

Creating a Fun Program that is Simple and Easy to Use



William J. Herrmann
Computer Science
Wittenberg University

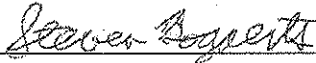
A thesis submitted for the degree of
B.A., Computer Science

December 2011

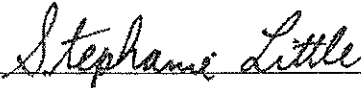
This thesis entitled:
Creating a Fun Program that is Simple and Easy to Use
written by William J. Herrmann
has been approved for the Department of Computer Science



Prof. Kyle Burke



Prof. Steven Bogaerts



Prof. Stephanie Little

December 8, 2011

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Dedication

To my family for their unending support of my endeavors and to the Wittenberg
Role-playing Guild for their overflowing enthusiasm.

Acknowledgements

I would like to thank first and foremost Dr. Kyle Burke, my advisor who worked closely with me on this project and gave me a great deal of encouragement at every step of the way. I'd also like to thank Jake Hsu, Chris McDaniel, and Chris Stockhaus who provided feedback about the program's user interface at various points in the design process and taught me a great deal about how valuable user-centered design is. I also wish to thank Patrick Copeland for providing me with a thesis template for L^AT_EX, which was used to make this thesis look as professional as it does. Finally, I'd like to give special thanks to Pinnacle Entertainment Group for creating *Savage Worlds* and inspiring me to create the program I have made.

Abstract

I have created a program to assist in making characters for the *Savage Worlds* tabletop role-playing game published by Pinnacle Entertainment Group. This program was written in Java and uses a SQLite database to store character options. Principles of user-centered design were followed to make the program intuitive and easy to use and feedback from potential users was gathered. Programming design patterns were used to make it easier for me to implement and maintain the program. Ultimately, this program has been a culmination of my learning here at Wittenberg University and has been an opportunity for me to independently expand my knowledge of Computer Science.



Contents

1	Introduction	1
1.1	What is a Tabletop Role-playing Game?	1
1.2	Purpose of this Program	1
1.2.1	Example of Use	2
1.2.2	Program Scope	2
1.3	User-Centered Design	2
1.4	Programming Language	3
1.5	Database Framework	3
2	Program Overview	4
2.1	Load Database Window	4
2.2	Main Window	5
2.3	Summary Area	5
2.4	Race Tab	6
2.5	Traits Tab	7
2.6	Hindrances Tab	8
2.7	Edges Tab	9
2.8	Gear Tab	9
2.9	Background Tab	10
2.10	File Menu	11
2.11	Edit Menu	12
3	User-Centered Design	13
3.1	Focus on Explorable Systems	13
3.2	Paper Prototyping	14
3.3	Gathering User Feedback	15
3.4	Computer Prototype	16
3.5	Final Product	17
4	Programming Practices	18
4.1	Graphical User Interface Layout	18
4.2	Accessing the Database	19
4.3	Model-View-Controller Architecture	19
4.4	Inheritance	19

4.5	Observer Pattern	22
4.6	Undo/Redo Functionality	23
5	Conclusion	25
5.1	Future Work	25
5.2	Final Thoughts	26
	Bibliography	27

List of Figures

2.1	Screenshot of the file chooser	4
2.2	Screenshot of the main window	5
2.3	Screenshot of the summary area	6
2.4	Screenshot of the <i>Race</i> tab	7
2.5	Screenshot of the <i>Traits</i> tab	7
2.6	Screenshot of the <i>Hindrances</i> tab	8
2.7	Screenshot of the <i>Edges</i> tab	9
2.8	Screenshot of the <i>Gear</i> tab	10
2.9	Screenshot of the <i>Background</i> tab	11
2.10	Screenshot of the <i>File</i> menu	11
2.11	Screenshot of the <i>Edit</i> menu	12
3.1	An initial sketch of the user interface.	14
3.2	A picture of the completed paper prototype.	15
3.3	Screenshot of an early user interface, used for interface testing.	16
3.4	Screenshot of the final user interface	17
4.1	Modified screenshot demonstrating GridBagLayout	18

Chapter 1

Introduction

My thesis project involved creating a database program to assist in creating characters for the tabletop role-playing game *Savage Worlds*, published by Pinnacle Entertainment Group. The working title for this program is “Wild Card Creator,” because the main characters in *Savage Worlds* are referred to in the rules as “Wild Cards.”[3] I specifically built the program for the “Deluxe Edition” of *Savage Worlds*, published in 2011, and am referring to it whenever I use the term “*Savage Worlds*.”

1.1 What is a Tabletop Role-playing Game?

Tabletop role-playing games, like *Savage Worlds*, are sometimes referred to as “pen and paper” role-playing games because they typically are played without a computer and the players’ characters are often written out “using a pen and paper.” These games are played with a group of people collectively working together to create a story with the action imagined or represented by miniatures, rather than acted out. One player is the “Game Master” (GM) who sets up the story and facilitates game play while the remaining players each describe the actions that their character is taking in response to the situation. Hundreds of tabletop role-playing games exist, the most well-known of which is *Dungeons & Dragons*.

1.2 Purpose of this Program

Each role-playing game includes a set of rules for how to create characters, which can then be used to play the game. In recent years, it has become increasingly common to create these characters using computer software. A computer program is able to store all of the rules provided in the rulebooks and keep track of the various options available to a character. Using a computer program also insures that errors are not made and allows the user to print typewritten character sheets that are easier to read than handwritten ones. Often times, players are able to produce characters much more quickly with a character creator program than they would if they were flipping through books and copying down character options by hand. Commercial programs like this already exist such as the *Dungeons & Dragons Character*

Builder, by Wizards of the Coast, and *Heroforge*, by Heroforge Software. Neither are built to support the *Savage Worlds* role-playing game.

1.2.1 Example of Use

In role-playing games, characters typically consist of statistics for the probability that they will succeed at certain types of tasks as well as any special talents or drawbacks that they may possess. For instance, *Savage Worlds* has 109 “Edges,”¹ or special talents that a player can choose for their character to be proficient at. Many of these have prerequisite requirements that must be met before a character can qualify for it. So a character might be able to take the “Fleet-Footed” Edge, making his pace much faster, if his Agility attribute is already sufficiently high.[3]

A character creator program would be able to help players quickly determine whether the character qualifies for a particular Edge, make a note of the benefits the Edge provides, and ensure that no errors were made. Although this is simple enough to do manually, using a computer program is faster and prevents mistakes such as giving the characters options that they do not qualify for.

1.2.2 Program Scope

Savage Worlds is special in that it is a role-playing game system that is designed to be used for virtually any genre, including fantasy, sci-fi, or pulp. The “core book” (the most recent of which is *Savage Worlds Deluxe*) contains rules and character options for use in any genre, which have been included in this program.

Additional “setting” books have been produced, providing a specific game world that the players can set their games in. These settings typically come with additional game options that are supplemental to the core rulebook. For instance, *Deadlands* is a Western-Supernatural setting and the book for it contains additional character options to make it easier for a player to create gunslingers and other characters for use in the setting. Although I hope at some point in the future to provide the ability to create characters in supplemental settings such as *Deadlands*, users are currently restricted to creating characters using the options from the core rulebook.

1.3 User-Centered Design

While creating this program, I have done my best to apply principles of user-centered design. User-centered design is defined as “a philosophy based on the needs and interests of the user, with an emphasis on making products usable and understandable.” [4] Although user-centered design can be applied to the design of any object that a user might interact with, it is particularly applicable for computer programs. Program that follows user-centered design principles are generally considered to be intuitive and easy to use.

¹In *Savage Worlds*, certain game terms such as Edges, Hindrances, and Traits are capitalized. I have chosen to use the same capitalizations and other formatting when referring to these terms.

An important part of applying these principles involved showing prototypes of the program's user interface to potential users and asking for them to provide feedback about what aspects of the user interface made sense to them and what aspects could be improved. This process is further described in section 3.2.

1.4 Programming Language

I decided to create my program in the Java programming language because it is the programming language I am most familiar with. I briefly considered writing the program in C++ or ObjectiveC, but these are two programming languages that I am not as familiar with and I felt that learning a new language (in addition to everything else) would have resulted in me producing a less developed program. Java also has the benefit of being cross-platform; a program written once in Java can be run on Mac OS X, Windows, Linux, or other operating systems without any modifications to the code. Furthermore, Java includes the Swing package, a powerful, platform-independent framework for creating a Graphical User Interface (GUI), which enabled me to easily create a user interface that followed user-centered design principles.

1.5 Database Framework

Early on I decided that I wanted to use a database to manage the various character options. Doing so would allow for more flexibility—no code would need to be modified when new character options were added to the database—and would provide me an opportunity to learn the skills needed to use databases in future programs that I might write.

After making this decision, I had to choose which database framework to use and examined several free versions. Some, like PostgreSQL (*pn. post-gres-que-el*), are very powerful, but are intended to be used on servers so that multiple computers can access the database. Because I only needed a program to access a database on the same computer, these sorts of databases were unsuitable for my purposes. The Java programming language includes the Java Database Connectivity (JDBC) framework, but it is only usable in Java. I was hoping to use a framework that would be available not only for Java, but also for programming languages that I might be using in the future.

I ultimately decided to use SQLite (*pn. see-quel-ite*), a free database framework that is in the public domain. This framework is cross-platform and does not require a server. Instead, a program using SQLite directly accesses the database file on the same computer. Furthermore, SQLite advertises itself as “the most widely deployed SQL database engine in the world,” being used by a number of notable companies including Adobe, Apple, Mozilla, Nokia, and Oracle[1]. I decided that since it was so widely used, it would be advantageous for me to become familiar with working with SQLite.

SQLite databases are not accessible in the standard Java language, so I needed to use a “wrapper” to translate Java instructions into the native C programming language instructions that SQLite uses. I decided to use “sqlite4java,” created by Almnworks and released under the Apache License 2.0, and used it extensively throughout the program.

Chapter 2

Program Overview

2.1 Load Database Window

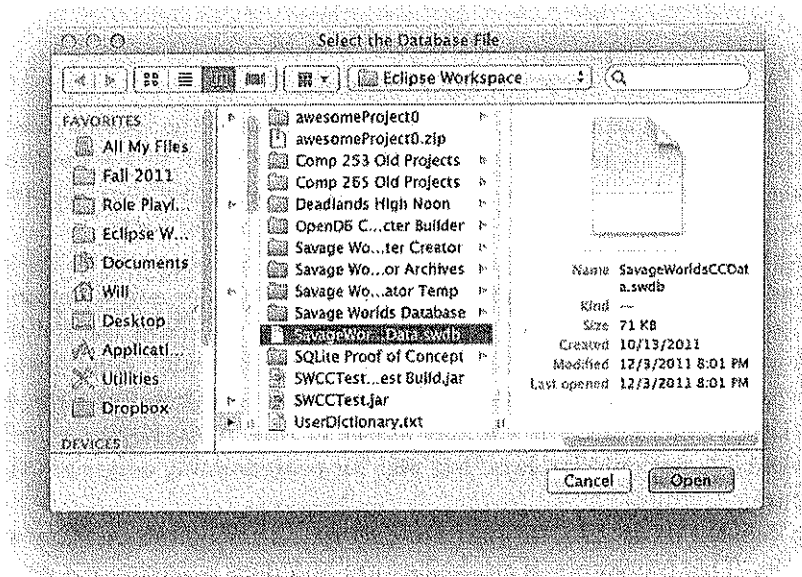


Figure 2.1: Screenshot of the file chooser for loading the database file

When the program is first launched, a message appears informing the user that they will need to load a database file of character options, which has the extension “.swdb” (standing for *Savage Worlds DataBase*). After the user clicks the “OK” button, a file chooser appears in which the user can select the location of the database. The file chooser matches the default appearance of file choosers on the operating system that the program is running on (in Figure 2.1, it matches the standard appearance on Mac OS X).

If the user selects an invalid file (i.e. one that is not a *Savage Worlds* character creator

database) or does not select a file at all, a message appears informing the user of their mistake and they are brought back to the file chooser. When the user selects a valid database file, the user is brought to the main window.

2.2 Main Window

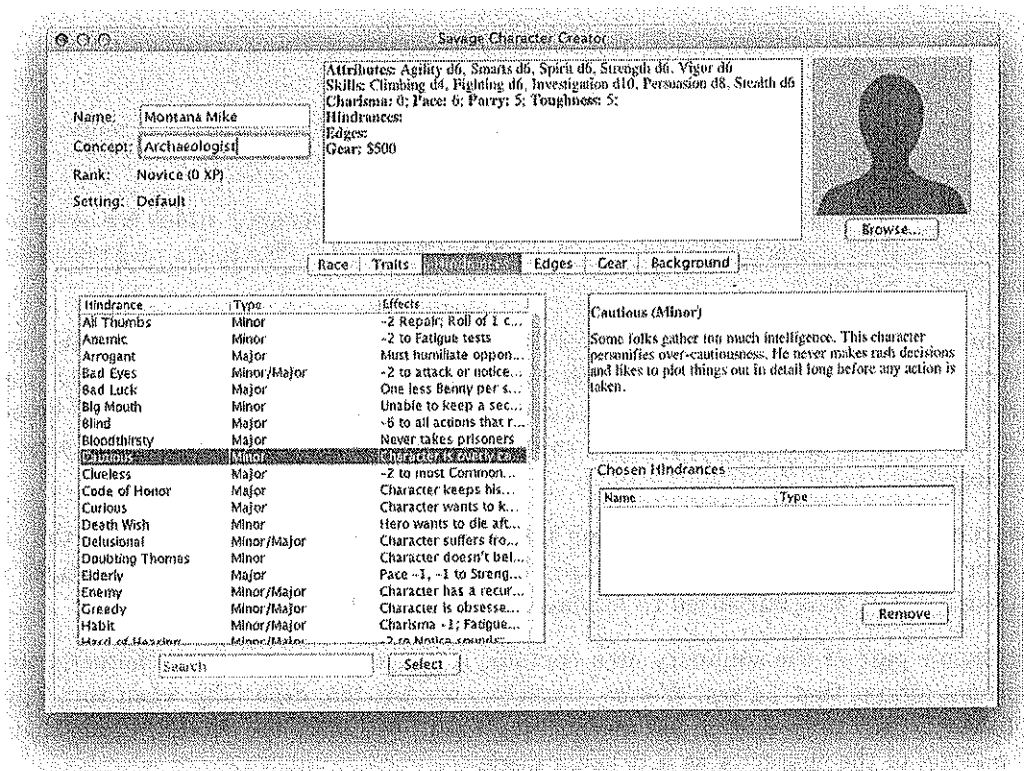


Figure 2.2: Screenshot of the main window

The main window (Figure 2.2) is where the user will spend the majority of their time. The top third of the screen contains a group of elements collectively referred to as the “summary area” because they provide a summary of the current state of the character being created. The bottom two thirds contain a tabbed panel which contains several different tabs, each of which handle a different aspect of character creation. The user can access the tabs in any order.

2.3 Summary Area

The left part of the summary area contains four labels: “Name,” “Concept,” “Rank,” and “Setting.”

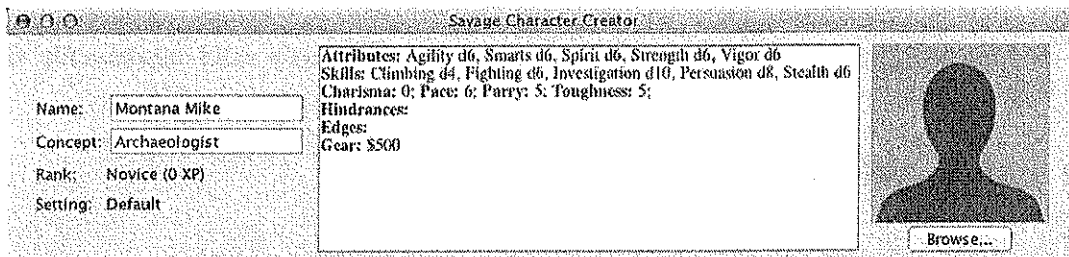


Figure 2.3: Screenshot of the summary area

- **Name:** This label is for a text field in which the user can provide a name for the character.
- **Concept:** This label is for a text field for the character’s “concept,” which is a description of what sort of character they are, such as “Mad Scientist,” “Professor of Archaeology,” or “Space Marine.”
- **Rank:** This label is for the character’s Rank and Experience Points, which in *Savage Worlds* is an indication of how much experience they have. This is currently initialized to Novice at 0 XP, allowing the creation of a basic character. Because creating an experienced character is not yet supported by the program, this option cannot be changed.
- **Setting:** This label is for the setting that the character is built in. Because the character creator does not yet support creating characters from supplemental settings (see Section 1.2.2), it is initialized to “Default” and cannot be changed.

The center part of the summary area contains a text description of the character in the conventional format that *Savage Worlds* characters are written, especially when they appear in published books. This section automatically updates to reflect the modifications that the user is making to the character as it is being made.

In the right part of the summary area, the user can select a portrait for the character. By default there is an image showing the outline of a nondescript person. Below that is a button labelled “Browse...” When clicked, a file chooser appears that allows the user to select an image file for the character’s portrait. If an invalid image is selected, a message appears informing the user of the error. When a valid image is selected, the image of the nondescript person is replaced by the image that user selected.

2.4 Race Tab

Step 1 of the character creation process is to choose “any race available in your particular setting,” the default of which is *Human*.^[3] Some settings, allow for additional races such as Elves, Dwarves, Androids, Atlanteans, or other non-human characters. The left section of the the tab (see Figure 2.4) allows the user to select any of the races that are listed in *Savage Worlds Deluxe* and it is currently up to the user to ensure that they are compatible with the

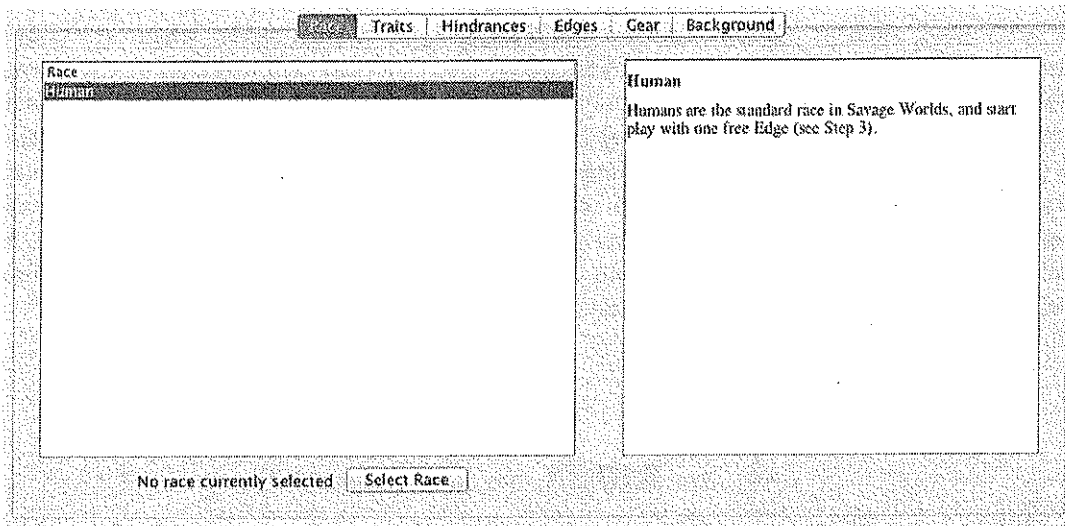


Figure 2.4: Screenshot of the *Race* tab

setting that they are making the character for. Below that is a button to select the race and a label informing the user which race is currently selected. The right section of the tab shows a full description of the selected race and the benefits a character would get by selecting it.

2.5 Traits Tab

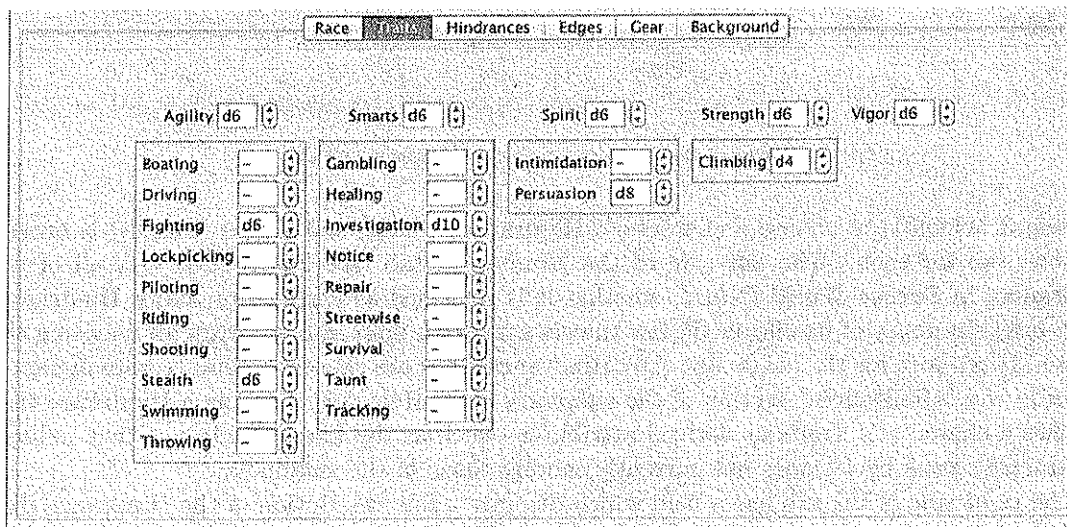


Figure 2.5: Screenshot of the *Traits* tab

Next is Step 2, which is purchasing “Traits.” Traits collectively represent a character’s

basic attributes, such as Agility, Smarts, and Strength, as well as skills, which are trained aptitudes such as Shooting, Stealth, and Persuasion. Because each skill is linked to a particular attribute, the program’s interface physically groups skills below the attribute they are linked to (see Figure 2.5). Traits are represented by polyhedral dice, ranging in even numbers from a d4 (4-sided die) to a d12, with higher-sided dice representing greater ability. Each Trait in the tab has a spinner next to it that only allows a user to select valid die types (i.e. the user cannot select values for which a polyhedral die doesn’t exist, like a d7). Modifying a Trait via one of the spinners is instantly updated in either the “Attributes” or the “Skills” sections of the summary area above the tabbed panels.

2.6 Hindrances Tab

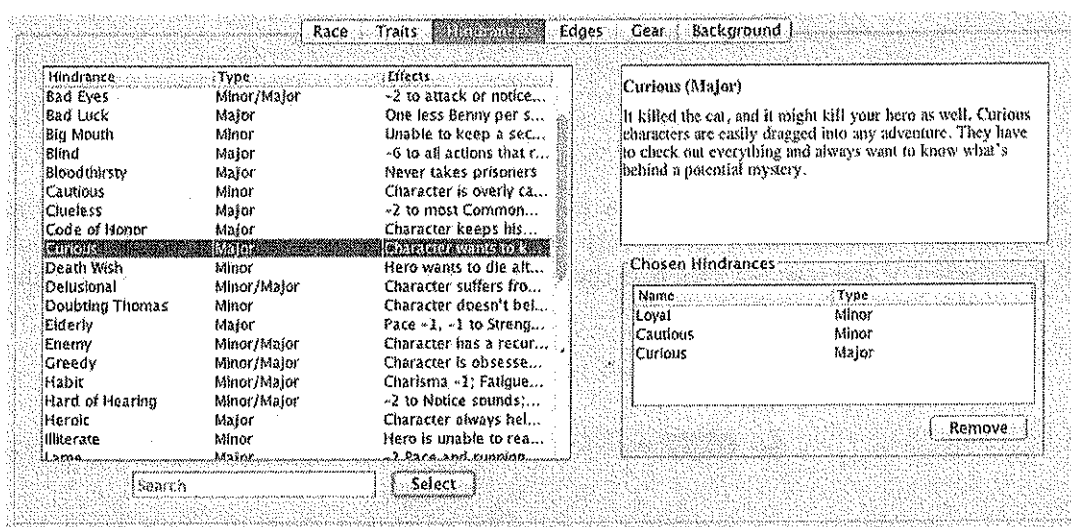


Figure 2.6: Screenshot of the *Hindrances* tab

Step 3, which is to choose a character’s Hindrances (flaws) and Edges (talents) is broken into one tab for each. The left side of the *Hindrances* tab (see Figure 2.6) lists all of the Hindrances in *Savage World Deluxe*, whether it is classified as a minor or major Hindrance, and a brief summary of its effects. Below that is a search filter and a button for choosing the selected Hindrance for the character. Any Hindrances that are chosen or removed are instantly shown in the “Hindrances” section of the summary area. However, choosing a Hindrance that modifies a character’s Traits or derived attributes—such as the *Lame* Hindrance that reduces a character’s pace by 1—does not currently modify those in the summary.

The right side of the tab has a field which displays a full description of the Hindrance that is selected from the list on the left. Below that is an area where Hindrances that are chosen are listed and can be removed.

2.7 Edges Tab

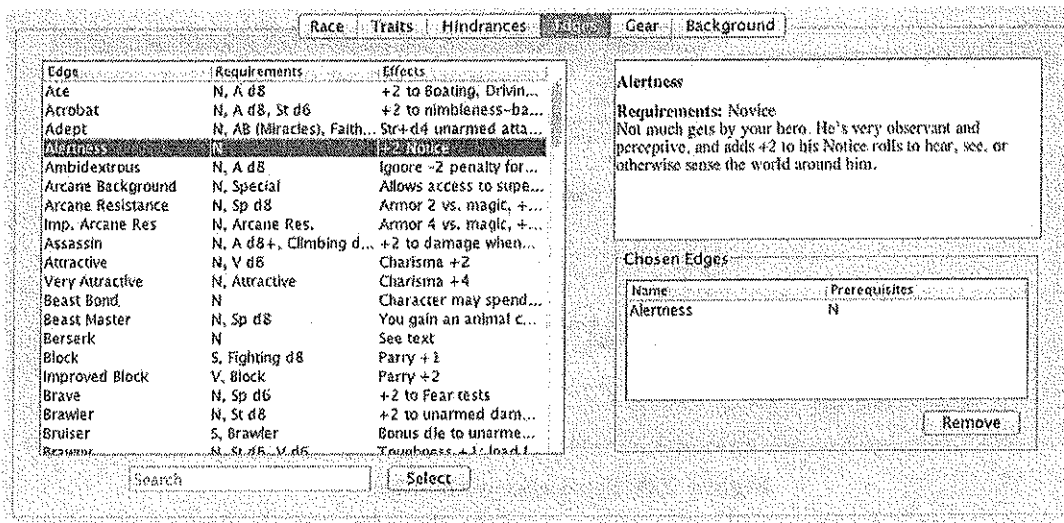


Figure 2.7: Screenshot of the *Edges* tab

The *Edges* tab (see Figure 2.7) continues Step 3 of the character creation process. The left side of this tab shows a list of Edges much like the one in the *Hindrances* tab and also has below it a search filter and a button for choosing the selected Edge for a character. Edges that are chosen for a character are instantly shown in the “Edges” section of the summary area, but do not currently modify any other part of the character. For instance, the *Alertness* Edge provides a +2 bonus to the *Notice* skill, but there is currently no change in the “Skills” section to reflect this.

On the right side of the *Edges* tab is a large area that displays a full description of the Edge that is selected in the list on the left. Below that is an area that displays which Edges the character has taken and includes a button which allows a chosen Edge to be removed.

Each Edge has prerequisite requirements that must be met before it can be purchased. This program does not yet check to make sure that only characters who meet the prerequisites can take the Edge.

2.8 Gear Tab

Step 4 is to purchase the character’s starting gear. New characters have \$500 with which to purchase their gear, but some supplemental settings allow a character to start off with more or less money. This program assumes that the default amount of \$500 is being used.

The left side of the tab (see Figure 2.8) includes a list of the different types of gear that can be purchased, such as “Hand Weapons,” “Armor,” and “Mundane Items.” When one of these is selected, the table in the upper-right section of the tab updates to list gear of the selected type.

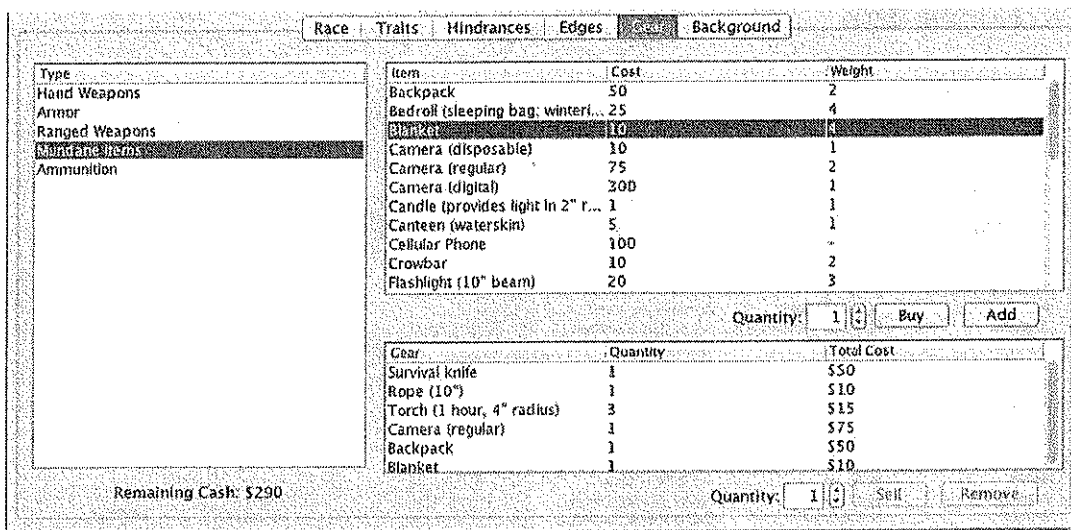


Figure 2.8: Screenshot of the *Gear* tab

Below this table is a spinner for specifying the quantity that the user would like to purchase along with two buttons, which are grayed out unless an item is selected. The “Buy” button adds a number of pieces of the selected gear equal to the number in the quantity spinner. The total cost of the transaction is then subtracted from the character’s total money or an error message is shown if there is not enough money for the purchase. The “Add” button simply adds the selected gear to the character without modifying the character’s money. This might be useful if the GM of a game declares that all players can create a character who starts with certain equipment for free.

A second table lists the gear that has been purchased for the character. Items bought or added using one of the aforementioned buttons are instantly displayed in both this table and the summary area at the top of the window. A quantity spinner and two buttons are below this table which allow the user to “Sell” or “Remove” a piece of selected gear in the quantity specified by the spinner. Finally, the bottom left of the tab includes a label displaying how much money the character has.

2.9 Background Tab

This section is for Step 5 of the character creation process: “Finish your character by filling in any history or background you care to.” [3] Because this is a fairly open-ended step, this tab only includes one large text area that the user can use to type whatever particular background details that they wish to add (see Figure 2.9).

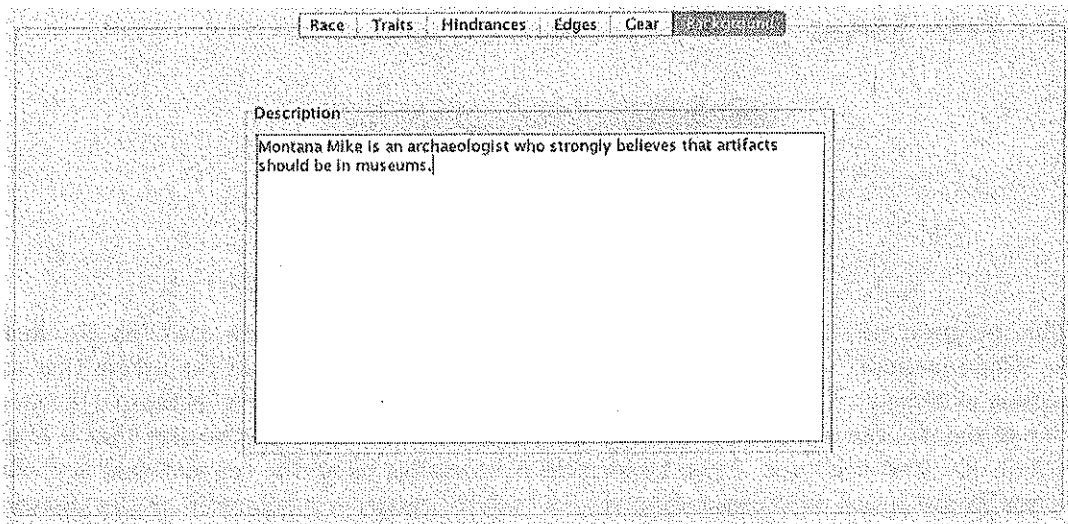


Figure 2.9: Screenshot of the *Background* tab

2.10 File Menu

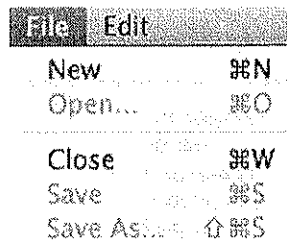


Figure 2.10: Screenshot of the *File* menu

It is common practice for programs to contain a menu labelled "File" which contains menu items for the following actions: "New," "Open," "Close," "Save," and "Save As..." This program also includes this menu (see Figure 2.10) to provide the standard functions that users expect. Each menu item is accessible using the conventional keyboard shortcuts for the operating system that the program is being run on (for instance, the keyboard shortcut for "New" on Mac OS X is Command-N and on Windows is Control-N). Loading and saving characters has not yet been implemented, so these menu items are grayed out to visually show the user that these actions are not available.

When the program is running on Windows, an additional menu item named "Exit" is listed, as it is conventional to include such a menu item in the File menu of Windows programs. This does not appear on other operating systems where it is not conventional, thus users are given an experience that is consistent with what they would expect to see.

2.11 Edit Menu

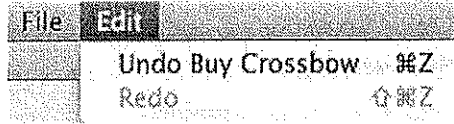


Figure 2.11: Screenshot of the *Edit* menu

This program also includes an “Edit menu (see Figure 2.11) which contains the menu items “Undo [action]” and “Redo [action]” where [action] is the name of the action that can be undone or redone. If either function is not available because there is no action to be either undone or redone, the corresponding menu item is grayed out to inform the user that it is not possible to do it at this time. Each menu item is accessible using the conventional keyboard shortcuts for the action on the operating system that the program is being run on (for instance, the keyboard shortcut for “Edit” on Mac OS X is Command-Z and on Windows is Control-Z).

Chapter 3

User-Centered Design

User-centered design is an important concept for making computer programs easy to use. Specifically, the creator of a program should “make sure that (1) the user can figure out what to do, and (2) the user can tell what is going on.” [4] A program that does not meet these two requirements is often considered to be difficult to use and unintuitive.

3.1 Focus on Explorable Systems

The Design of Everyday Things states that one of the major ways that computers, and by extension the programs that run on them, can be made easier to use is to invite exploration from the user. Norman names three requirements that a computer must meet in order for it to be considered explorable:

1. In each state of the system, the user must readily see and be able to do the allowable actions.
2. The effect of each action must be both visible and easy to interpret.
3. Actions should be without cost. When an action has an undesirable result, it must be readily reversible. . . In the case of an irreversible action, the system should make clear what effect the contemplated action will have prior to its execution; there should be enough time to cancel the plan. [4]

During my initial planning, I attempted to make sure each action was readily visible by drawing rough sketches of the user interface (see Figure 3.1), making sure that there were visible means of accessing each function. Whenever possible, I used conventional user interface elements, such as buttons, text fields, and menu items, which would behave in the manner that the user would expect. If it was not possible to perform an action at a certain time, I either grayed out the element or, in the case of text fields, made them unable to be edited.

In order to meet the second requirement, I gave all user interface elements a label that described their function. The third requirement was met by creating undo and redo functionality, which is described in Section 4.6.

Name	Type	Setting	Description
Arrogant	Minor	Care	Blat, Den
Gluter	Major	Measurement	Blat, Den
Bad Eyes	Minor	Conf	Blat, Den
Bad Eyes	Major	Care	Blat, Den

Figure 3.1: An initial sketch of the user interface. This one shows how I intended to lay out the list portion of the *Hindrance* tab.

However, Norman asserts that designers are not typical users and that “designers often become expert with the *device* they are designing. Users are often expert at the *task* they are trying to perform with the device.” [4] Even though the program seemed easy for me to use (since I designed it), ultimately it needed to be intuitive to someone who wanted to use the program but did not have any initial familiarity with program itself. Therefore, it was necessary to gather feedback from potential users and determine if they too found that the program was explorable and therefore intuitive.

3.2 Paper Prototyping

Paper prototyping is a means of testing for user-interface design by making a mock-up of the interface on paper before anything is programmed and then asking for feedback from potential users. Because the potential interface is either drawn or pasted together, it is very cheap and easy to create. Professional companies creating software, web applications, and websites frequently use paper prototyping and are ultimately able to save both time and money by creating a better program for their users without the need to make major revisions later. [2]

For my project, I created a paper prototype by cutting out small pieces of paper and drawing a small sketch of an element, such as a button or a text field, on each one. I then attached each piece to a larger piece of paper, which represented a section of the program window. I decided to affix each element using a roll of tape so that elements could be easily removed and relocated if necessary to visually show the users what the interface would look like if elements were arranged differently.

Using this process I created a model for the header part of the window, a model for the tabs, and one model for each of the tabbed panels, which could be swapped to create the effect of the user switching between tabs (see Figure 3.2).

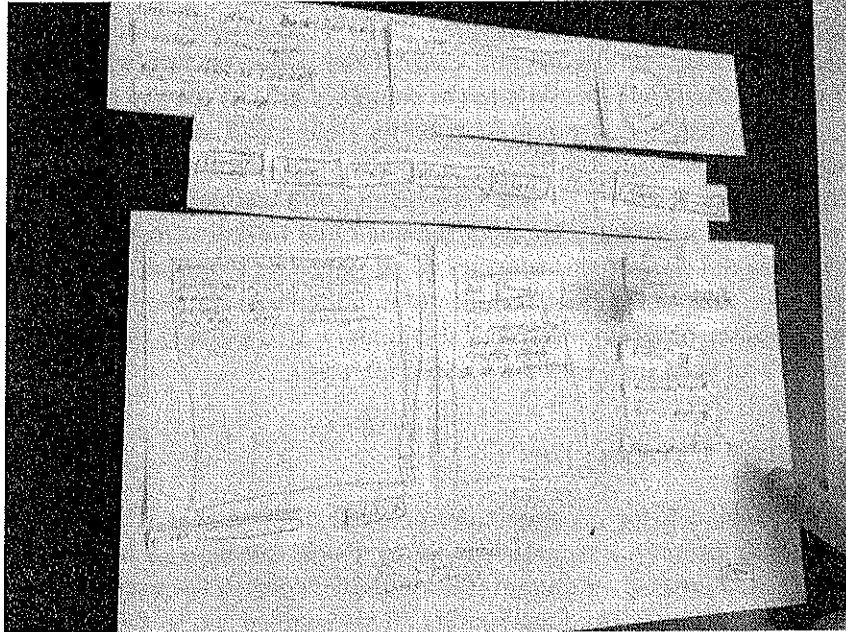


Figure 3.2: A picture of the completed paper prototype. The largest piece of paper represents the panel that appears when the *Hindrance* tab is selected and can be swapped with similar pieces of paper representing other panels.

It took me about an hour and a half to create the entire paper prototype interface. While making it, I already began to discover elements of my interface that wouldn't work the way that I had intended. For instance, I discovered that there was not enough space on a certain panel to fit all of the elements in the arrangement that I had originally planned, so I needed to make changes to create a model that had a better arrangement.

3.3 Gathering User Feedback

Although the interface seemed logical and intuitive to me, I needed to show it to others who might use the program to determine if they were also able to find it easy to use. I showed the paper prototype to three students at Wittenberg University who were familiar with *Savage Worlds* and would likely use this sort of program.

When I met with each potential user, I showed them the paper prototype as if they were using the program. I asked each user what they thought of the arrangement of elements and if they had any suggestions to make it more intuitive. I also asked what they would expect to happen when they interacted with a certain user interface element, such as pressing a particular button. If what they thought would happen was different than what I intended to happen, I asked what could be changed in order to better convey the intended action.

In my original prototype, there was an additional tab named *Concept* which allowed a user to input the character's name, concept, Rank, and the setting that they were in. When I

showed this to the potential users, they thought that this was rather cumbersome and did not like the fact that the character's name and concept could not be seen when other tabs were selected. So they suggested that I move it to the summary area at the top of the window. After some thought, I decided that this would be a better design and so the computer prototype I later created included this suggestion.

Much of the feedback I was given was very helpful and enabled me to create an interface that I discovered was far more intuitive than the one I had initially created. However, some of the users provided feedback that was contrary to another user's feedback or made suggestions for the user interface that were either technically impossible to implement or violated typical convention for computer program interfaces. Ultimately, I considered the feedback that I felt was most valuable and used it to create a revised interface.

3.4 Computer Prototype

After receiving feedback for my paper prototype, I decided that it was not necessary to create a second paper prototype to gather more feedback and instead began to program the user interface in Java. I created a program that roughly displayed the user interface of the final product, but did not yet have any functionality. This enabled me to create a new prototype of the user interface (which looked exactly like the final product since it was on the computer) that I could show to potential users (see Figure 3.3).

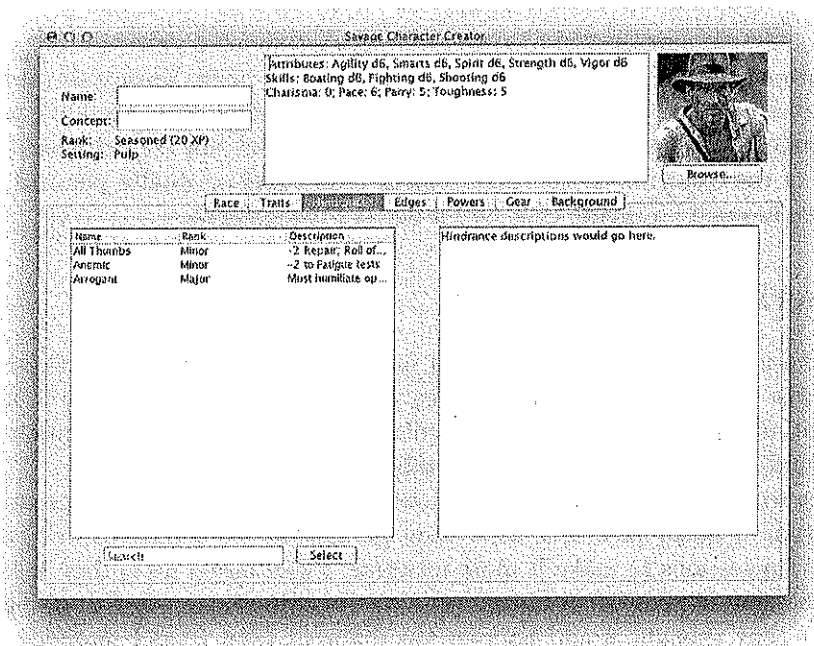


Figure 3.3: A screenshot of an early version of the user interface, which was shown to users during the second round of user interface testing.

I showed the computer interface to the same three potential users in order to receive additional feedback. This time, I received feedback that the *Gear* tab seemed to be very unintuitive and my testers had ideas for a better layout, many of which I implemented. Besides that, I received surprisingly few suggestions about the interface and decided that I did not need to do an additional round of interface testing.

3.5 Final Product

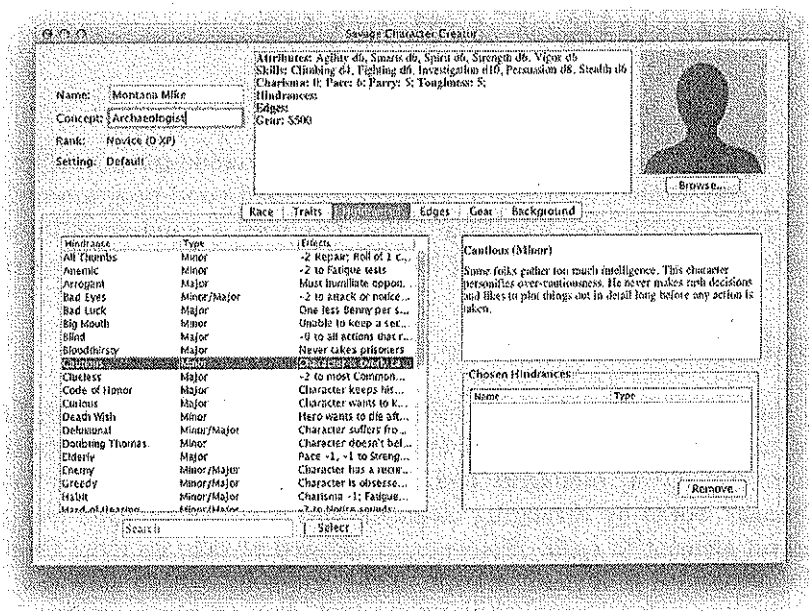


Figure 3.4: A screenshot of the final version of the user interface.

The final version of the user interface (see Figure 3.4) incorporates much of the feedback that was given during the user interface testing phase and is much more intuitive than my initial designs. When I have informally shown nearly-complete versions of the program to others, I have often received comments that the program looks nice and appears to be simple to use. These compliments indicate to me that following user-centered design principles in my programs is an effective way to make it better for the users who will ultimately be using it.

Chapter 4

Programming Practices

In addition to creating a program that followed user-centered design principles and was easy for a user to use, I also wanted to create a program that used good programming practices so that it would be easy to write and maintain. I therefore used a number of design patterns in order to efficiently implement functions and frameworks to better organize my code.

4.1 Graphical User Interface Layout

To organize the user interface elements in the manner that I had imagined, I needed to use a flexible layout manager that would allow me to position elements in an intuitive way. Java Swing contains several layout managers and out of them, I decided to use GridBagLayout. This layout manager works by arranging elements into a grid with rows and columns that resize to the largest element in each. Elements are allowed to take up more than one grid cell and may also be positioned in a small portion of the grid cell. Within each cell, elements can be set to align to a certain position in the cell or to fill all available space (see Figure 4.1).

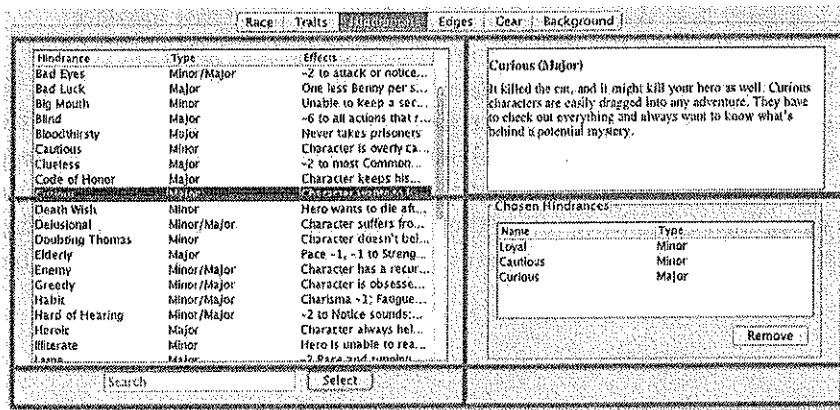


Figure 4.1: Screenshot of the program, with gridlines overlaid on it to show how GridBagLayout is being used.

4.2 Accessing the Database

All character options are stored in a SQLite database file, which I have given the arbitrary file extension `.swdb` (an acronym for *Savage Worlds DataBase*). Within this file is a database for each type of character option. Hindrances, Edges, Attributes, Skills, and all related data are included in rows within the appropriate tables.

In order to access the database, a SQL command needs to be sent to the SQLite database. For instance, to load all of the Skills in *Savage Worlds*, the the SQL command `"SELECT * FROM "Skills";"` is sent. This returns each of the rows in the Skills table, which the program can then process. The Search Field in tabs such as the Hindrances tab similarly access the database by sending the SQL command `"SELECT Description FROM Hindrances WHERE Hindrance '%' + [SearchTerm] + '%';"` where `[SearchTerm]` is the text that is in the search field. This returns all Hindrances whose name contains the text in the search field, even if it is not a complete match. Powerful commands such as these drive much of the program's functionality.

In order to interact with the program, a SQL command needs to be embedded into a line of Java code. This program then use of the `"sqlite4java"` wrapper to convert the instruction in the Java programming language to an instruction in SQLite's native C programming language.

4.3 Model-View-Controller Architecture

The model-view-controller architecture is a way of organizing classes and other sections of code based on their function. Models contain data, but not information about how it is going to be displayed or how the user will interact with it. Views are capable of displaying a model without knowing ahead of time what particular data values are being displayed. Controllers handle interactions between the user and the model without being told how they will be stored or displayed. [5]

In this program, the character that the user is making is a "model" that is accessed by a view which displays it on the screen and is modified by a controller that allows a user to make changes via mouse clicks and keyboard commands. The main benefit of this organization is that data models can be viewed or controlled in different ways with minimal changes to the code. This separation makes it easy to, for instance, add an additional view of the character that allows it to be displayed as a PDF file rather than as a window on the screen.

4.4 Inheritance

A feature of object-oriented programming languages like Java are that they allow for "inheritance" which "greatly increases the reusability of classes and also minimizes the duplication of code." [5] This works by having one class include all of the properties of another class.

One way that the program makes use of inheritance is with a character's gear. Gear is a broad category that covers mundane items, ranged weapons, hand weapons, and armor. Each of these have common pieces of information, such as cost and weight, but some also have unique information such as damage and armor protection.

I began by creating a class simply named "Gear" containing information and methods that were common to all pieces of gear. The methods in the class were as follows:

```
/**
 * A model for a piece of gear that a character may have
 * @author Will
 *
 */
public class Gear implements Cloneable{

    /**
     * Constructs a piece of gear,
     * @param name the name of the gear
     * @param cost the cost of the gear in dollars
     * @param weight the weight of the gear
     * @param notes additional notes about the gear
     */
    public Gear(String name, Money cost, double weight, String notes)

    /**
     * Returns the name of the gear
     * @return the name of the gear
     */
    public String getName()

    /**
     * Returns the individual cost of the gear
     * @return the individual cost of the gear
     */
    public Money getIndividualCost()

    /**
     * Returns the total cost of the gear based on quantity
     * @return the total cost of the gear based on quantity
     */
    public Money getTotalCost()

    /**
     * Returns the weight of the gear
     * @return the weight of the gear
     */
    public double getWeight()
}
```

```

    /**
     * Returns the notes about the gear
     * @return the notes about the gear
     */
    public String getNotes()

    /**
     * Returns the quantity of the gear
     * @return the quantity of the gear
     */
    public int getQuantity()

    /**
     * Sets the quantity of the gear to the specified vaue
     * @param quantity the new quantity of the gear
     */
    public void setQuantity(int quantity)

    @Override
    public String toString(){

    @Override
    public Gear clone()
}

```

This class is sufficient for storing mundane gear such as flashlights or trail rations, but does not have enough information to store, for instance, a sword because there is no way to store a "damage" amount. Thus it became necessary to create a HandWeapon class. But all of the methods contained in the Gear class are also required in the HandWeapon class. Rather than duplicating code, I subclassed HandWeapon so that it was a more specific form of Gear and could use all of the methods contained in the Gear class. The HandWeapon class has the following methods:

```

/**
 * A model representing a hand weapon
 * @author Will
 *
 */
public class HandWeapon extends Gear{
    /**
     * Constructs a hand weapon.
     * @param name the name of the weapon
     * @param damage the amount of damage the weapon does (e.g. Str+d6)
     * @param weight the weight of the weapon
     * @param cost the cost of the weapon

```

```

    * @param notes any additional notes about the weapon
    */
    public HandWeapon(String name, String damage, double weight,
        Money cost, String notes) {
        super(name, cost, weight, notes);
        this.damage = damage;
    }

    /**
     * Returns the damage of the weapon
     * @return the damage of the weapon
     */
    public String getDamage()

    @Override
    public String toString()
}

```

Because of inheritance, the HandWeapon class contains everything in the Gear class shown above. This prevents code duplication which in turn makes the program simpler and easier to maintain.

4.5 Observer Pattern

One common problem when creating a graphical program is how to determine when a button or other interface element has been activated so that other parts of the program can execute their code. An excellent solution to this problem is the observer pattern. In this design pattern, a button or similar element is called a “publisher” and the parts of the program that need to be informed of when the element has been activated are called “subscribers.” “Publishers maintain a list of subscribers and, whenever there is something to publish, they notify all their subscribers.[5]” So a button is given a list of the elements that need to know when it is pressed and, when that occurs, it sends a signal to all of those elements letting it know that it is been activated. In Java, a subscriber to a button or other user interface element is called an “ActionListener.”

I also used the observer pattern for informing different parts of the program when the character had been updated. The internal model containing the data for the character functions as a publisher. Whenever it is changed, it informs the character summary area at the top of the window, which is a subscriber, that the character state has been changed. Thus the character summary area knows that it needs to update its display of the character at the moment that the character has been changed.

4.6 Undo/Redo Functionality

An important tenet of user-centered design is that "Actions should be without cost. When an action has an undesirable result, it must be readily reversible." [4] Many programs include an Undo menu, which reverts to the state before the undesirable action, as well as a Redo menu item, which returns to the state before the reversion. I decided that I wanted to include these in my program.

Dale Skrien's *Object-Oriented Design Using Java* suggests the "command pattern" for implementing undo/redo functionality[5]. In this implementation, when the state of the program is modified, an Object containing the encapsulated (i.e. self-contained) instruction is generated. This instruction includes both the code to perform the intended action as well as the code to reverse it. The interface looks like this:

```
public interface ReversibleAction {

    /**
     * The code that executes the action
     */
    public void action();

    /**
     * The code that undoes the action that was previously done
     */
    public void reverseAction();

    /**
     * Allows the action to be identified (e.g. Buy Gear), which can then
     * be used in the menu items (e.g. Undo Buy Gear).
     */
    public String toString();
}
```

The following is an example of an encapsulated instruction using the interface above that I created to add a Hindrance:

```
private class AddHindranceAction implements ReversibleAction{

    //The Hindrance being added
    private Hindrance hindrance;
    //The panel that will display that the hindrance is added
    private HindrancePanel panel;
    //The internal model keeping track of taken hindrances
    private HindranceTracker tracker;

    public AddHindranceAction(HindrancePanel panel,
```

```

        HindranceTracker tracker, Hindrance hindrance){
    this.panel = panel;
    this.tracker = tracker;
    this.hindrance = hindrance;
}

@Override
public void action() {
    this.tracker.addHindrance(this.hindrance);
    this.panel.addHindrance(this.hindrance);
    this.tracker.notifyObservers(); //Used for the Observer pattern
    DebugMode.print("Added the " + this.hindrance + " Hindrance.");
}

@Override
public void reverseAction() {
    this.tracker.removeHindrance(this.hindrance);
    this.panel.removeHindrance(this.hindrance);
    this.tracker.notifyObservers(); //Used for the Observer pattern
    DebugMode.print("Undid addition of the " +
        this.hindrance + " Hindrance.");
}

@Override
public String toString(){
    return "Add " + this.hindrance + " Hindrance";
}
}

```

This encapsulated instruction is placed onto a stack data structure, henceforth referred to as the “undo stack.” If a modification was made to a character, then instead of storing the entire state of the character before and after the change, only the code that performed the change, encapsulated into its own class, is pushed onto the stack. As more changes are made, additional encapsulated pieces of code are pushed onto a stack.

If the user invokes the Undo command, the instruction on the top of the undo stack is popped, the command is undone (using the encapsulated code’s *reverse()* method) and then is pushed onto another stack, referred to as the “redo stack.” If the user invokes the Redo command, the instruction at the top of the redo stack is popped, the original action is executed (using the encapsulated code’s *action()* method) and then is pushed onto the undo stack. If any additional instructions are executed without using either command, the instruction is pushed onto the undo stack and the redo stack is cleared.

The Command Pattern is relatively space efficient and easy to program and therefore it was the way I decided to implement undo and redo functionality in my program.

Chapter 5

Conclusion

This project has been both an application of what I have learned while at Wittenberg University as well as a learning opportunity to further my knowledge of concepts in Computer Science. Computer Science 250 was helpful for teaching me the Java programming language and how to use data structures. I made extensive use of design patterns and object-oriented programming that I learned in Computer Science 353 (formerly 253). And the background in different programming languages that I learned in Computer Science 260 made it easier to learn how to use SQLite.

However, this project would not have been possible without looking beyond the classroom to learn. I did my own research about user-centered design and using a database, both of which were used extensively in the project. I also realized that there were some things I could not learn from books and gathered extensive feedback from potential users to make a program that was not only usable, but also enjoyable.

5.1 Future Work

Although I have created a fine program for this project, I still feel that it can be improved and would like to continue working on it. Some features of *Savage Worlds* are still unimplemented, such as “Arcane Backgrounds” (characters with magical or otherwise superhuman powers). With more time, I hope to be able to implement missing features such as those and allow the creation of any character that can be made with the core book.

I also hope to make the program extensible, enabling users to add character options from supplemental books to the program and be able to make characters who can use them. For instance, Pinnacle Entertainment Group has a line of books for the *Deadlands* setting, a Western with some supernatural elements. Additional Edges, Hindrances, and Gear are available to help players create characters that are better suited for that setting. Ideally I would like for my program to be extensible enough that the majority of the published supplements from both Pinnacle and third-party publishers will be able to be added to this program.

Finally, I hope to present my finished program to Pinnacle Entertainment Group and ask for their feedback and permission to distribute it for others to use. It would be a great honor to have the company who created *Savage Worlds* to give their blessing for me to share this

program that helps players create characters for their game.

5.2 Final Thoughts

I have spent many, many hours working hard on this project and I am very proud to present this program that I have created. Several people who play *Savage Worlds* have already asked me when they might be able to use this program and I hope that, with some improvements, it will be very soon. I am very thankful that Wittenberg has given me the opportunity to further my studies in Computer Science by creating a program for a game I love and I hope that what I have learned from this project will help me in all of my future endeavors.

References

- [1] SQLite homepage. <http://sqlite.org>. [Online; accessed 25-November-2011].
- [2] What is paper prototyping [examples]. http://paperprototyping.com/what_examples.html. [Online; accessed 5-December-2011].
- [3] Shane Lacy Hensley et al. *Savage Worlds Deluxe Edition*. Pinnacle Entertainment Group, 2011.
- [4] Donald A. Norman. *The Design of Everyday Things*. Basic Books, 2002.
- [5] Dale Skrien. *Object-Oriented Design Using Java*. McGraw-Hill Science/Engineering/Math, 2008.

