brouillon, additional levels of hierarchy are handled by the filesystem, because notebooks are tied to folders (directories) within the filesystem.

Pak et al. (2007) show that while tag-based systems generated a great deal of enthusiasm when they first reached wide deployment, this enthusiasm is not borne out by empirical study: while users can organize information more quickly in a tagging system, *retrieval* is slower, and users find organizing information through tags more *frustrating*. Such a conclusion seems confirmed by the work of Bergman et al. (2013), who found that users preferred a tag-based organization to search for *others'* content, but wanted a folder-based organization for *their own* content—for PIM. They provide a strong hypothesis for why this is likely to be the case:

> We think that the key issue here is familiarity. When looking for content which other users uploaded to the web, users could not possibly know where it is located. Therefore, searching by tags seems a much better option. In PIM, on the

> other hand, users are very familiar with their own information organization; after all, they stored the information somewhere according to their own subjective needs.... Moreover, they become more familiar with their organization scheme each time they navigate through it to retrieve their files. Therefore, in the great majority of cases, they are able to quickly and efficiently retrieve their own personal information.... Thus, maybe the transfer of the tagging approach from Web 2.0 to PIM systems is questionable. (Bergman et al. 2013, 2008)

The containment model bears several advantages over a tag-based system. First, it provides a visual metaphor: one puts something *in* a folder or notebook. (Brouillon adds the visual metaphor of *going through* a notebook from start to finish, remembering one's thoughts in the order generated through scrolling.) Second, it is hard to browse a *list* of tags—when the number of tags becomes large, one will either get a tag cloud where less frequent items disappear, or a prohibitively large flat list. Folders encapsulate: one can start from the top level of a hierarchy and browse to what is most relevant. In a hierarchical system the organizing concepts are themselves organized, as they are not in a tagging system. Third, folders provide a *sense of control*: users studied by Civan et al. (2008) noted that "folders provided control over their information by helping them to get items out of their way..."

Still, Civan et al. (2008) go on to note their users found one key advantage to tag-based systems over traditional folders: the flexibility and "freedom to keep and re-find information in multiple ways..." This is the key advantage of a model of multiple containment like Brouillon's: it allows *both* tagging's ability to give a single note

multiple classifications, *and* the visual metaphor, encapsulation and sense of control associated with a hierarchical taxonomy. Such a combined approach is not new to Brouillon; it is the approach used by WordPress categories, for example. But it has not yet been implemented for a notetaking or text-editing system.

IMPLEMENTATION

Implementation of Brouillon on the Mac proved more difficult than expected, because the Mac's native Cocoa framework is designed to work with the old project-based paradigm that Brouillon rejects. Cocoa involves a *document architecture*, designed to make it easy to build "document-based applications" by handling all the functions associated with a document, linking an editable text view directly to saving on the filesystem. The flaw in this system is that the single document file is tightly coupled to a single window in which it is edited. The document architecture provides for having multiple windows associated with a single document, but there is no provision at all for having multiple documents in a window—the heart of Brouillon's file concatenation approach.

The Core Data framework, Apple's native data-modelling system, is even worse when it is used in a document-based application: it creates a separate database per document. The idea that one might want a database whose point is *keeping track of the documents* seems to have escaped Apple's engineers.

Instead, Brouillon implements the document architecture only *partially*: the document architecture handles part of the saving of documents, but is not linked to window views. This means that Brouillon is *not* a "document-based application" in Apple's sense; the application requires generalizing Apple's document framework, with the additional work that that implies. A great deal of the design work for Brouillon was spent on generalizing the framework, which allowed less project time for feature development than originally hoped, though the core features described above still exist.

The document framework nevertheless proved useful for its saving functionality.

So likewise did Core Data, for the system requires a database to keep track of the multiple relationships between notes and notebooks. Each notebook is associated with a folder in the filesystem; each note has a "home" notebook corresponding to the folder in which its file is stored. Linking each folder to a notebook is important for intake of new notes (via Dropbox/mobile sync or otherwise): the system knows to incorporate newly created notes because they appear in the folder associated with that notebook. A note's additional notebooks are therefore implemented as a sort of aliases, which the persistent store keeps track of. To open a notebook in Brouillon's current version, the user clicks on the associated folder. When a folder is opened as a notebook for the first time, Core Data's persistent store is told to keep track of that folder as a notebook, identifying which aliased notes will live in that notebook.

Because of the difficulties mentioned above, Brouillon is not yet feature-complete enough for release; it is currently at a proof-of-concept stage. It has been developed enough to implement the four key features listed above. The user can use the pull-down menu or a keyboard shortcut (command-O, the standard shortcut for opening on the Mac) to open a folder of notes. If the specified folder is not registered as a notebook in the Core Data persistent store, the program registers it. It also gives the user the option to create a new, empty folder. These options are all handled using the standard Mac Open dialogue box, provided by the Cocoa framework; see figure 4.

New notes are automatically named with their notebook folder's name followed by a number corresponding to the number of files in that folder plus one (folders aliased

from other notebooks do not count). They cannot yet be renamed by the user. Notes within a notebook are sorted by creation date so the user can view them in order.

Figure 5 illustrates Brouillon in action in its main window, displaying a notebook containing multiple notes including one aliased from a different notebook.

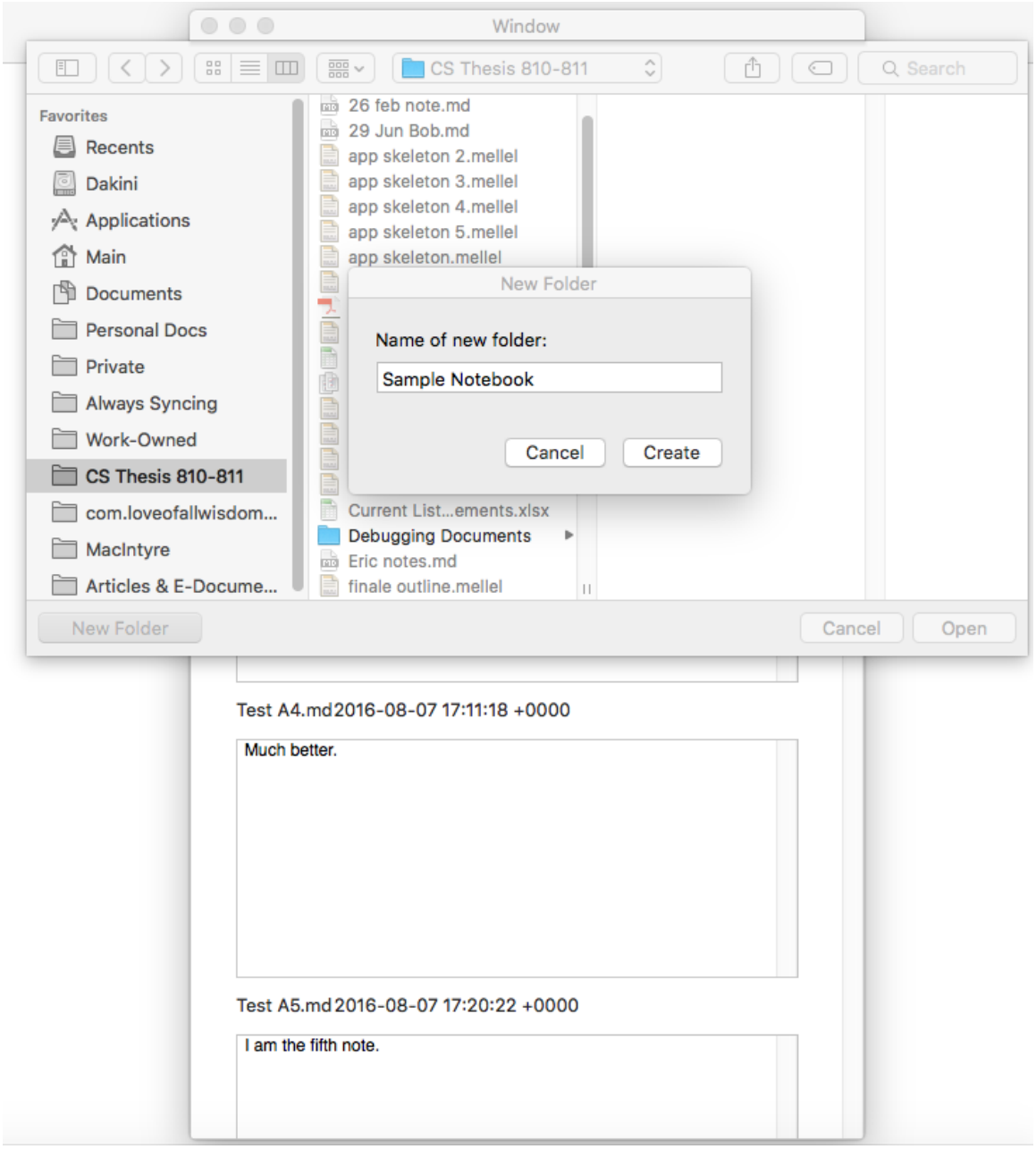

Figure 4: Creating a new notebook folder

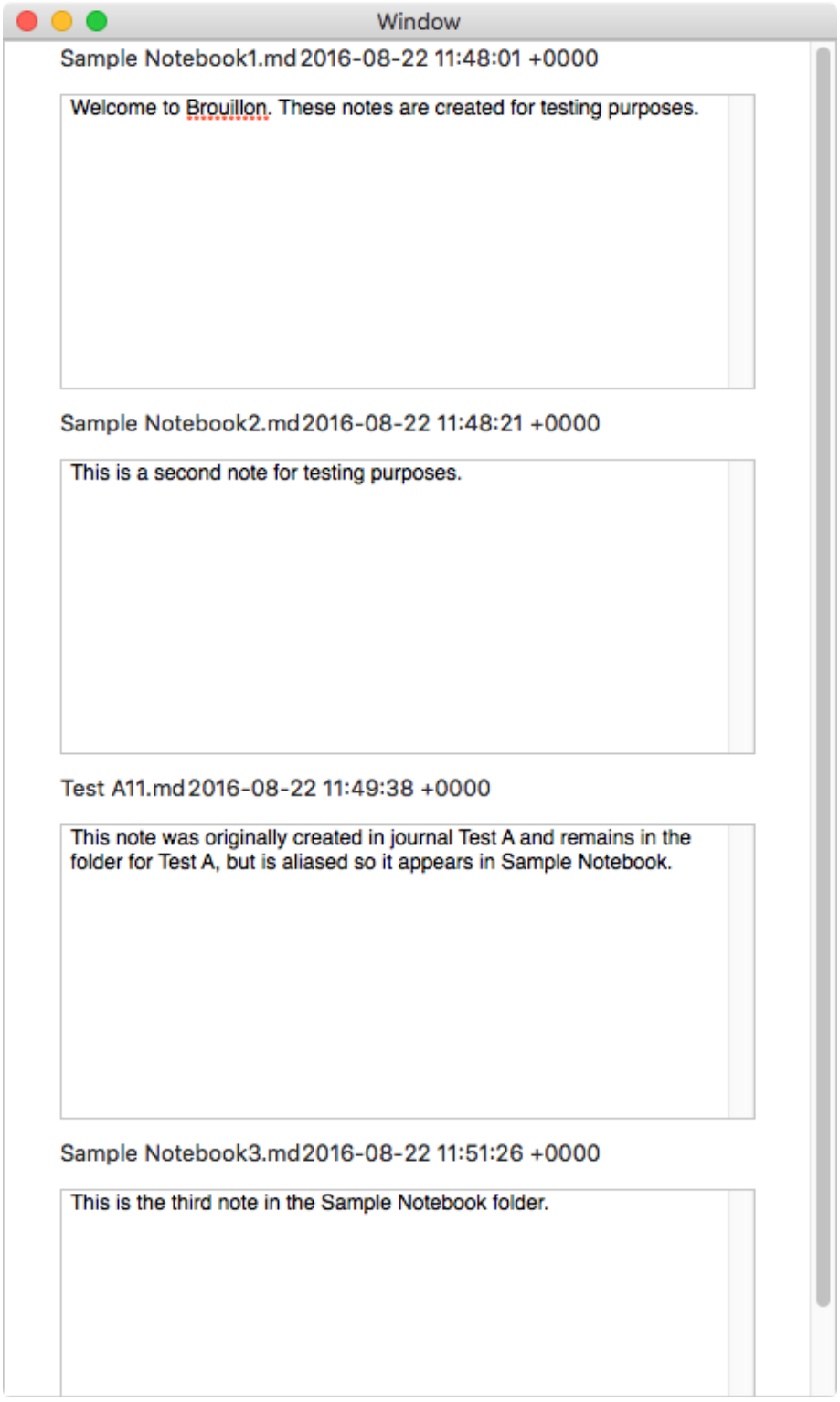

Figure 5: Typical Brouillon window

FUTURE WORK

As noted, Brouillon's functionality is currently limited to the core features described above. It is currently focused on the "bottom-up" aspects of early-stage writing (brainstorming, freewriting) rather than "top-down" aspects such as outlining. Future versions will hopefully contain a number of additional features, many of which will help with top-down writing and the progress from early-stage writing to drafting:

- **Markdown parsing** so that Markdown codes appear in their appropriate HTML WYSIWYG forms (asterisks rendered as italics and so on) within the editor, as is found in Markdown clients like Byword and MultiMarkdown Composer. This should allow the copying and pasting of formatted text to and from other applications with the ability to preserve the formatting. Including native converters to import and export Markdown to .docx, .pages, .mellel, PDF and other formats would be valuable for the process of converting notes to "zero drafts".
- **Live intake**, such that the system will scan folders for new notes even while it is already running.
- **Internal hyperlinking** of notes via the Core Data system, to allow the connections between notes in different projects to be developed in a briefer way than the multi-notebook notes.
- **Snippet or title view** to allow the user to easily scan through a notebook by seeing its first notes.
- **Bulleted lists** (like the present one) to allow the user to organize thoughts within a

note.

In addition to these “marquee” features, some current features need to be more fleshed out. Among these: new notes do not currently get focus when added (the user must click to start typing in them); views for notes within a notebook do not resize with the note, so one must scroll within them; notes cannot be renamed within the program or sorted by an order other than creation date; note dates are displayed in a clumsy and difficult-to-read format.

APPENDIX: SOURCE CODE

This Brouillon code includes the data model as built in the Core Data framework and exported directly from Apple's Xcode IDE, and the source code of the classes written for this project. It does not include classes automatically generated by Xcode and Core Data, the Info.plist file, or the view files (.xib and .storyboard) from Xcode's Interface Builder (which are not easily printable and would leave out important information). Classes are listed in alphabetical order. The notebooks run out of a view controller, so the ViewController class's viewDidLoad() method is effectively the main method.

Figure 6: Brouillon Core Data data model as displayed by Xcode IDE

NotebookManager.swift

```
//
//  NotebookManager.swift
//  Brouillon
//
//  Created by Amod Lele on 2016/08/06.
//  Released as open source under GNU General Public Licence.
//

import Cocoa

class NotebookManager: NSObject
```

```
{
    // A model controller created so the view controller doesn't
have to handle the notebook model.
    // The idea of this class is to manage the *one individual*
Notebook entity for a ViewController. It should be called, in
turn, by *one individual* ViewController that has that Notebook
open, which it should have as a variable. And we re-instantiate
this every time we open a new notebook within the ViewController.

    // MARK: - Variables and constants

    // theViewController needs to be optional so that the two
classes can refer to each other.
    var theViewController: ViewController?
    let documentController =
NSDocumentController.sharedDocumentController()
    let sv = ShortVars()

    // The URL of the folder for this manager's notebook.
    var folderURL: NSURL

    var noteSet: NSMutableSet?
    var openNotes: [NoteDocument] = []

    // var openNotes: NSMutableArray = []


    // Apple docs: "An NSFileManager object lets you examine the
contents of the file system and make changes to it."
    // defaultManager returns "The default NSFileManager object
for the file system." Not clear exactly what that means, but
we'll leave it for now.
    let fileManager = NSFileManager.defaultManager()



    // MARK: - Methods

    init(controller: ViewController, folder: NSURL)
    {
        // Post: Has determined whether there is already a
notebook managed object corresponding to the current folder.
        // If yes, loadExistingNotebook has been passed that
folder; if not, createNewNotebook has been called.

        theViewController = controller
        folderURL = folder
```

```
        // NB: In NSURL's variables absoluteString includes
file:// and path does not.
        let searchedString = folderURL.absoluteString
        var request = NSFetchRequest(entityName: "Notebook")
        request.predicate = NSPredicate(format: "folderURLstring
== %@", searchedString)

        // sv.testNotebook(folderURL)

        super.init()

        do
        {
            let notebooks = try
sv.context.executeFetchRequest(request) as? [Notebook]

            for book in notebooks!
            {
                print(book.folderURLstring!)
            }

            switch(notebooks!.count)
            {
            case 0:
                print("No notebooks found at
"+String(folderURL)+". Create a new one.")
                self.createNewNotebook()

            case 1:
                print("One notebook found. Load it.")
                self.loadExistingNotebook(notebooks![0])

            default:
                print("Oops! More than one notebook with that URL
found! Problem!")
            }

        }
        catch let error as NSError
        {
            print("Could not run fetch request. \(error),
\(error.userInfo)")
        }


    }
```

```
    func addNoteDocument(noteURL:NSURL)
    {
        // Post: NoteDocument object is created with specified
URL and added to the note array and the shared document
controller.
        // Two ways to do the latter: with addDocument and with
openDocumentWithContentsOfURL. The latter worked badly.
        // Note this doesn't interact with the database or
persistent store. RegisterNoteInDatabase is for that.

        let theDocument = NoteDocument(receivedURL: noteURL)
        documentController.addDocument(theDocument)
        openNotes.append(theDocument)
        // print("Added "+noteURL.lastPathComponent!)
    }

    func closeCurrentNotebook()
    {
        // Post: All notes are removed from the openNotes array,
and all notes currently in the document controller have been
removed from it.
        // The latter will need to change if we work with
multiple windows, of course.

        sv.commit()

        for doc in documentController.documents
        {
            // doc.canCloseDocumentWithDelegate(self,
shouldCloseSelector: nil, contextInfo: nil)
            doc.close()
        }

        print("Closed docs in the document controller.")

        // sv.testNotebook()

        showDocumentController()

        openNotes = []
    }


    func countFilesInFolder() -> Int
    {
        // Returns the number of files in the disk folder for
this notebook.
```

```
        var fileCount = Int()

        let dirContents = try?
fileManager.contentsOfDirectoryAtURL(folderURL,

includingPropertiesForKeys: nil,

options: NSDirectoryEnumerationOptions.SkipsHiddenFiles)

        // Beware, the documentation says this is "shallow"
enumeration and might count subdirectories and such.

        if let theCount = dirContents?.count
        {
            fileCount = theCount
        }
        else
        {
            print ("dirContents must have been nil. Let's just
say it's zero.")
            fileCount = 0
        }

        print(String(fileCount)+" files in folder.")

        return fileCount

    }


    func createNewNoteInNotebook()
    {
        // Post: A new, numbered blank note file exists in
homeFolder, recognized by the application as belonging to
theNoteSet for the notebook.
        // SG1: An appropriate note number has been generated.
        // SG2: The note file has been created.
        // SG3: The note has been added to the shared document
controller, and to the managed object context such that it
belongsTo the notebook for the current folder.
        // SG4: The note gets focus (is made the first
responder).

        // SG1: Auto-number (or auto-name) the note.

        // Get the notebook's name from the name of the
directory.
```

```
        let notebookName = folderURL.lastPathComponent
        print("notebookName is "+notebookName!)

        // Get the note number.
        let newNoteNumber = countFilesInFolder() + 1

        // Notebook name + note number = note name, per my usual
format.
        var newNoteName = String()
        newNoteName = notebookName! + String(newNoteNumber) +
".md"
        print("newNoteName is "+newNoteName)

        // SG2: Create the note file.

        // Get the directory path where you will eventually tell
it to store the folder.
        let folderPath = folderURL.path
        print("folderPath is "+folderPath!)

        // Identify a properly formatted path for the notebook
with this name.
        let combinedPath = (folderPath! + "/" + newNoteName)
        print("combinedPath is "+combinedPath)

        // Create a new blank note file at the set path.
        // Note it *wants* spaces rather than percents for
createFileAtPath!
        let blankString = ""
        let blankFile =
blankString.dataUsingEncoding(NSUTF8StringEncoding)!

        if fileManager.createFileAtPath(combinedPath, contents:
blankFile, attributes: nil)
        {
            print("The program created a new file at
"+combinedPath)
        }
        else
        {
            print("The program failed to create a file at
"+combinedPath)
        }

        let newNoteURL = NSURL.fileURLWithPath(combinedPath)
        print("newNoteURL is "+newNoteURL.absoluteString)
```

```
        // Create a new note document object for that file.
        // let newNote = NoteDocument(receivedURL: newNoteURL)

        // SG3: Register the note with our places for keeping
track of notes.

        registerNoteInDatabase(newNoteURL)
        addNoteDocument(newNoteURL)

        // SG4: Give the note focus.

        theViewController!.theCollectionView.reloadData()


    }

    func createNewNotebook()
    {
        // Post1: New notebook entity has been created and added
to the managed context.
        // Post2: Mutable set has been created for the notebook's
notes.
        // Post3: Existing files in disk folder have been added
to mutable set, or a new note has been created if there aren't
any.

        // Post1

        let newNotebook = Notebook(entity: sv.notebookType,
insertIntoManagedObjectContext: sv.context)

        print(folderURL.absoluteString)

        // alternate way of doing it:
        // newNotebook.setValue(fromFolder.absoluteString,
forKey: "folderURLstring")
        newNotebook.folderURLstring = folderURL.absoluteString

        // Post2

        noteSet = newNotebook.mutableSetValueForKey("contains")

        // Post3

        let fileCount = countFilesInFolder()

        if fileCount == 0
```

```
        {
            print ("No files found here!")
            createNewNoteInNotebook()
        }
        else
        {
            // Found files in the folder, intake them.
            intakeFilesFromFolder()
            sv.commit()
        }

        showDocumentController()

        theViewController!.theCollectionView.reloadData()


    }



    func getFilesURLFromFolder() -> [NSURL]?
    {
        // Post: Returns an array of URLs corresponding to the
files in that folder.

        // Taken from the DirectoryLoader class in SlidesMagic.


        // This seems self-explanatory.
        let options: NSDirectoryEnumerationOptions =
            [.SkipsHiddenFiles, .SkipsSubdirectoryDescendants,
.SkipsPackageDescendants]

        // Some online instructions about these, from
monobjc.net:
        // NSURLIsRegularFileKey: "Key for determining whether
the resource is a regular file, as opposed to a directory or a
symbolic link. Returned as an NSNumber object with value 0 or 1."
        // NSURLTypeIdentifierKey: "Key for the resource's
uniform type identifier (UTI), returned as an NSString object."
        // Both properties of an NSURL.
        let resourceValueKeys = [NSURLIsRegularFileKey,
NSURLTypeIdentifierKey]

        // An object to allow you to enumerate a directory,
including "deep enumeration."
```

```
        guard let directoryEnumerator =
fileManager.enumeratorAtURL(folderURL,
includingPropertiesForKeys: resourceValueKeys,

options: options, errorHandler:
            {
                url, error in
                print("`directoryEnumerator` error: \(error).")
                return true
            }
            )
            else
        { return nil }

        // Presumably this gives us an array filled with the
NSURLs for the files in the directory. It has seemed to work so
far so I will let it do its thing.
        var urls: [NSURL] = []
        for case let url as NSURL in directoryEnumerator
        {
            do
            {
                let resourceValues = try
url.resourceValuesForKeys(resourceValueKeys)
                guard let isRegularFileResourceValue =
resourceValues[NSURLIsRegularFileKey] as? NSNumber else {
continue }
                guard isRegularFileResourceValue.boolValue else {
continue }
                urls.append(url)
            }
            catch
            {
                print("Unexpected error occured: \(error).")
            }
        }
        return urls
    }

    func intakeFilesFromFolder()
    {
        // Post1: The document controller is populated with
NoteDocuments from those files in the notebook folder that were
not already retrieved from the persistent store.
        // Post2: the newly added NoteDocuments are added to the
persistent store *related to* the Notebook for this folder.
```

```
        // Earlier note says: You will need some way to check
whether the *files in* the folder are already in the persistent
store, so as to see whether to add them or not. This is probably
done best by relating to checking the date-modified on the files
in the persistent store.


        // Get the NSURLs for all available files in the folder
specified by currentFolderURL.
        let noteURLs = getFilesURLFromFolder()

        for u in noteURLs!
        {
            // Check: do we already have this note open in
openNotes? (Using the file URL as a unique ID/key.)
            // Key is to create a doc object if we *don't*
already have one.

            var alreadyOpen = false

            for o in openNotes
            {
                // Notice if you don't force unwrap here it won't
give you an error, it just won't work!
                if u == o.fileURL!
                {
                    alreadyOpen = true
                    // print("Document at "+u.path!+" was already
open from database.")
                    break
                }
            }

            if (!alreadyOpen)
            {
                // Add an entry for the note to the notebook.
                registerNoteInDatabase(u)

                // Create a NoteDocument for the note and add it
to the shared document controller and the array of open notes.
                addNoteDocument(u)

                // print(u.lastPathComponent!+" was not already
open from database, so we got it from the filesystem, registered
it and added it.")

            }
```

```
            // print("Ran note check on "+u.lastPathComponent!)
        }

        // print("Before we save the mutable set it looks
like:"+String(theNoteSet))
        sv.commit()

        sortOpenNotes()

        // print("After file intake the notebook looks like:")
        // sv.testNotebook()

        // print("After intakeFilesFromFolder, there are
"+String(openNotes.count)+" open notes")
        theViewController!.theCollectionView.reloadData()
    }


    func loadExistingNotebook(theNotebook: Notebook)
    {
        // What we want to happen now is make sure that openNotes
contains both the notes from notebookNoteEntities *and* any new
notes from the disk folder. We would presumably get the former
here and the latter in intakeFilesFromFolder.

        // Post1: Mutable set is returned for this notebook.
        // Post2: Notes contained in the database entry for this
notebook have been added to the document controller.
        // Post3: New notes contained in the filesystem for the
folder have been added to the document controller.

        // Post1
        // Get a mutable set of the notes contained in the passed
notebook. (remember it is a ManagedObject.)
        noteSet = theNotebook.mutableSetValueForKey("contains")
        let altSet = noteSet as! Set<NoteEntity>
        // print("Got our mutable set. Its count is
"+String(noteSet!.count)+" and the altSet version's count
is"+String(altSet.count))

        // Post2

        // Get the note URLs from the mutable set's noteEntities
and get documents attached to them.
        for n in altSet
        {
            let thePath = n.noteURLstring
```

```
            // print("Trying to add a note document for
"+thePath!)
            let theNSURL = NSURL.fileURLWithPath(thePath!)

            addNoteDocument(theNSURL)

        }

        print("The file system has not been touched yet.")

        // Post3

        // when we're *done* loading what's in there, we should
*then* check for file intake.
        intakeFilesFromFolder()

    }

    func noteIsAlreadyInBook(noteURL: NSURL) -> Bool
    {
        // Annoyingly, you have to cast the set in order to
access NoteEntity values.
        let altSet = noteSet as! Set<NoteEntity>

        // Check whether the NoteEntity is already in the
specified notebook.
        var noteInBook = false

        let URLstring = noteURL.path

        for n in altSet
        {
            if n.noteURLstring == URLstring
            {
                noteInBook = true
                print("Found note "+URLstring!+" in book.")
            }
        }

        if (!noteInBook)
        {
            print("Did not find note "+URLstring!+" in book.")
        }

        return noteInBook
    }
```

```
    func registerNoteInDatabase(noteURL: NSURL)
    {
        // Post1: the specified note has been added to the
managed object context if it was not already there.
        // Post2: the note belongsTo the notebook for the current
folder if it did not already.

        if (!noteIsAlreadyInBook(noteURL))
        {
            // Create database entity for the new note.
            let newNoteEntity = NSManagedObject(entity:
sv.noteType,

insertIntoManagedObjectContext: sv.context)

            // Generate a globally unique ID and assign it to the
note's database entity.
            let newGUID = NSUUID().UUIDString
            newNoteEntity.setValue(newGUID, forKey: "uniqueID")

            // Set the entity's noteURLstring value to the one
for the new document object.
            // This could potentially happen earlier.
            let notePath = noteURL.path
            newNoteEntity.setValue(notePath, forKey:
"noteURLstring")

            // Relate note entity to the current notebook in the
database.
            noteSet!.addObject(newNoteEntity)
            print("Added new database entity for file at
"+notePath!)
            // print("Created a new database entry with GUID
"+newGUID)

            sv.commit()
        }
        else
        {
            print("Note was already in book so we did not add
it.")
        }
    }


    func showDocumentController()
```

```
    {
        let theDocs = documentController.documents

        print("Documents in the controller:")
        for doc in theDocs
        {
            print(doc.fileURL!.lastPathComponent!)
        }
    }

    func showOpenNotes()
    {
        print("Documents in openNotes:")
        for doc in openNotes
        {
            print(doc.filename)
        }
    }

    func sortByDate(d1: NoteDocument, d2: NoteDocument) -> Bool
    {
        // added so sortOpenNotes can work
        return (d1.dateCreated) < (d2.dateCreated)
    }


    func sortOpenNotes()
    {
        print ("OpenNotes before sort:")
        showOpenNotes()

        openNotes.sortInPlace(sortByDate)

        print ("OpenNotes after sort:")
        showOpenNotes()
    }


}


// MARK: - Extension to allow date comparison

public func ==(lhs: NSDate, rhs: NSDate) -> Bool {
    return lhs === rhs || lhs.compare(rhs) == .OrderedSame
}
```

```
public func <(lhs: NSDate, rhs: NSDate) -> Bool {
    return lhs.compare(rhs) == .OrderedAscending
}

extension NSDate: Comparable { }
```

NoteDocument.swift

```
//
//  NoteDocument.swift
//  Brouillon
//
//  Created by Amod Lele on 2016/07/14.
//  Released as open source under GNU General Public Licence.
//

import Cocoa

class NoteDocument: NSDocument
{
    var filename: String
    var noteText: String
    var attributes: NSDictionary?
    var dateCreated: NSDate
    var dateString: String?

    init (receivedURL: NSURL)
    {
        // print ("Note initializer was called.")

        // You have to initialize the subclass variables before
you call super.init – and you have to call super.init before you
change any of its variables (like fileURL).
        self.filename = ""
        self.noteText = ""

        // Try to get attributes, most importantly date-created.
        let fileManager = NSFileManager.defaultManager()
        do
        {
            // print("The path is "+(fileURL!.path)!)
            attributes = try
```

```
fileManager.attributesOfItemAtPath(receivedURL.path!)
            // print("The attributes are "+String(attributes))
        }
        catch let error as NSError
        {
            print("The error was: "+String(error))
        }

        if let blurp = attributes?.fileCreationDate()
        {
            dateCreated = blurp
            // print("dateCreated is "+String(dateCreated!))

            // Format the date-created to an appropriate string.
            // For some reason, the date formatter's string
method totally doesn't work, but a simple string conversion does.
            // It's not pretty but it will do for now.
            // let dateFormatter = NSDateFormatter()
            // dateString =
dateFormatter.stringFromDate(dateCreated!)
            dateString = String(dateCreated)
            // print("dateString is "+dateString!)

        }
        else
        {
            print("Did not find the attributes for "+filename)
            dateCreated = NSDate()
        }

        super.init()

        self.fileType = "net.daringfireball.markdown"
        self.fileURL = receivedURL


        if let name = self.fileURL?.lastPathComponent
        {
            filename = name
        }
        else
        {
            filename = "Unnamed File"
        }


        noteText = ""
```

```
        do
        {
            noteText = try NSString(contentsOfURL: self.fileURL!,
encoding: NSUTF8StringEncoding) as String
        }
        catch let error as NSError
        {
            print("Error trying to get note file:"+String(error))
        }
    }

    // MARK: - Document functions

    override class func autosavesInPlace() -> Bool
    {
        // print ("autosavesInPlace ran.")
        return true
    }


    override func dataOfType(typeName: String) throws -> NSData
    {
        // print ("dataOfType ran.")

        var outError: NSError! = NSError(domain: "Migrator",
code: 0, userInfo: nil)
        // dataOfType is the Cocoa document save function.
        // Post: Document is saved to a file specified by the
user.

        // If outError != nil, ensure that you create and set an
appropriate error when returning nil.
        // You can also choose to override
fileWrapperOfType:error:, writeToURL:ofType:error:, or
writeToURL:ofType:forSaveOperation:originalContentsURL:error:
instead.
        outError = NSError(domain: NSOSStatusErrorDomain, code:
unimpErr, userInfo: nil)


        if let value =
self.noteText.dataUsingEncoding(NSUTF8StringEncoding,
allowLossyConversion: false) {
            // Convert noteText to an NSData object and return
that.
            return value
```

```
        }
        print("dataOfType ran.")

        throw outError

    }

    override func readFromData(data: NSData, ofType typeName:
String) throws
    {
        // Currently unused; came free with NSDocument.

        // Insert code here to read your document from the given
data of the specified type. If outError != NULL, ensure that you
create and set an appropriate error when returning false.
        // You can also choose to override -
readFromFileWrapper:ofType:error: or -readFromURL:ofType:error:
instead.
        // If you override either of these, you should also
override -isEntireFileLoaded to return false if the contents are
lazily loaded.
        print("readfromData ran.")
        throw NSError(domain: NSOSStatusErrorDomain, code:
unimpErr, userInfo: nil)

    }


}
```

NoteViewItem.swift

```
//
//  NoteViewItem.swift
//  Brouillon
//
//  Created by Amod Lele on 2016/07/18.
//  Released as open source under GNU General Public Licence.
//

import Cocoa

class NoteViewItem: NSCollectionViewItem
{
```

```
    // MARK: - Variables and constants

    @IBOutlet weak var nameLabel: NSTextField!
    @IBOutlet weak var dateLabel: NSTextField!
    @IBOutlet var theTextView: NSTextView!

    var timer: NSTimer?
    var timerStarted = false
    var docChanged = false

    // Shorthand to avoid continued declarations.
    let sv = ShortVars()

    var theNote: NoteDocument?
    {
        didSet
        {
            // Pre: The NoteViewItem's theNote property is set.
            // Post: This observer has set the content of the
*item's text view*, and label if it has one.

            guard viewLoaded else { return }

            if let theNote = theNote
            {
                // print("Creating noteViewItem for
"+theNote.filename)
                nameLabel.stringValue = theNote.filename
                dateLabel.stringValue = theNote.dateString!
                theTextView.string = theNote.noteText
                theTextView.display()

                // print("theTextView.string set to
"+theTextView.string!+" in NoteViewItem "+String(self))
            }
            else
            {
                nameLabel.stringValue = "Empty note?"
            }
        }

    }

    // MARK: - Functions

    @IBAction func addNoteToNotebook(sender: AnyObject)
    {
```

```
        // This is the key Multi-Notebook Note function.
        // Rename it to Add Note To Other Notebook to be
clear...?

        // Post1: User has selected a notebook other than the
currently open one.
        // Post2: Database now treats this note as part of the
selected notebook.

        // Should be called by the Add Note To Notebook button on
the main menu, because that button should be tied to the Sent
Action of First Responder addNoteToNotebook.

        // Post1:
        // OpenPanel as in a panel from which one opens files,
not a panel that is open.
        let openPanel = NSOpenPanel()
        openPanel.canChooseDirectories  = true
        openPanel.canChooseFiles        = false
        openPanel.canCreateDirectories  = true
        openPanel.showsHiddenFiles      = false

        // Wait until OK is pressed.

openPanel.beginSheetModalForWindow(collectionView.window!) {
(response) -> Void in
            guard response == NSFileHandlingPanelOKButton else
{return}

            // Now OK has been pressed and notebook selected, so
            // Post2:

            let folderURL = openPanel.URL!

            print ("You requested to add
"+self.theNote!.filename+" to: "+folderURL.path!)

            // TODO: Some very messy error handling about to
happen here. Figure out whether you can make folderURLstring
unique to avoid all this garbage.
            // If you can't do that, probably at least outsource
the fetch requests to a separate class and method, seeing as you
use at least the Notebook one in ViewController too?

            // Get the NoteEntity for theNote.
            let noteURLpath = (self.theNote?.fileURL!.path)!
            var noteRequest = NSFetchRequest(entityName:
```

```
"NoteEntity")
            noteRequest.predicate = NSPredicate(format:
"noteURLstring == %@", noteURLpath)

            var notes = [NoteEntity]()
            var note = NoteEntity?()

            do
            {
                notes = try
self.sv.context.executeFetchRequest(noteRequest) as! [NoteEntity]

            }
            catch let error as NSError
            {
                print("Could not run fetch request. \(error),
\(error.userInfo)")
            }

            switch(notes.count)
            {
            case 0:
                print("No note object in the database for
"+String(noteURLpath)+". We could try creating a new one, but not
yet because that could get messy.")
                // createNewNote(noteURL)

            case 1:
                // print("One note object found, as it should
be.")
                note = notes[0]
                // print(String(note!))
            default:
                print("Oops! More than one note object with that
URL found! Problem!")
            }

            // Get the Notebook entity for openPanel.URL.
            let searchedString = folderURL.absoluteString
            var request = NSFetchRequest(entityName: "Notebook")
            request.predicate = NSPredicate(format:
"folderURLstring == %@", searchedString)

            var notebooks = [Notebook]()
            var notebook = Notebook?()

            do
```

```
            {
                notebooks = try
self.sv.context.executeFetchRequest(request) as! [Notebook]

            }
            catch let error as NSError
            {
                print("Could not run fetch request. \(error),
\(error.userInfo)")
            }

            switch(notebooks.count)
            {
            case 0:
                // TODO: Handle this error.
                print("No notebook object in the database for
"+String(folderURL)+". We should try creating a new one, but not
yet because that could get messy.")
                // self.createNewNotebook(folderURL)

            case 1:
                print("One notebook object found, as it should
be.")
                notebook = notebooks[0]
                print(String(notebook!))
            default:
                print("Oops! More than one notebook object with
that URL found! Problem!")
            }

            // Open the contains relationship as a mutable set.
            let noteSet =
notebook!.mutableSetValueForKey("contains")
            // Annoyingly, you have to cast the set in order to
access NoteEntity values.
            let altSet = noteSet as! Set<NoteEntity>

            // Check whether the NoteEntity is already in the
specified notebook.
            // We can't currently use NotebookManager's
noteIsAlreadyInBook code for this because currently that's tied
to *its* notebook, not an alternate one.
            var noteAlreadyInBook = false

            for n in altSet
            {
                if n.noteURLstring == note?.noteURLstring
```

```
                {
                    noteAlreadyInBook = true
                }
            }

            // Add the NoteEntity to that set if it's not already
there.
            if (!noteAlreadyInBook)
            {
                noteSet.addObject(note!)
            }

            self.sv.commit()

        }
    }

    override func viewDidLoad()
    {
        super.viewDidLoad()

        // Hopefully this will set the note's background to
white.
        view.wantsLayer = true
        view.layer?.backgroundColor =
NSColor.whiteColor().CGColor

    }

    override func viewDidDisappear()
    {
        super.viewDidDisappear()

        timer?.invalidate()

        if (docChanged)
        {
            theNote?.saveDocument(self)
        }

    }

    func startTimer()
    {
        self.timer = NSTimer.scheduledTimerWithTimeInterval(
            5.0,
            target: self,
```

```
            selector: #selector(NoteViewItem.timerTick(_:)),
            userInfo: "Hello!",
            repeats: true
        )

        print ("Timer began running on "+(theNote?.filename)!)
        timerStarted = true
    }


    func timerTick(timer:NSTimer)
    {
        // As specified by the selector in startTimer, this is
the function that should run once at the timer's intervals once
the timer is started.

        print("Timer says hello from "+(theNote?.filename)!)

        // Save the document
        theNote?.saveDocument(self)
        sv.commit()
        docChanged = false

    }

    // MARK: - Testing

    func testChangeInNotebook()
    {
        // Test whether this change propagated to the *notebook*
associated with the note entity's home folder.

        // Get the folder URL by deleting the filename part.
        let folderURL =
theNote?.fileURL?.URLByDeletingLastPathComponent

        sv.testNotebook(folderURL!)

    }


}

extension NoteViewItem : NSTextFieldDelegate
{
    // Must implement the delegates as extensions so we can tell
which is which!
```

```
    // At least, this will likely be important if one can select
the label to do Rename Note.

}



extension NoteViewItem : NSTextViewDelegate
{
    func textDidChange(notification: NSNotification)
    {
        // Pre: Should be called when the user has changed the
text in the view.
        // Post: Has changed the text of the note within the
controlling Notebook. *Should* also have saved it on disk, if
we're getting our autosaving working as I hope to.

        theNote?.noteText = theTextView.string!
        // print("The text of "+(theNote?.filename)!+" *should*
now be:"+(theNote?.noteText)!)

        docChanged = true

        if (!timerStarted)
        {
            print("Starting the timer.")
            self.startTimer()
        }
        else
        {
            // print("Timer was already started so didn't need
to.")
        }

        // testChangeInNotebook()

    }


}
```

ShortVars.swift

```
//
//  ShortVars.swift
//  Brouillon
//
//  Created by Amod Lele on 2016/07/24.
//  Released as open source under GNU General Public Licence.
//

import Cocoa

class ShortVars: NSObject
{
    // MARK: - Variables and constants

    let theAppDelegate =
(NSApplication.sharedApplication().delegate as! AppDelegate)
    var context: NSManagedObjectContext
    var noteType: NSEntityDescription
    var notebookType: NSEntityDescription

    // MARK: - Programmatic functions

    override init()
    {
        // Very important to note: you need to instantiate this
from every class that calls it, so it should *only* include
variables that can be recognized when they're re-created! With
reference to the App Delegate's managed object context in
particular. This class exists only as a shorthand to substitute
for lines of code re-declaring all these variables in every class
- but we still *are* re-declaring them with each new
instantiation of the class. *These are not global variables.*
        // Thus the name of this class is "ShortVars" rather than
the original "Context Manager". We could totally use it for
variables and constants that are not Core Data-related and that
would be okay.

        // Initialize the managed object context and Core Data
types.
        context = theAppDelegate.managedObjectContext
        noteType =
NSEntityDescription.entityForName("NoteEntity",
inManagedObjectContext:context)!
        notebookType =
NSEntityDescription.entityForName("Notebook",
```

```
inManagedObjectContext:context)!

        super.init()

    }


    func commit()
    {
        // Commits managed-object changes to the persistent
store.
        do
        {
            try context.save()
            print("Saved to the managed context.")
            // testNotebookInContext()
        }
        catch let error as NSError
        {
            print("Could not save \(error), \(error.userInfo)")
        }
    }

    // MARK: - Test functions

    func testNotebook(folderURL: NSURL)
    {
        // To see what the state of the persistent store looks
like.
        // Pre: Has received a URL for a folder.
        // Post: Has run testNotes on every notebook that
corresponds to that folder URL.


        let searchedString = folderURL.absoluteString
        var request = NSFetchRequest(entityName: "Notebook")
        request.predicate = NSPredicate(format: "folderURLstring
== %@", searchedString)

        print("What the notebook(s) at "+searchedString+" look
like:")

        do
        {
            let notebooks = try
context.executeFetchRequest(request) as? [Notebook]
```

```
            for book in notebooks!
            {
                testNotes(book)
            }
        }
        catch let error as NSError
        {
            print("Could not run fetch request. \(error),
\(error.userInfo)")
        }
    }


    func testNotes(theNotebook: Notebook)
    {
        // To see what the state of the persistent store looks
like.

        // Pre: Has received a Notebook object, which is an
NSManagedObject.
        // Post: Has printed the filename and noteText of all
that Notebook's notes to the console.
        // Get a non-mutable set of the notes contained in the
passed notebook. (remember it is a ManagedObject.)
        let noteSet = theNotebook.valueForKey("contains")
        let altSet = noteSet as! Set<NoteEntity>

        // Get the note URLs from the set's noteEntities.
        // No longer tries to get the note documents because they
will no longer live in the database.
        for n in altSet
        {
            let docURL = n.noteURLstring
            print("File URL: "+docURL!)
        }
    }


}
```

ViewController.swift

```
//
//  ViewController.swift
//  Brouillon
//
//  Created by Amod Lele on 2016/07/14.
//  Released as open source under GNU General Public Licence.
//

import Cocoa
import CoreData

class ViewController: NSViewController
{
    // MARK: - Variables and Constants

    @IBOutlet weak var theCollectionView: NSCollectionView!
    let viewDocumentController =
NSDocumentController.sharedDocumentController()

    // A class created as a convenience to store variables and
constants we don't want to have to keep re-declaring.
    let sv = ShortVars()

    // The notebook manager absolutely needs to be an optional,
not just because you can't init a ViewController (though there is
that), but because we don't even want to call it right away on
viewDidLoad - it should be set when and only when we load a
folder into the notebook's memory. That might change when we have
multiple windows up at once (which would allow each view
controller to be more tightly coupled to its notebook), but right
now we need to change it regularly.
    var nbm: NotebookManager?

    let defaultFolder = "/Users/Main/Documents/Always
Syncing/Academic/Computer Science/CS Thesis 810-811/Test
Documents/Test A"


    // MARK: - Functions called directly

    override func viewDidLoad()
    {
        // This is the main thing that runs when the program
starts up, effectively the main method.
```

```
        super.viewDidLoad()

        configureCollectionView()

        // Set current folder to default.
        let currentFolderURL =
NSURL.fileURLWithPath(defaultFolder)

        // Initialize a notebook manager with that default
folder.
        nbm = NotebookManager(controller: self, folder:
currentFolderURL)

    }

    @IBAction func newNote(sender: AnyObject)
    {
        // print ("Hi! I'm newNote!")
        nbm?.createNewNoteInNotebook()
    }


    @IBAction func openAnotherNotebook(sender: AnyObject)
    {
        // Should be called by the Open Notebook button on the
main menu, because that button should be tied to the Sent Action
of First Responder openAnotherNotebook.

        // OpenPanel as in a panel from which one opens files,
not a panel that is open.
        let openPanel = NSOpenPanel()
        openPanel.canChooseDirectories  = true
        openPanel.canChooseFiles        = false
        openPanel.canCreateDirectories  = true
        openPanel.showsHiddenFiles      = false

        // Wait until OK is pressed.
        openPanel.beginSheetModalForWindow(self.view.window!) {
(response) -> Void in
            guard response == NSFileHandlingPanelOKButton else
{return}

            self.nbm!.closeCurrentNotebook()

            print(String(self.nbm!.openNotes.count)+" notes in
openNotes after close.")
```

```
            self.nbm = NotebookManager(controller: self, folder:
openPanel.URL!)

            print(String(self.nbm!.openNotes.count)+" notes in
openNotes after new NotebookManager instantiated.")


        }
    }


    // MARK: - Supporting functions

    private func configureCollectionView()
    {
        // Configure theCollectionView with a flow layout.

        // Resetting the collectionViewLayout (below) crashes if
you don't nil it first:
        theCollectionView.collectionViewLayout = nil

        let theFlowLayout = NSCollectionViewFlowLayout()

        theFlowLayout.itemSize = NSSize(width: 400.0, height:
200.0)

        print("String of theCollectionView outlet is:
"+String(theCollectionView))
        print("String of collectionViewLayout
is:"+String(theCollectionView.collectionViewLayout))

        theCollectionView.collectionViewLayout = theFlowLayout

        // I'm not *entirely* sure what the layer does, but we
have used layers for selection and stuff before.
        // Leave it in for now.
        view.wantsLayer = true

    }

    func countNoteViewItems() -> Int
    {
        return collectionView(theCollectionView,
numberOfItemsInSection: 0)
    }

    func getCollectionView() -> NSCollectionView
```

```
    {
        return theCollectionView
    }


    // MARK: - NSCollectionViewDataSource

    // ShowFileNames says: The data source somehow seems to
appear as an outlet in Main.storyboard, which we can connect to
the View Controller itself. See A223 for all this stuff.

    func collectionView(collectionView: NSCollectionView,
numberOfItemsInSection section: Int) -> Int
    {
        print("numberOfItemsInSection finds
"+String(nbm!.openNotes.count)+" open notes")

        return nbm!.openNotes.count
        // return viewDocumentController.documents.count
    }

    func collectionView(collectionView: NSCollectionView,
itemForRepresentedObjectAtIndexPath indexPath: NSIndexPath) ->
NSCollectionViewItem
    {
        // Post: Should return a NoteViewItem corresponding to
the index provided in the indexPath.

        // print("Open Notes include:"+String(nbm!.openNotes))

        // Recycle or create a NoteViewItem
        let item =
collectionView.makeItemWithIdentifier("NoteViewItem",
forIndexPath: indexPath)
        guard let theNoteViewItem = item as? NoteViewItem
            else
        {
            print("Whoa, weird error, dude.")
            return item
        }

        // Get the note from the current notebook
        let itemNote = nbm!.openNotes[indexPath.item]
        // let itemNote =
viewDocumentController.documents[indexPath.item]
        // print("Showing note with filename
"+itemNote.filename+" and text: "+itemNote.noteText)
```

```
            theNoteViewItem.theNote = itemNote

            return theNoteViewItem
        }
    }
```

BIBLIOGRAPHY

Association of College & Research Libraries. 2003. “Principles and Strategies for the Reform of Scholarly Communication 1”. http://www.ala.org/acrl/publications/whitepapers/principlesstrategies (accessed 4 January 2016).

Bergman, Ofer, Noa Gradovitch, Judit Bar-Ilan, and Ruth Beyth-Marom. 2013. Folder Versus Tag Preference in Personal Information Management. *Journal of the American Society for Information Science and Technology* 64 (10): 1995–2012.

Bernstein, Mark. 2003. Collage, Composites, Construction. In *Hypertext 03: The Fourteenth ACM Conference on Hypertext and Hypermedia, HT ‘03*, 122–123. New York: Association for Computer Machinery. (Accessed 5 Jan 2016 from ACM Digital Library).

Bolker, Joan. 1998. *Writing Your Dissertation in Fifteen Minutes a Day: A Guide to Starting, Revising, and Finishing Your Doctoral Thesis*. London: Macmillan.

Borgman, Christine L. 2007. *Scholarship in the Digital Age: Information, Infrastructure, and the Internet*. Cambridge, MA: MIT Press.

Civan, Andrea, William Jones, Predrag Klasnja, and Harry Bruce. 2008. Better to Organize Personal Information By Folders or By Tags? The Devil is in the Details. *Proceedings of the American Society for Information Science and Technology* 45 (1): 1–13.

Ede, Lisa. 2003. *Work in Progress: A Guide to Academic Writing and Revising*. Sixth edition. Boston: Bedford/St. Martin’s.

Elbow, Peter. 1973. *Writing Without Teachers*. Oxford: Oxford University Press.

Elmer-De Witt, Philip. 2010. “Big Macs on Campus: In Five Years, Apple Has Switched Places With Dell as the Student Laptop of Choice”. http://fortune.com/2010/08/07/big-macs-on-campus/ (accessed 6 January 2016).

Flower, Linda, and John R. Hayes. 1981a. A Cognitive Process Theory of Writing. *College Composition and Communication* 32 (4): 365–387.

Flower, Linda, and John R. Hayes. 1981b. The Pregnant Pause: An Inquiry Into the Nature of Planning. *Research in the Teaching of English* 15 (3): 229–243.

Force11. 2011. “Force11 Manifesto: Improving Future Research Communication and E-Scholarship”. https://www.force11.org/about/manifesto (accessed 5 Jan 2016).

Galanis, Nikolas, Marc Alier, María Josē Casany, Enric Mayol, and Charles Severance.

2014. TSUGI: A Framework for Building PHP-Based Learning Tools. In *Proceedings, TEEM '14: Second International Conference on Technological Ecosystems for Enhancing Multiculturality*, 409–413. New York: ACM. (Accessed 5 January 2016 from ACM Digital Library).

Garvey, W.D., and B.C. Griffith. 1964. Scientific Information Exchange in Psychology. *Science* 146 (3652): 1657.

Glynn, Shawn M., Bruce K. Britton, K. Denise Muth, and Nukhet Dogan. 1982. Writing and Revising Persuasive Documents: Cognitive Demands. *Journal of Educational Psychology* 74 (4): 557–67.

Gould, Amanda Starling. 2014. Doing Humanities Scholarship Online: A Case Study for the Literary Digital Humanities Writing Course. *Interdisciplinary Humanities* Spring 2014: 23–41.

Gruber, John. 2004. "Markdown". https://daringfireball.net/projects/markdown/ (accessed 5 January 2016).

Hart, Michael. 2015. "Poll: Most College Students Prefer Laptops Over Tablets for School". https://campustechnology.com/articles/2015/09/23/poll-most-college-students-prefer-laptops-over-tablets-for-school.aspx (accessed 5 January 2016).

Moore, Jessie L., Paula Rosinski, Tim Peeples, Stacey Pigg, Martine Courant Rife, Beth Brunk-Chavez, Dundee Lackey, Suzanne Kesler Rumsey, Robyn Tasaka, Paul Curran, and Jeffrey T. Grabill. 2016. Revisualizing Composition: How First-Year Writers Use Composing Technologies. *Computers and Composition* 39: 1–13.

Murray, Donald M. 2003. *Teach Writing as a Process Not Product*. In *Cross-Talk in Comp Theory: A Reader*, edited by Victor Villanueva. Urbana, IL: National Council of Teachers of English.

National Survey on Student Engagement. 2007. *Experiences That Matter: Enhancing Student Learning and Success*. Bloomington, IN: National Survey on Student Engagement.

Pak, Richard, Steven Pautz, and Rebecca Iden. 2007. Information Organization and Retrieval: A Comparison of Taxonomical and Tagging Systems. *Cognitive Technology* 12 (1): 31–44.

Perkins, D.N., and Gavriel Salomon. 1988. Teaching for Transfer. *Educational Leadership* 46 (1): 22–32.

Sommerville, Ian. 2009. *Software Engineering*. Ninth ed. Boston: Addison-Wesley.

CURRICULUM VITAE

Amod Jayant Lele
Born 1976
Educational Technology, Office of Digital Learning and Innovation, Boston University

Prior Degrees

HARVARD UNIVERSITY, Ph.D. 2007: Study of Religion (field: South Asia)
BOSTON UNIVERSITY, M.S. in progress: Computer Science
CORNELL UNIVERSITY, M.S. 2001: Development Sociology major, Government minor
McGILL UNIVERSITY, B.A. (Hons) 1997: Geography/Sociology joint honours

Refereed Publications

"Ethics." In *Oxford Bibliographies in Hinduism*, ed. Alf Hiltebeitel. New York: Oxford University Press, 2015.

"The Metaphysical Basis of Śāntideva's Ethics." *Journal of Buddhist Ethics* 22 (2015): 249–83.

"The Compassionate Gift of Vice: Śāntideva on Gifts, Altruism and Poverty." *Journal of Buddhist Ethics* 20 (2013): 702–34.

"Beyond Enacted Experiences." *Journal of Integral Theory and Practice* 7.2 (2012): 72–87.

"Śāntideva." *The Internet Encyclopedia of Philosophy*, http://www.iep.utm.edu/santideva (2009).

"The Various Forms of Constructive Buddhist Studies." In *Epistemology and Hermeneutics*, ed. Rita DasGupta Sherma and Adarsh Deepak, Contemporary Issues in Constructive Dharma, vol. 2, 131–40. Hampton, VA: Deepak Heritage Books, 2005.

"State Hindutva and Singapore Confucianism as Responses to the Decline of the Welfare State." *Asian Studies Review* 28 (September 2004): 267–82.

Other Publications

The Indian Philosophy Blog. Frequently updated group blog at http://indianphilosophyblog.org (begun 2014), founded with Elisa Freschi and Matthew Dasti.

Love of All Wisdom. Regularly updated individual blog on topics in philosophy and religious studies at http://loveofallwisdom.com (begun 2009).

"The Rejection of Comparative Religion in North America." In *Perspectives on Comparative Religion*, ed. Gauri Chattopadhyaya, Hari Dutt Sharma and Anamika Roy. Sagar, India: Vishwavidyalay Prakashan, 2007.

"Ethical Revaluation in the Thought of Śāntideva." Dissertation, Harvard University, 2007. Advisor: Prof. Parimal Patil. Available at http://loveofallwisdom.com/other-writings/

"Make Yourself Fall from Happiness: Moral Masochism in the Thought of Shantideva." In *Religion and the Senses*, ed. Michael Ostling, CSR Symposium Papers, vol. 2, 57–66. Toronto: Centre for the Study of Religion, 2002.

Conference Presentations

"Intimacy, Integrity and India: Rethinking 'Asian Philosophy'." Dialectical Thinking in the Humanities Conference, Harvard University, Cambridge, MA, USA (May 2016).

"How to Build a Synthesis of Incommensurable Traditions: Reflections on Philosophical Methodology from Alasdair MacIntyre." American Comparative Literature Association, Cambridge, MA, USA (March 2016).

"Promoting Collaborative Learning in the Study of Religion: A Case Study using Google Apps and WordPress Blogs." Teaching Religion Section, American Academy of Religion Annual Meeting, Atlanta, GA, USA (November 2015).

"Adopting ePortfolios on a Large University Campus: Program Assessment and Beyond." ePIC 2015 (ePortfolios, Open Badges, and Identity Conference), Barcelona, Spain (June 2015). Also delivered at Assessment Institute, Indianapolis, IN, USA (October 2015, with Gillian Pierce).

"ePortfolios and Assessment at Boston University." Association for Authentic, Experiential and Evidence-Based Learning Northeast US Regional Conference, Boston, MA, USA (March 2015). (with Gillian Pierce)

"Balancing Cost and Quality in Open Access: The Indian Philosophy Blog," Digital Classics Seminar New England, Waltham, MA, USA (March 2015). (with Matthew Dasti)

"How Outcomes Assessment Led Systemic Change: ePortfolio Assessment at Boston University's College of General Studies," Association for Educational Communications and Technology, Jacksonville, FL, USA (November 2014). (with John Regan)

"The MediaKron Project at Boston University," NERCOMP Special Interest Group, Norwood, MA, USA (May 2013). (with Kacie Cleary)

"Developing a Culture of ePortfolio Use at Boston University," American Association of Colleges and Universities, Atlanta, GA, USA (January 2013). (with Natalie McKnight and John Regan)

"The Connection between Metaphysics and Ethics in the Bodhicaryāvatāra," Society for Asian and Comparative Philosophy, Asilomar, CA, USA (June 2010).

"The Role of External Goods in Benevolence and Compassion: Applying Śāntideva's Thought to the Work of Martha Nussbaum," Comparative Religious Ethics Group, American Academy of Religion Annual Meeting, Washington, DC, USA (November 2006).

"An Ethical Approach to the Bodhicaryāvatāra's Literary Form," Buddhist Studies Graduate Student Conference, Princeton University, USA (April 2005).

"The Rejection of Comparative Religion in North America," National Seminar on Comparative Religion, University of Allahabad, India (February 2005).

"The Various Forms of Constructive Buddhist Studies," Dharma Association of North America, American Academy of Religion Annual Meeting, San Antonio, TX, USA (November 2004).

"Constructive Buddhist Studies: What It Is and Why It Matters", Buddhist Studies Graduate Student Conference, Harvard University, USA (April 2004).

"Make Yourself Fall from Happiness: Moral Masochism in the Thought of Shantideva," Study of Religion Conference, University of Toronto, Canada (June 2001).

"Legitimation through Tradition: A Comparative Study of Hindutva and Singapore Confucianism," Confucian Traditions Group, American Academy of Religion Annual Meeting, Nashville, TN, USA (November 2000).

"Global Commodification of Spirituality," Sociology at the Turn of the Century, University of Toronto, Canada (April 1999).

Teaching Experience

LECTURER, Philosophy, Boston University, Boston, MA
Spring 2014: *Indian Philosophy*

VISITING ASSISTANT PROFESSOR, Religious Studies, Stonehill College, Easton, MA
Fall 2008 through Spring 2010: *Encounters in the History of Religions*

Spring 2009: *Hindu Tradition*
Fall 2008 and Spring 2009: *The Good Life in Cross-Cultural Perspective*

VISITING ASSISTANT PROFESSOR, Religion, Colorado College, Colorado Springs, CO
Spring 2008: *Sexuality in South Asian Traditions; Introduction to Hindu Tradition; Hindu-Muslim Relations*
Fall 2007: *Introduction to Hindu Tradition; The Good Life in Cross-Cultural Perspective; Islam*

TUTORIAL INSTRUCTOR, Study of Religion, Harvard University, Cambridge, MA
The Good Life: Classical Views in Cross-Cultural Perspective (fall 2002)
Sexuality in Indian Religions (fall 2001)

TEACHING FELLOW, Harvard Divinity School, Harvard Core Curriculum and Harvard Extension School, Harvard University, Cambridge, MA
"If There Is No God, All Is Permitted": Theism and Moral Reasoning (Prof. Jay Harris, spring 2006)
Justice (Prof. Michael Sandel, fall 2003 and fall 2005)
Understanding Islam and Contemporary Muslim Societies (Prof. Ali Asani, spring 2004)
Sources of Indian Civilization (Prof. Diana Eck, spring 2003)
Indian Religions Through Their Narrative Literatures (Prof. Anne Monius, spring 2003)
World Religions: Diversity and Dialogue (Prof. Diana Eck, fall 2002)
Issues in Buddhist Philosophy (Prof. Janet Gyatso, spring 2002)

TEACHING ASSISTANT, Sociology, Cornell University, Ithaca, NY
Introduction to Sociology (Prof. Michael Macy, fall 1999; Prof. Szonja Szelenyi, spring 2000)

TUTOR, The Academic Approach: Tutored students in English, writing and mathematics, at levels from middle school to freshman college, one-on-one and in classroom. (2010–2012)

Fellowships and Awards (selected)
Graduate Society Dissertation Completion Fellowship, Harvard University (2006–2007)
Language Training Fellowship, Shastri Indo-Canadian Institute (2005)

Service Experience
Information Services & Technology, Boston University (2011–present): Full-time senior educational technologist.

New Books Network (2011–present): Conducting podcast interviews with authors of new scholarly books in Buddhist studies.

Journal of Buddhist Ethics (2010–present), *Journal of Value Inquiry* (2016–present): Reviewing articles for publication.

Society for Asian and Comparative Philosophy (2010–2012): Managed and updated website.

Department of Information Technology, Stonehill College (2010): Part-time educational technologist.

Wisdom Publications, Somerville, MA (2009): Reviewed books on contemporary Buddhism in French for potential translation and publication.

Buddhist Studies Graduate Student Conference, Harvard University (2004): Secured funding for conference; participated in review/selection of abstracts; introduced presentations.

Workshop in Cross-Cultural Philosophy, Harvard University (2001–2004): Founder and coordinator