1. Introduction

This document will serve as a guide through Suffice and to our diagrams as a whole. Various important features, constraints, and concerns are addressed here. Please review this document as your first step into Suffice, and use it as an ongoing reference alongside the accompanying diagrams.

2. A Tour of Suffice

2.1. Controller: Where the action gets started...and persisted

The controller is a grouping of classes which provide application initialization, model level initialization, persistence, and other processing functionality.

2.1.1. Bootstrapping: Kick the tires and light the fires

Bootstrapping, or application initialization is performed by the controller.SufficeBootstrapper class. This main method in this class is called, and after performing any initialization logic, launches the view, and returns. The current design calls for no "initialization logic" per say, but could be easily extended to allow for command line parameter processing (for example, "-o [filename]", to open a file at launch)

2.1.2. Workbooks: Read, initialize, persist.

The model.Workbook class is the highest level model unit, but details about that later. The controller.SufficeController class manages model , Workbook initialization, serialization, and *if you're feeling lucky,* copy and paste or printing.

The decision to seperate these concerns out of the model was based on the rationale that they really weren't central to the model's function as a domain specific database for spreadsheets... oops, I've said too much --

2.2. Model: The heart of Suffice

The model was designed as a "domain specific database" -- in layman's terms, the model is designed as an abstract representation and simulation of a spreadsheet. Think of it as a "smart model".

2.2.1. Smart model?!?

Rather than storing the 'state' of a spreadsheet and relying on a controller to implement the 'business logic' of a spreadsheet, we baked the business logic right into the representation.. This makes sense from a separation of concerns standpoint in that all the business logic for the model is by definition, highly coupled to that model.

This also has the added benefit of putting an kind API right into the model...

2.2.2. The Workbook as a database

So we know that models are loaded by the controller, and that models come pre-loaded with all shiny the business logic necessary to actually use it like a spreadsheet... So we decided to use it like one...

Basically the idea is to instantiate a model (either empty, or loading the initial state from a file) and then visualize and manipulate it using the view, or process it with the controller. The model doesn't really care; it doesn't (shouldn't!) have to. This "application inspecific"design allows you to use the model for many different tasks. Couple this with the controller.SufficeBootstrapper functionality, and you've got the makings for UI-less printing, and all kinds of other fun, interesting, and depressingly unlikely features.

2.3. View: How users interact with the program

The view represents how the user will primarily interact with the model. It receives events from the model through the Swing.TableModel interface, and interacts with the controller. SufficeController to serialize the model.

The view directly manipulates the model for normal editing, as it seems illogical to place this functionality anywhere else. As with most user interfaces, it'll be mostly event driven, so a solid understanding of Swing's (a few resources are mentioned later) architecture is absolutely critical.

Simplicity is the name of the game with the view; just try to respect the separation of concerns laid out in the Architectural Concern Breakdown diagram.

3. About the Design

3.1. Libraries used

Some important decisions about resources and other concerns must be addressed here. In Suffice, two components libraries are used. The identification and understanding of these components are key to a successful implementation. The components used include the Jeks library and a set of Swing components including JFrame, JPanel, and JTable.

3.1.1. JeksParser (and 'friends')

JeksParser is a Java expression language parsing library that's been built into a Swing based spreadsheet application known as Jeks.. Somewhere along the way JeksParser was coupled to Jeks' spreadsheet utility, so you actually need both to compile JeksParserDon't fret too much, as we've included a JAR distribution of the Jeks "master" package; you should simplybe able to include it in your class path, and then reference the classes contained within.

The Jeks Library integrates into Suffice entirely in the model package. This is clearest in the Sheet class which contains an JeksParser.Expressions Parser and the JeksParser. ExpressionParameter interface which is implemented. The other attachment that the model has to Jeks is the JeksParser.CompiledExpression which stores the compiled representation of an expression for the cell. There is some documentation of the Jeks library available on their website:

http://www.eteks.com/jeks/en/

3.1.2. Swing

Swing components make a large portion of the View module. It is important to note that JTable will be used to represent the "grid" of the spreadsheet. Please refer to the Swing documentation

since the JTable needs more developed row/column functionality in order to function properly as a spreadsheet.

The following swing resources might be of interest:

A Visual Index to Swing Components

http://java.sun.com/docs/books/tutorial/uiswing/components/components.html

JTable and TableModel: Row Headers

http://www.chka.de/swing/table/row-headers/

3.2. Critical Concerns: What we think you really need to get right

3.2.1. The Model

As noted before, the model is really the core of the application from our standpoint. Getting the model right will make creating the controller and the view significantly easier.

Please take the time to understand the model's design; it's critical.

3.2.2. Circularity checking

If you think about a spreadsheet that supports cell references, it's easy to come up with a scenario where a cyclic dependency exists between cells. If you think about what would happen when you tried to evaluate the cells without noticing the cycle, it should be relatively clear that the behavior is unacceptable. Avoid it.

3.3. Known Issues: Where we think we've gone really wrong

3.3.1. Circularity checking

It's relatively simple to implement a check for cyclic dependency, but the integration with the JeksParser model has made this difficult. Make sure you flesh out exactly how cyclic dependency

is going to be handled before you even start on the model, or else you might find yourself with some nasty complications later.

An interesting note: Jeks itself implements a solid system for circularity checking, but leveraging it would force us to roll even more Jeks-specific, inter-module dependencies into the model; evaluate it's benefits and proceed with caution.

3.3.2. The View

Unfortunately we weren't able to flesh out the view as much as we probably should have.. We've provided a concern breakdown and a user interface mockup, but as you've probably already noticed, the design is a little sparse in this area.

First, Swing is often a bit of a beast to work with... JTable takes you pretty far in terms of the table, but it does have some serious drawbacks, including it's relative inability to render row-headers, or manipulate row order. We've included links in the Swing section which should get you started.

Second, we haven't come up with a solid scheme for managing inter-cell dependency and re-evaluation. This is largely a product of the 'rough' integration with Jeks library, and should be investigated and specified before you try to develop the view itself.