**IAT 355**: Developing Design Tools

Project Phase 3

April 19, 2012

- Lanz Singbeil

- Ethan Johanson

- Nathan Tsang

**Just Write**

By Writing ++

**Table of contents**

3 Project description

4 - 5 Object-oriented design and system architecture

6 Detailed Class Diagram

7 (Review) Prototype Design and Implementation: Phase 2

8 (Review) Usability Assessment: Phase 2

9 – 12 Prototype 3 design and assessment

13 Conclusion

For Use Cases and associated UI Sketches, please refer to the separate Use Case document

**What is Just Write?**

Just Write is a tool for creative writers. It is geared towards writers who want to focus on their craft without worrying about dealing with a heavy-duty editor like Microsoft Word. It features a text editor with support for multiple windows to organize their work the way the writer wants. And if they want to work on their outline, there is a window for that too. It uses a timeline to organize the significant events in the story. As may already be quite clear, the features are all custom-tailored for the creation of stories.

**Features:**

**Writing:** Just write in the easy to use text editor.

**Outline Management:** Create a full outline of your story with characters, locations, notes and events. Position them on a timeline to organize the who, what, where, when, and why of your story.

**In-Line References:** While you write, you have the option to create references to different parts of your outline.

**Dynamic Work-Flow:** Go back and forth between editing your outline and working on your story. Or work on sub-sections of your story in different windows. Any changes you make in one view will update in the others.

**Branching:** Create alternatives at any point in a paragraph and manage which branch to show.

**XML Data Structure:** All your files are saved as XML and can be easily shared with other applications. This also makes it very easy to use version control to collaborate with other people.

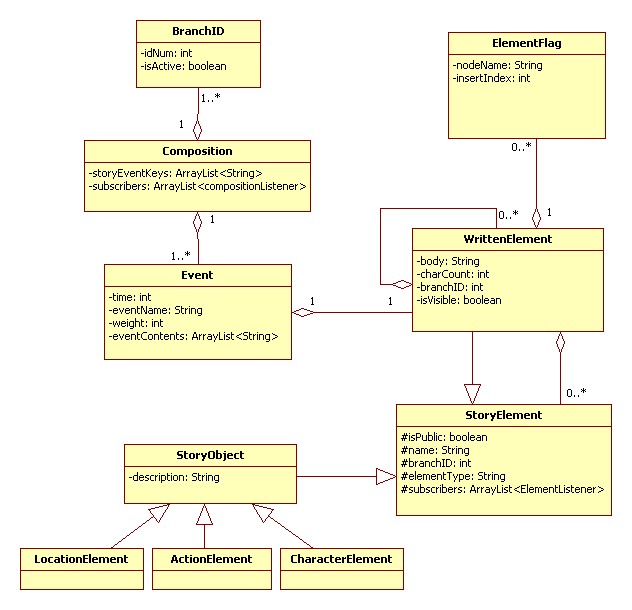
**Changes from Phase 2:**

In phase 2 we decided that we would merge the two alternative ideas we had suggested in Phase 1. Those were the timeline/outline editor and the recursive text editor. We built our underlying model structure and made some initial headway with some simple prototypes which will be discussed later in this report.

In this final phase, we’ve built in our proposed XML data structure that supports saving/loading and set up communication between our two windows. Changes made to the timeline will adjust the organization of the composition. The composition window supports a node hierarchy to organize alternatives as well as features for showing/hiding and simply writing content.

In the conclusion we will discuss features that we did not quite get to implement.

**Object-oriented design and system architecture**

****

**Fig 1.1: A simplified view of the class diagram containing only attributes. For a more complex view, refer to the end of this section.**

Our system follows the Model-View-Controller convention, with Fig 1.1 and 1.2 showcasing the Model aspect of the system's design. Visualized in the document is the structure of how we manage data.

At the lowest level is the concept of a StoryNode, located in the lower right of the diagram, which contains some basic attributes and operations that all of its subclasses will contain.

Extending the StoryNode class are the StoryElement and WrittenNode classes. The former adds one extra attribute complete with get and set operations, and provides some new operations to affect private variables it inherits from StoryNode.

The WrittenNode is one of our most complex classes, which adds a wide array of attributes and functionality to the basic StoryNode class. These include a branch identifier, and visibility toggle (these are used for managing alternatives in our system). Other added attributes include a body text, a list of subscribers (used to interface with our event controller component), and a collection of both other StoryNodes (or extensions) and a collection of another class, the NodeFlag. These collections allow the class to manage a variety of contents within the piece of writing, be they references to characters or locations, or sub-written components that act as branched alternatives. All the operations added to the WrittenElement extension have to do with managing these new attributes, whether through simple gets and sets on elements like branchID and bodyText, or recursive search and retrieval or addition management for the newly available child arrays.

The ElementFlag class is comparatively simple when presented alongside the WrittenElement. It contains a simple name and index value. The name is a reference to the stored name of a sub-object within the system to aid in retrieval of objects from hashmaps, and the index value is a reference for at what character count within a written node a sub component is located.

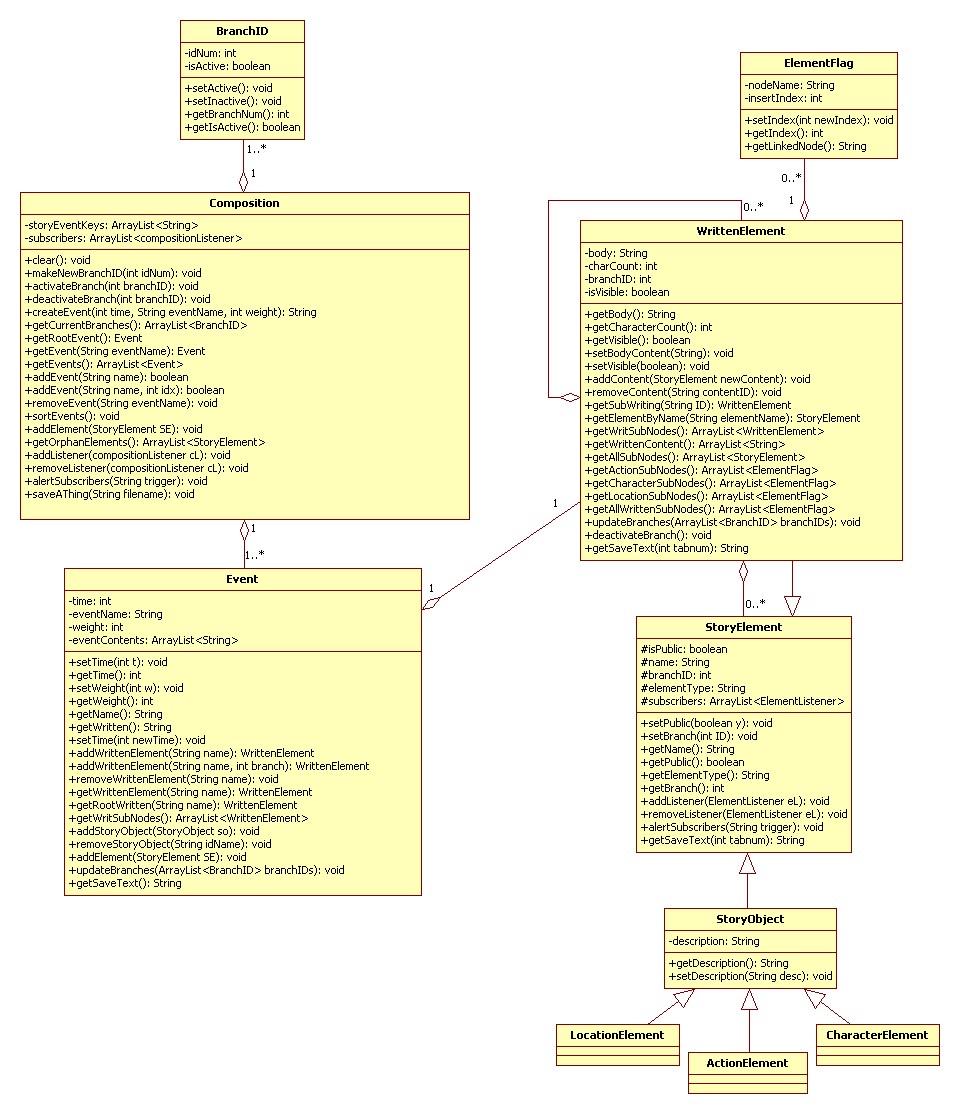
Another class that contains StoryElement is the Event class. This class has a time attribute as well as a name, but its key attributes are its collections of sub-components in the form of StoryElements. All event classes will contain at least one of these in the form of a root WrittenElement. The Event contains a variety of tools to manage addition and removal of these children from it, as well as providing references to specific types of these child objects, attributes of them as a group, and of course individual access to them. It also contains operations to insert elements into its root WrittenElement at a specific point.

The composition class provides many of the same functionalities of the event class, but as the top level parent it also has a variety of differences. As a single composition, it does not maintain a time attribute as events do. It also contains a collection of Events in addition to a collection of StoryElements, and provides functional operations to gather information from, manage, and add/remove both of these types of children. The composition class is also the management class for controlling branches (alternatives) within the system. This is accomplished through iterative actions upon the subcomponents in combination with a collection of branchID items.

The branchID item is the final structural class in our model, and its functionality is quite simple. The class contains a branchID and a boolean state of whether the branch is active or not. This combined with the composition class provides an easily accessible list of all of the branch alternatives and whether they are activated or not for display in the view.

The View component of our system is able to interface with our model through Controllers via EventListeners and subscribers. This is also designed to allow multiple window/view instances to know about changes in state to the model and change accordingly.

**(The next page has a more complete class diagram)**

****

**Fig 1.2: an expanded view containing operations as well as attributes**

**Prototype Design and Implementation: Phase 2**

In this phase we built 3 prototypes:

**Prototype 1: Domain Objects: Eclipse IDE: Java**

**Addressed Features: *In-Line References, Branching***

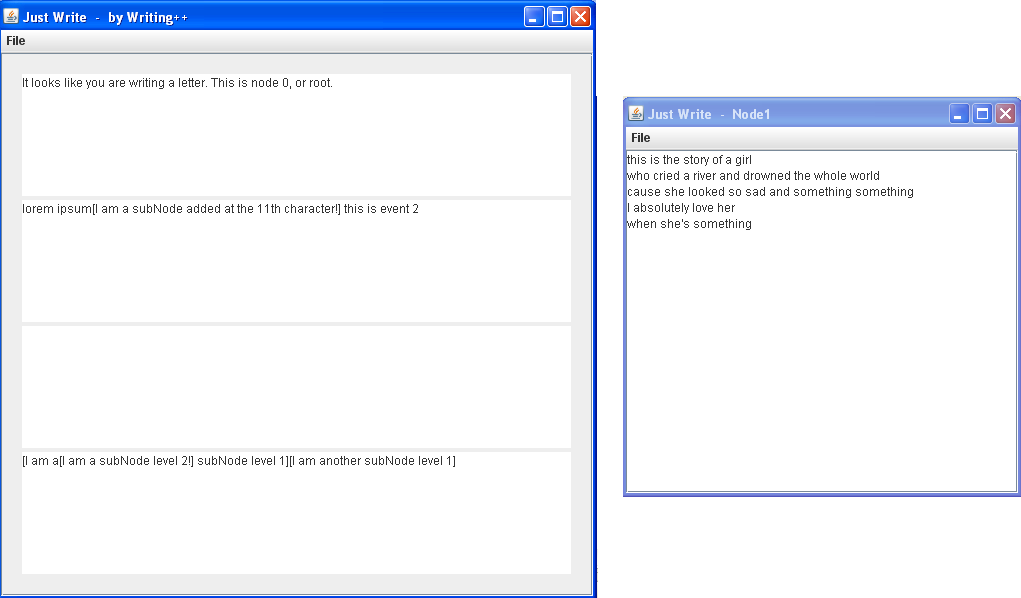
This first prototype is a structural. We built our underlying data structure in Java in order to test whether we were underestimating how complex the system needed to be.

**Prototype 2: Writing and Window Management: Eclipse IDE: Java**

**Addressed Features: *Dynamic Work-Flow, Writing***

This prototype was a way of testing user interaction with a multi-window interface. It contains a Main Composition Window that manages the display of other windows. It also supports loading text files and basic editing.

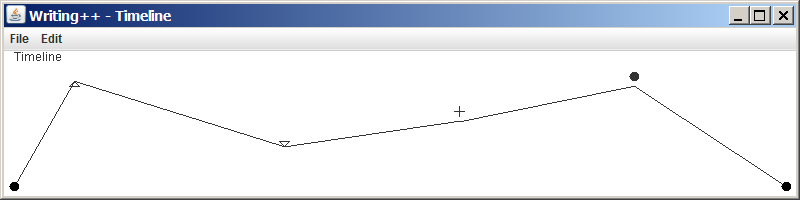
This prototype allowed us to explore the capabilities and limitations of the Java Swing libraries for handling window management and text editing. A particular emphasis was put on developing an event structure for notifying components of different kinds of changes.



**Prototype 3: Timeline Mockup: Eclipse IDE: Java**

**Addressed Features: *Outline Management***

This prototype allowed us to become familiar with the graphics and graphics2D API within Java Swing and Java.AWT. It is also a test-bed for designing the layout for the timeline.



**Usability Assessment: Phase 2**

Prototype 2: **Writing and Window Management** was the only prototype that featured significant user interaction. The following is our initial assessment:

* Undo: The system needs to fully support undo operations. JTextArea does not support undo on its own and it is very frustrating for writers to not have it. Initial research suggests that we will have to create an undo-manager object that includes behavior for accessing the system clipboard and the underlying document model that JTextArea contains.
* Pages: Most document editors provide support for separating the document into different pages. We may want to consider some way to integrate that into our current system. The alternative is to use multiple windows to create new pages as long as this is easy and intuitive for the user to do. More testing needed.
* Confirmation: Although asking whether the user really wants to close a window can be a very useful feature, it can also get annoying when you are opening and closing many node windows. Perhaps the system should know if you’ve already saved the document, in that case it won’t prompt you to save before it closes. This is a common convention. Alternatively we could ask the user to specify a choice and have it always preform that action from that point on.

**Prototype Design and Evaluation: Phase 3**

The features we have implemented can be divided into 2 sections: the timeline window and the composition window.

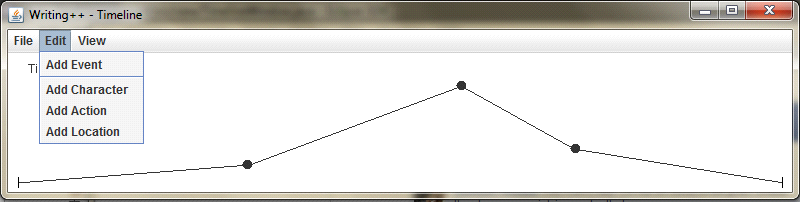
The Timeline Window supports adding events, moving them, deleting them, and adding different story elements to selected events. Story elements include characters, actions and locations.

The Composition Window supports saving and loading an XML representation of the entire model and updating the view to display the proper hierarchy. It also allows the user to manage what branches are active and input new content.

**Heuristic Evaluation:**

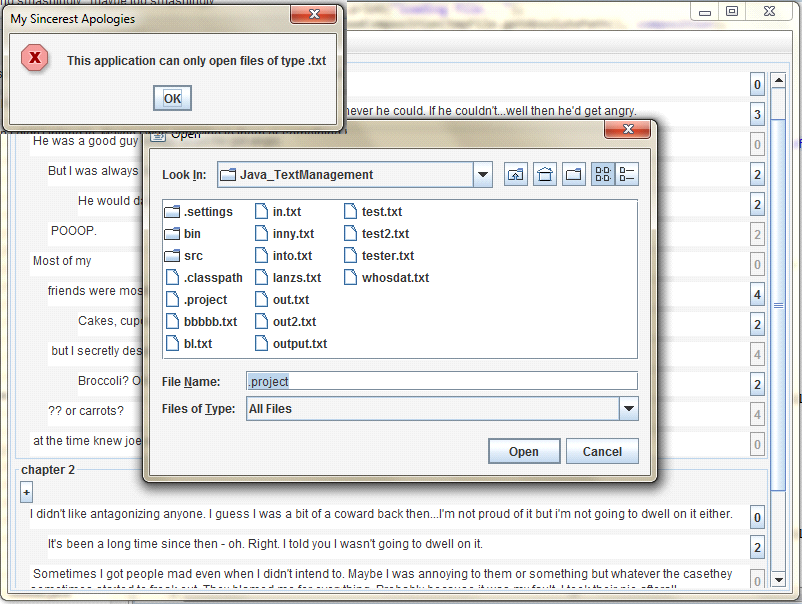
-Just Write supports the Visibility of System Status heuristic through the use of greyed out options in the Content Window display, as well as the utilization of updates between the Timeline Window and Content Window that dynamically rearrange the order of content as events move on the timeline.

-Simple and natural dialogs are aimed for in all of our dialog windows, reducing needless complexity and separating tasks into distinct actions rather than cramming them all into the same dialog window.



*Fig 1. Heirarchy in the Timeline Menus*

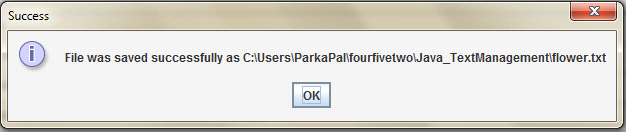
-Aiming to speak the users language we attempt to avoid complex terminology and technical language in dialog windows and functions. This involved changing some early versions of dialogs that were not completely thought through to simpler language.



*Fig 2. Clear and simple communication language*

-Minimize user memory load was one of our weaker heuristics, although we strived to create consistent language in the use of event, character, location, and actions in order to reduce user confusion. This also ties into the next heuristic evaluation of consistency. With more time we would ideally have utilized more icons to further reduce memory load and increase consistency.

-Providing feedback is done through visibly seeing the results of actions on the timeline affect the composition window, as well as through actions such as saving which announce success and the saved filename.



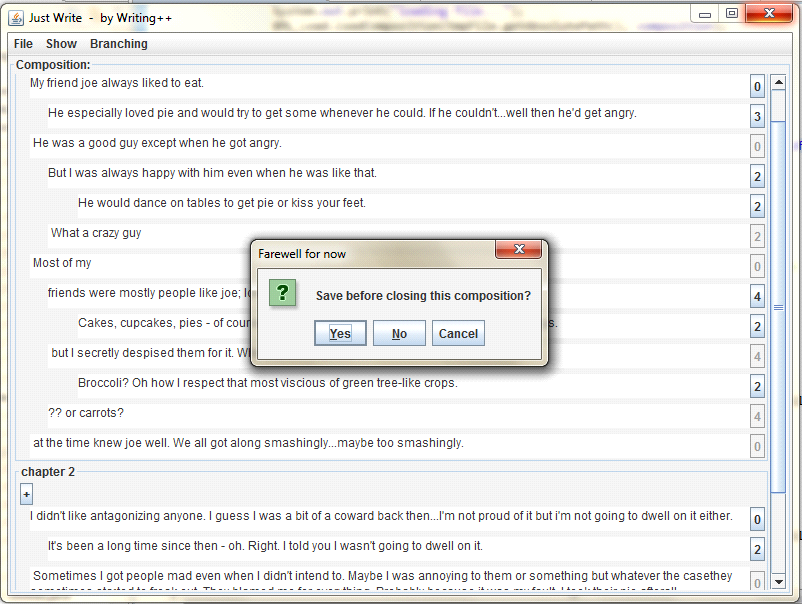
*Fig3. Save Dialog feedback*

-In terms of providing User Control, we provide escape (cancel) keys for all choice based dialogs and have clear heirarchies of menus so the users feel a sort of familiarity and mastery of the system.

*See figure2 for an example of cancellation options in the save menu*

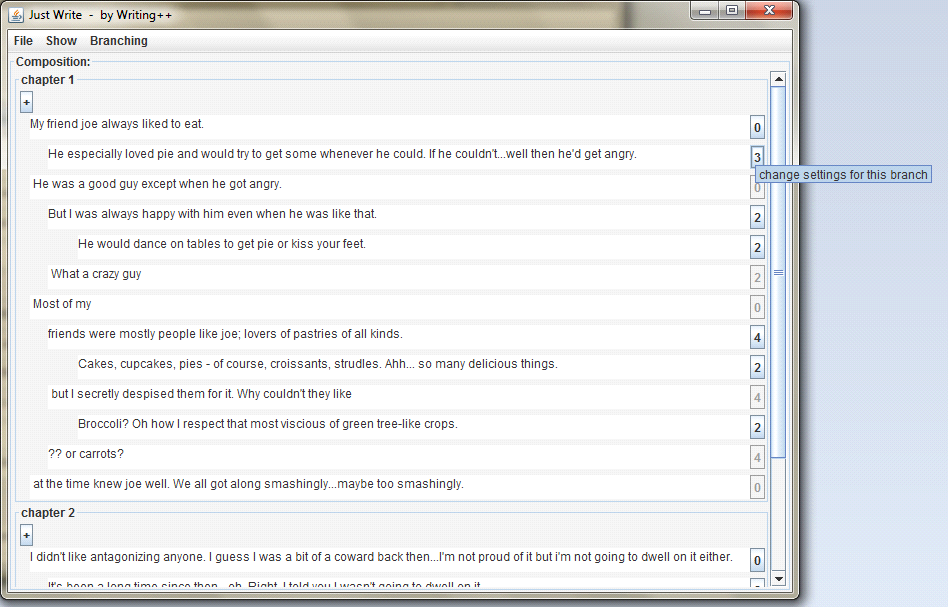
-Providing Shortcuts was an area we ended up somewhat lacking unfortunately, largely due to focus on creating single action options that were functional before focusing on providing alternative methods. In the future, we would aim to provide more of these sort of options

-Preventing Errors was a major heuristic we all recognized as important, so we allowed for users to delete Events they mistakenly added to the timeline as well as providing a window prompting for save when the window was closed. While we would ideally have a universal undo command, our code proficiency with Java Swing and event handling did not allow us to be able to keep track of that information in the timespan we had for this project



*Fig 4. Save Dialog Prompt*

Help and Documentation was another area we did not really go into depth on due to time constraints and coding difficulties, but the low amount of information necessary for our project caused us to place this heuristic at a lower priority. Regardless, we did include tooltips on some action items.



*Fig 5. Tooltips utilized*

In the process of evaluating our system, we developed a sense of what information we needed from the system to create artifacts with our tool. Through creating dummy stories, we learned what functions were wanted to use most often and what functions were lower priorities. We found that managing events via the timeline was a useful visual representation of the writing and that adding elements through that tool was a cohesive experience. While in the composition writing, it was found that adding elements was a lower priority and that managing branching was a higher importance function, giving that aspect prominence on the right of the window as a result.

Unfortunately, many of our heuristic problems that we identified as higher priority such as the need for a universal undo and the use of alternative shortcuts for accomplishing tasks and goals were limited by our ability to work with Java and particularly Swing to manage such complex tasks and were unable to fully implement these elements. We did update the look and feel of various elements to provide greater heirarchy and associations where our user tests led us to realize there was any sense of confusion with the system whenever our coding skills were able to accomplish the goal. Another medium level priority we were ultimately unable to address was the use of greater iconography, mostly in part due to time constraints and choosing to focus on attaining a functional set of coded actions.

**Conclusion:**

At the end of phase 2 we identified the most essential features of our system were that it support dynamic work-flow, outline management, and writing

We have been able to successfully combine the different functions of our project into a working, cohesive prototype but each feature still needs considerable work before we will be completely satisfied with the result.

Much of the trouble was integration with Java Swing to dynamically add and remove indentation when needed. We also needed to be able to support writing to the model and having other story elements that refer to the same content update as needed

In the future we plan on exploring the flexibility of our XML data structure to allow us to create a web-version of our application. We also want continue to test different ways that the multi-view system can support writing and outline building simultaneously.