2006

Family tree manager

Catherine Sullivan

Family Tree Manager

Master's Project Report

Catherine Sullivan (colsewski@rochester.rr.com)
Rochester Institute of Technology
Computer Science

October 28, 2005

Table of Contents

1. Abstract

Many people are interested in learning about their family history. Discovering who has come before us can help us learn more about who we are, and possibly why we are the way we are. Over the years, more and more

people have turned to software products to help them build and manage their family tree information. These products allow users to easily enter and retrieve the information, as well as provide graphical representations without the arduous task of drawing by hand.[1,2,3,4] In addition, some manufacturers have made agreements with genealogy search services, and integrated the process of searching for relatives into the software. A user can initiate a search for a family member at the same time he is entering information into the GUI.[3] While this is certainly an important advancement, it appears that it has come at the expense of advancements in the user interface.
The display of genealogy data is not a simple problem. Family tree information lends itself most to a sort of tree structure, but one in which there can be any number of levels, any number of elements per level, and any number of children per element. Making matters more difficult is the fact that a user really needs some way to visualize the structure and the content of the tree simultaneously. The main focus of this project is to develop a genealogy software product implemented in Java that makes use of a graphics toolkit to create a graphical view of family tree data that allows the user to visualize the content and structure of his family tree at the same time. This graphics toolkit will need to support user selection of graphical objects, panning, zooming, and animation

2. Introduction

The first step of the design phase for this project was to determine how the data should be presented to the user, how the user will enter that data, and how the data will be stored. The best course of action seemed to be surveying genealogy software products currently on the market, examining how they address these problems, and researching what users think of these products. According to 2005 Genealogy Software Report[1], the top three products are Legacy by Millennia Corporation[2], Family Tree Maker by MyFamily.com[3], and Ancestral Quest by Incline Software[4], with Family Tree Maker being the highest rated.

3. Data Entry

The first issue is determining how the user will enter data into the system. Upon opening Family Tree Maker, the user is presented with what Family Tree Maker refers to as the “Family View” screen.[3] An image of this screen taken from the product tour on FamilyTreeMaker.com is shown below. The main couple for this screen is shown at the top. This section contains fields for name, birth date, birth place, death date, death place, marriage date and marriage place. Above each individual is a button containing the names of the person’s parents. Clicking on this button actually reloads the screen so that parents become the main couple for the screen. At the bottom of the screen is a list of children of the main couple.[3] According to the reviews, most users seem generally happy with this means of entering information.[1] While this screen provides a complete and manageable display of the data, it seems like it could be a bit confusing for a first-time user entering the screen without any existing data to look at for reference. This project will improve the first-time user’s experience by including a start-up wizard through which the user can enter the beginnings of his family tree. Once the wizard has been completed, a screen similar to Family Tree Maker’s Family View will be displayed. Since users seem happy with the Family Tree Maker’s Family View screen, it makes the most sense for this project to include something similar. Looking at the Family Tree Maker Family View also helps determine which fields this project must include for each individual. Family Tree Maker includes name, birth date, birthplace, marriage date, marriage place, death date and death place, so this project should allow the user to save at least these fields.

Figure 3.1: A screenshot of Family Tree Maker's Family View from the product demo available at http://www.familytreemaker.com.[3]

4. Data Views

The products on the market all include data views in addition to the basic family view described above. These include some means of viewing multiple levels of parents, and some means of viewing multiple levels of descendants.[1,2,3,4] Common sense also indicates that there must be some way of searching for individuals currently stored in the system. FamilyTreeMaker.com did not include any description of internal search capability in the demos[3], but it seems reasonable that a user would want to be able to search for individuals by name at a minimum, and possibly by birth and death date.

4.1. Progeny View

The progeny view consists of a tree with one individual at the root, and that individual's parents, grandparents and so on coming off the root in a recursive fashion. An image of Family Tree Maker's Progeny View taken from the product tour on FamilyTreeMaker.com is shown below. This view allows the user to see the names of parents five generations back relatively easily, but the screen looks a bit crowded once you get past the third generation. Also, it appears that the user needs to scroll through the use of arrow buttons if he wants to see more than six levels.[3] That seems to be a bit awkward. This project will make use of a graphics toolkit supporting selection of graphical objects, panning, zooming and animation to come up with a better implementation of a progeny view.

Figure 4.1.1: A screenshot of Family Tree Maker's Progeny View from the product demo available at http://www.familytreemaker.com.[3]

4.2. Search View

This project will include a search screen that allows the user to search for individuals by name, birth date, and death date. To be more user-friendly, the name fields should allow for wildcard searches since the spelling of some names can be difficult to remember. Also, it seems better to allow the user to search by birth and death date ranges in addition to being able to search for one particular date. The search results should contain enough information so the user can easily distinguish which individual he was looking for, and should ideally be sort-able. The user should also be able to jump to the family view of an individual from the search results.

4.3. Descendant View

The only means of viewing descendants that Family Tree Maker advertises is a series of printable charts and reports. They're very attractive, and as the website advertises, could easily be used to create a family tree book.[3] Unfortunately, the size of the printed page restricts the number of generations and the number individuals per generation that can be included without making the presentation of the information unattractive and unreadable. None of the other family tree software products researched appear to have anything better, and the majority of complaints posted on the software review website express dissatisfaction with the descendant view as opposed to other views.[1,2,3,4]

The problem of how to present a descendant view is not a simple one. This view is more complicated than the progeny view because while we can be reasonably sure that a person has no more than two biological parents, we cannot know in advance how many children a person will have. Suppose we start a descendant view with one couple, and their children beneath them. Unless the couple had a large number of children, this doesn't seem too difficult to show on a screen. Problems arise, however, when one attempts to include more levels of descendants in the view. Suppose you had three children, each of your children had three children, and each of your children's children had three children. In only four generations, we already have forty individuals. The fourth generation of descendants has twenty seven individuals. Ideally, a user would like to be able to traverse through the levels of his descendants, and be able to get an idea of the structure of the descendant tree as well as be able to see who the individuals are. This project will make use of a graphics library that supports selection of graphical

objects, panning, zooming and animation to come up with a descendant view that allows the user to visualize the structure and contents of his descendant tree at the same time.

5. Data Storage

The software products do not advertise how they store data, but there are two obvious options for data storage, a file or a database.

The file option seems the simplest, especially since it does not require any additional software to implement. The file could be loaded into memory at start-up, and the data in memory could be written back to the file when the user exits the application. Each record in the file could contain the information for one individual. In addition to information like names and birth dates, the file will also have to contain information about how the records are linked to one another. It would be nice if this project could handle multiple users. This could be managed by either having a separate file for each user, or by marking each record in order to indicate its owner. It would also be nice if future developers could add fields to the system. Whatever file format is chosen must take into account the fact that new fields may be added. Though this project will not offer them, it would be reasonable to assume that future developers may want to add search options more complex than just name and birth date. Suppose someone wanted to add a search option that returned all the descendants of one individual. Search options like this would be fairly complicated to implement if all we have is a flat file and data stored in memory.

In the end, it seemed that the extra complications involved in using a database would be well worth the benefits. The free version of MySQL, available from http://www.mysql.com/, seemed to be the perfect fit. It contains all the functionality needed for this project, and is free of charge as long as it is not being used for a product offered for sale, or for other business purposes.[5] The MySQL option offers a few advantages over a file.

First, there is no need to store large amounts of data in memory. Any data structure that was used to store family tree data in memory would need to be fairly complex in order to maintain all the linkages between data elements, so any interaction with this data structure would also be complicated. There could also be problems with data loss if we only saved data when the user exited the application, but writing to and reloading from a file on a more frequent basis could be pricey. The software would have to manually find the record in question in the file before it could update it. With a SQL database, information can be easily queried as it is needed, and easily updated as the user makes changes. The database design will maintain all the linkages for us, further reducing the complexity of the software.

Second, all that needs to be done to allow multiple users is to add a user identifier to each individual stored in the system. The MySQL database has built-in concurrency control.[5] Should future developers wish to make this into a true client-server application, there will not be any additional work required as far as data storage and retrieval is concerned. This project will restrict all direct database access to one class in order to simplify this process, and to keep the rest of the project independent of the data source.

Third, it is relatively simple to add fields. Since we can store data for all users in one database, one SQL statement is all it takes to add a new field, occupation, for example, to the system. This same statement can also give a default value for the new field if desired. This new field can be added to the SQL select, insert and update queries in the one class that interacts directly with the database, and then to the GUI, and then we're done.

Lastly, the only limit to the complexity of searches that can be implemented is the developer's ability to write a SQL query. There will be no need to manually search through a large data structure to find the records we need. The database can do the work for us, and more efficiently than we could do it by hand.

6. Project Goals

6.1. Building a Better View

One goal of this project is to make use of a graphics library to come up with a means of presenting descendant and progeny information that improves upon those offered by existing products. For this project, the Piccolo graphics library developed by Jesse Grosjean at the University of Maryland[8] will be used to implement better progeny and descendant views. The Piccolo 2D graphics toolkit allows the programmer to develop interfaces with panning, zooming, animation and selection of graphical objects without worrying about the low-level details.[8]

If the new family tree views are to be a true improvement over what is currently available, they should meet the following criteria:

- The view should be some sort of tree structure. A list view would only display the data elements themselves, not the relationships between the data elements.
- The user should be able to open the entire tree, or only pieces of the tree as desired. The user should be able to hide branches of the tree in order to focus on particular sections at a time.
- The user should be able to tell whether a node in the tree has children that are currently hidden, or has no children to display.
- The user should be able to move the tree around in order to concentrate on one area at a time.
- The ability to zoom in and out on the tree would be nice.

6.1.1. Space Tree Functionality

The Space Tree[10] developed by Catherine Plaisant, Jesse Grosjean, and Benjamin Bederson at the University of Maryland is the inspiration for the view used in this project. In summary, this tree view contains the following functionality:

- It allows users to collapse and expand branches of the tree.
- It provides an icon adjacent to a node indicating that the node has hidden children, and how many hidden nodes there are.
- It opens as many levels as possible given the available space on the screen.
- Node size is determined dynamically based upon the available space. If the screen is crowded, the nodes will be smaller when they are opened.

6.1.2. Progeny and Descendant Tree Functionality

The tree view developed for this project will contain the following functionality:

- It will allow users to expand or contract nodes of the tree by clicking on them with the mouse. Clicking on an unexpanded node will open one level of that node's children. Clicking on an expanded node will hide all of that node's children.
- Nodes that have hidden children will be easily distinguishable from nodes without hidden children.
- The size of the nodes will be constant. The node size will be determined based upon the normal size of the text that will be contained in the node. There isn't much point in displaying a node if you cannot read what it contains.
- The size of the tree will not be restricted to the screen size.
- Users will be able to move the tree around the screen.
- Users will be able to zoom in and out on the tree.

6.2. Selecting a Good Database Structure

Another goal of this project is to select a solid database structure that is appropriate for the information the

database must contain, and is flexible enough to allow for future development. This project will contain some limitations that are important to consider when designing the database. First, the system will not allow unlinked individuals. For an individual to be in the system, he must be the parent or child of someone else in the system. The only exception to this rule is the first person added. Second, the software will not attempt to handle multiple marriages. Each person in the system can be a member of at most one couple. Third, it will not attempt to handle same sex marriage. Each couple in the system is assumed to contain no more that one male member and no more than one female member. Future developers may wish to remove some of these restrictions, so the database design should allow for this.

Conceptually, it seems reasonable that the first table needed would be a Person table. As discussed earlier, the database needs to store the person's name (first, middle, last and maiden), birth date, birth place, death date and burial place. (Storing a "Death Place" field seemed a bit too morbid, so "Burial Place" is stored instead.) We also need to store a marriage date and marriage place. However, the marriage date and marriage place do not apply to just one individual, they apply to a married couple. There are two options here. The first option would be to use a Couple table instead of a Person table and store the marriage information there. The second option would be to leave the Person table, but move the marriage information to another table where it would be linked back to the Person table by the IDs of the two people in the couple. The first option is bad for a few reasons. First, the data would not be normalized. Second, if a future developer decided to change the software to allow for multiple marriages, the database would require a major overhaul. Therefore, option two was selected, and the database will include a Person table and a Marriage table.

Next, we need a way to uniquely identify a person in the system. Depending upon names or any other combination of a person's information being unique is not reliable, and we hardly want to copy the entire Person table into the Marriage table for each foreign key. MySQL allows for identity, or auto-increment, columns, so each person in the system can receive a unique ID through an auto-incremented column in the Person table. However, remember that this project is also supposed to allow for multiple users. We need a way to identify different users, and to distinguish which Person records belong to which user. To do this, a new table, TreeUsers, is added, and the auto-incremented ID of the user is copied into a new Person field named TreeCreatorID.

Next, we have to figure out how to store children in the database. Though this version of the software will not handle multiple marriages per person, the database design should allow for the future handling of multiple marriages. Therefore, we need to include the ID of both parents with the ID of the child. The table ParentChild will contain the PersonID of the mother and father as well as the ID of the child. Each child gets his own record in the table. This table is intended to store biological parent-child relationships rather than social parent-child relationships. Step-child and step-sibling relationships could be determined through the Marriage table.

In the end, the overall fitness of the chosen database schema should be determined by analyzing the degree of normalization achieved by the design, and by determining whether the schema could result in the creation of additive joins (also called spurious tuples).[17] The fitness of the database design will be analyzed in depth in the results section of this report.

6.2.1. Database Design

The chosen database structure is described below.

Table **TreeUsers**

Field Name	Data Type	Definition
TreeUserID	integer	Auto-increment primary key giving each user a unique ID. Not nullable.
TreeUserName	varchar(50)	Unique login name for a user. Not nullable.

Table **Person**

Field Name	Data Type	Definition
PersonID	integer	Auto-increment primary key providing a unique identifier for each individual in the system. Not nullable.
TreeCreatorID	integer	The TreeUserID of the user this person belongs to. Non nullable.
FirstName	varchar(100)	The first name of the individual. Not nullable.
MiddleName	varchar(100)	The last name of the individual. Nullable.
LastName	varchar(100)	The middle name of the individual. Not nullable.
MaidenName	varchar(100)	The maiden name of the individual, if female and married. Nullable.
Gender	char(1)	M (male) or F (female). Not nullable.
BirthDate	date	The birth date of the individual. Nullable.
BirthPlace	varchar(255)	The birth place of the individual. Nullable.
DeathDate	date	The death date of the individual. Nullable.
BurialPlace	varchar(255)	The burial place of the individual. Nullable.
Notes	varchar(255)	Notes field. Nullable.

Table **Marriage**

Field Name	Data Type	Definition
HusbandID	integer	The PersonID of the husband. Nullable, but either the HusbandID or the WifeID must be populated. The HusbandID+WifeID combination must be unique.
WifeID	integer	The PersonID of the wife. Nullable, but either the HusbandID or the WifeID must be populated. The HusbandID+WifeID combination must be unique.
MarriageDate	date	Date of marriage. Nullable.
MarriagePlace	varchar(255)	Where the marriage took place. Nullable.

Table **ParentChild**

Field Name	Data Type	Definition
ChildID	integer	Primary key. The PersonID of the child.
MotherID	integer	PersonID of the mother. Nullable.
FatherID	integer	PersonID of the father. Nullable.

7. User Workflow

7.1. Database Setup

Before the software can be used, the MySQL system administrator must create the FamilyTree database and create a user with permissions on that database. The reason for this is twofold. First, only the root user is allowed to create users and databases. Second, it seems bad practice to force the MySQL administrator to give the root password to a regular software user. The software will take care of creating the tables.

7.2. Startup

The user is presented with a login screen where he should enter his database username and password. The software then attempts to log into the database, create a TreeUser record for this user if none exists, and create the required database tables if they are not already present. If there is a problem, the user is asked to try again. Once the user successfully logs in, the behavior of the software is determined by whether the user has any records in the Person table.

If the user does not already have records in the Person table, the startup wizard will be automatically opened. The user will be guided through entering his own personal data, as well as that of his parents and grandparents. The last page of the wizard will display a summary of everything the user has entered so far just in case he wants to go back and fix something. In order for a person's information to be saved, at least the person's first and last name must be entered. The pages the user sees will depend upon what he has entered. For example, if nothing is entered on the screen for his father's information, the user will never see a screen for his father's parents. Once the wizard has been completed, the family view for the tree creator will be opened.

If the user already has Person records, the wizard will not be available. Instead of the wizard, the search screen will be opened automatically.

7.3. Continued Use

From the family view, the user will be able to edit the main couple's information, and jump to the family view of the parents or children. The user will also be able to open the progeny and descendants views. The search screen will be available from a menu option on the main window.

8. Design

8.1. Class Overview

The classes in this project are divided into three main groups or packages; client, server and util.

8.1.1. Util

The util group contains the classes Person and Couple which used throughout the project. A Person object represents one individual with all the information stored about him or her. A Couple object represents a married couple, but may contain only one person if the individual is not married or information about a spouse is not known. The Couple object also contains marriage date and marriage place information.

8.1.2. Server

The server group contains only the GenealogyServer class. This class is responsible for managing all communication with the MySQL database, including all selects, inserts and updates. Encapsulating the database access in this manner prevents the rest of the application from needing to know about how the data is stored.

8.1.3. Client

The client group contains the classes GenealogyApp, ConnectFrame, GenealogyFrame and GenealogyDesktopManager, as well as five additional class groups; descendantview, familyview, progenyview, searchview and wizard. The GenealogyApp is the main application. The ConnectFrame is the login dialog, which is opened by the GenealogyApp. The GenealogyFrame is the main application window

from which all other windows are opened. The GenealogyDesktopManager is the desktop manager used by the GenealogyFrame.

8.1.3.1. DescendantView

This class group contains all the classes used by the descendant view. These classes include DescendantTreeNode and DescendantViewFrame. A DescendantTreeNode represents one node in the descendant tree. The DescendantViewFrame is responsible for building and drawing the descendant tree.

8.1.3.2. FamilyView

This class group contains all the classes used for the family view, as well as the dialogs used for adding children, and editing parents and the main couple. The FamilyViewFrame class is responsible for drawing the family view, and handling menu item selections made by the user. The AddChildFrame is used for adding new children from the child table options menu. The EditCoupleFrame is used for adding and editing parents, as well as editing the main couple itself.

8.1.3.3. ProgenyView

This class group contains all the classes used by the progeny view. These classes include ProgenyTreeNode and ProgenyViewFrame. A ProgenyTreeNode represents one node in the progeny tree. The ProgenyViewFrame is responsible for building and drawing the progeny tree.

8.1.3.4. SearchView

The SearchView class group contains only the SearchViewFrame class. This class is responsible for drawing the search screen, handling search requests, and displaying the results.

8.1.3.5. Wizard

The Wizard class group contains all the classes used by the start-up wizard. This group can be broken into sub-groups; classes used for maintaining the wizard itself, classes representing the wizard pages, and classes which are responsible for handling the results of each page.

The classes used for maintaining the wizard itself are mostly from the *Creating Wizard Dialogs with Java Swing*[6] article, with a few modifications. Additions were made to allow for validation of page data before allowing the user to proceed to the next page, repopulating the data on a previous page when the user clicks the Back button, holding the current state of the family tree in memory, and updating that tree as the user updates fields.

Classes with names ending in ‘Descriptor’ are responsible for creating the wizard page panel, validating the page data, storing the page data, and determining which wizard pages are next and previous.

The classes used for wizard page display are those ending in ‘Panel’. These classes are only responsible for display, and communicating with the page descriptor classes. The pages themselves fall into five categories; introductory, self, family member information, marriage information, and review. Since the pages for entering family member information are basically the same, they all inherit from the PersonalInfoPanel class which contains the drawing functionality. All the individual pages need to worry about is how to insert the final data into the tree in memory. Since all the marriage information pages are the same, and these pages do not need to worry about the tree structure as the personal information pages do, the same class, MarriageInfoPanel, is used for all the marriage panels. The descriptors are able to manage any specific processing for each page.

8.2. Overview of Basic Operations

8.2.1. Start-up

This section describes what occurs when the main application is started. The user is shown a login dialog. Success or failure of the login process is determined by whether the server can use that login information provided to contact the MySQL database, add tables as required, and add a new tree user record as required. If login succeeds, the main application window is opened. If the user has not already added individuals through the software in the past, the start-up wizard is opened automatically. Otherwise, the search screen is opened automatically.

- **GenealogyApp main()**

GenealogyApp main() creates a new GenealogyApp instance.

- **GenealogyApp constructor**

The GenealogyApp constructor creates a new ConnectFrame instance. The ConnectFrame is a login dialog with fields for username and password.

The user enters a username and password and clicks the OK button on the ConnectFrame.

The ConnectFrame calls the GenealogyApp contactServer() method with the username and password the user entered.

- **GenealogyApp contactServer()**

The GenealogyApp contactServer()method creates a new GenealogyServer instance, passing in the username and password to the constructor.

- **GenealogyServer constructor**

The GenealogyServer constructor attempts to connect to the FamilyTree database using

- Class.forName("com.mysql.jdbc.Driver");
- con = DriverManager.getConnection(m_strURL, m_strDBUserName, m_strDBPassword);
- where m_strURL is jdbc:mysql://localhost:3306/FamilyTree, and m_strDBUserName and m_strDBPassword are the username and password the user entered on the ConnectFrame.

The GenealogyServer constructor then attempts to create all the database tables needed by the application if the tables do not already exist.

The GenealogyServer constructor looks in the TreeUsers table for a record where the TreeUserName matches the username entered into the ConnectFrame. If a record is found, the TreeUserID is stored in a global variable, m_intUserID. If no such record can be found, a new TreeUser record is added for this user, and the TreeUserID generated by the identity column is stored in m_intUserID.

The Person table is queried for a count of records for this TreeUserID. If any records are found, the global variable m_blnUserHasData is set to true. Otherwise, it is left at false.

If the constructor makes it through all the steps above without error, the global variable

m_blnConnectionSucceeded is set to true. Otherwise, it will be left at false.

- **GenealogyApp contactServer()**

The GenealogyApp contactServer() method then calls the GenealogyServer getConnectionSuccessful() method which will return the state of GenealogyServer's m_blnConnectionSucceeded variable.

- If the result is false, an error message is displayed prompting the user to re-enter his username and password. When the user clicks OK, we start this flow over again from the ConnectFrame OK button click.

- If the result is true, the ConnectFrame will be set to invisible and a new GenealogyFrame instance will be created and set to visible.

  - The GenealogyServer getDataAlreadyExists() method is queried, which will return the state of the GenealogyServer m_blnUserHasData variable.
    - If getDataAlreadyExists() returns true, the GenealogyFrame method openSearchFrame() is called which tells the GenealogyFrame to open the search screen.
    - If getDataAlreadyExists() returns false, the GenealogyFrame method simulateWizardClick() is called to open the start-up wizard.

8.2.2. Opening the Progeny View

This section describes what happens when the user selects a progeny menu option. The first three levels of the tree are queried automatically, though only the first two levels are displayed when the form is first shown. The extra level is needed in order for the GUI to display nodes with children differently from those without children so the user can tell which ones he can expand.

- **FamilyViewFrame showProgenyView()**

The private FamilyViewFrame method showProgenyView() is called whenever the user selects a progeny view menu option. This method is passed the PersonID of the person for whom we're requesting a progeny view. The GenealogyFrame openProgenyView() method is called, passing in this PersonID.

- **GenealogyFrame openProgenyView()**

The GenealogyFrame method openProgenyView() is called, passing in the PersonID of the individual whose progeny view we're requesting.

The GenealogyServer reference is requested by calling getServerReference() on the GenealogyFrame's GenealogyApp reference. The GenealogyServer getPersonByID() method is called, passing in the PersonID from step 1.

A new ProgenyViewFrame instance is created, passing in the Person object returned from step 2.

- **ProgenyViewFrame constructor**

The ProgenyViewFrame constructor creates sets its m_objRootPerson variable to the Person passed in.

- **GenealogyFrame openProgenyView()**

The GenealogyFrame calls the ProgenyViewFrame initializeTree() method.

- **ProgenyViewFrame initializeTree()**

The ProgenyViewFrame initializeTree() method creates a new ProgenyTreeNode from the root Person saved in m_objRootPerson in step 8.2.2. This node is set to visible and expanded.

If getMotherID() or getFatherID() called on the root person returns a value > 0, the parent couple for this person is requested from the GenealogyServer by calling getCoupleByPersonID() passing in the ID of the mother or father.

If there is a father for the root person

- Create a new ProgenyTreeNode for the father, and add it to the progeny tree as a child of the root. This node is set to visible.
- Request the parent couple of the root's father from the server.
- If the father's mother is found, create a new ProgenyTreeNode for her and add it to the progeny tree as a child of the root's father. This node will be set to NOT visible and NOT expanded. The node for the father is set to NOT expanded.
- If the father's father is found, create a new ProgenyTreeNode for him and add it to the progeny tree as a child of the root's father. This node will be set to NOT visible and NOT expanded. The node for the father is set to NOT expanded.

If there is a mother for the root person

- Create a new ProgenyTreeNode for the mother, and add it to the progeny tree as a child of the root. This node is set to visible.
- Request the parent couple of the root's mother from the server.
- If the mother's mother is found, create a new ProgenyTreeNode for her and add it to the progeny tree as a child of the root's mother. This node will be set to NOT visible and NOT expanded. The node for the mother is set to NOT expanded.
- If the mother's father is found, create a new ProgenyTreeNode for him and add it to the progeny tree as a child of the root's mother. This node will be set to NOT visible and NOT expanded. The node for the mother is set to NOT expanded.

For each level of the tree, starting at the root, assign a column to each node. The root is always assigned the center column. Starting with the root, each node is assigned columns for its visible children, starting with the node in the center, and then the left and right, starting at the center and moving outward.

8.2.3. Opening the Descendant View

This section describes what happens when the user selects the descendant menu option. The first three levels of the tree are queried automatically, though only the first two levels are displayed when the form is first

shown. The extra level is needed in order for the GUI to display nodes with children differently from those without children so the user can tell which ones he can expand.

- **FamilyViewFrame showDescendantsView()**

The private FamilyViewFrame method showDescendantsView() is called whenever the user selects the descendant view menu option. This method is passed the PersonID of one member of the couple for whom we're requesting a descendant view. The GenealogyFrame openDescendantView() method is called, passing in this PersonID.

- **GenealogyFrame openDescendantView()**

The GenealogyFrame method openDescendantView() is called, passing in the PersonID of one member of the couple whose descendant view we're requesting.

The GenealogyServer reference is requested by calling getServerReference() on the GenealogyFrame's GenealogyApp reference. The GenealogyServer getCoupleByPersonID() method is called, passing in the PersonID from step 1.

A new DescendantViewFrame instance is created, passing in the Couple object returned from step 2.

- **DescendantViewFrame constructor**

The DescendantViewFrame constructor creates sets its m_objMainCouple variable to the Couple passed in.

- **GenealogyFrame openDescendantView()**

The GenealogyFrame calls the DescendantViewFrame initializeTree() method.

- **DescendantViewFrame initializeTree()**

The DescendantViewFrame initializeTree() method creates a new DescendantTreeNode from the root Couple saved in m_ objMainCouple in step 8.3.2. This node is set to visible and expanded.

Children of this couple are requested from the GenealogyServer by calling getChildCouplesByParentID (). A DescendantTreeNode is created for each of the child couples returned. Each node is added to the tree as a child of the root, and is set to visible.

For each child couple returned in the previous step, request their child couples from the GenealogyServer. A DescendantTreeNode is created for each of the child couples returned. Each node is added to the tree as a child of its parent, and is set to NOT visible and NOT expanded. Parent nodes are set to NOT expanded if child nodes are loaded for them.

For each level of the tree, starting at the root, assign a column to each node. The root is always assigned the center column. Starting with the root, each node is assigned columns for its visible children, starting with the node in the center, and then the left and right, starting at the center and moving outward.

8.2.4. Opening the Family View

This section describes what happens when the user opens a family view. The main couple (the couple we want the family view of), the main couple's parents, and the main couple's children are queried in order to populate the main couple, husband's parents, wife's parents, and children sections of the form.

- **GenealogyFrame openFamilyView()**

A new FamilyViewFrame instance is created, passing in the PersonID of the person we're requesting the Family View for.

- **FamilyViewFrame constructor**

The GenealogyServer reference is requested by calling getServerReference() on the GenealogyFrame's GenealogyApp reference. The GenealogyServer getCoupleWithParentsAndChildren() method is called, passing in the PersonID from above.

Panels for the main couple, husband's parents, wife's parents and child table are created and drawn.

Menu items are added for each panel's options button depending upon the contents of the panel.

- For the main couple menu
  - The Edit option is always visible
  - The Husband's Progeny menu option is available if the main couple has a male member.
  - The Wife's Progeny menu option is available if the main couple has a female member.
  - The Descendants menu option is always available.
- For the husband's parents menu
  - The Edit option is visible if the main couple has a male member
  - The Family View option is visible if the main couple has a male member and at least one of the husband's parents have been added.
- For the wife's parents menu
  - The Edit option is visible if the main couple has a female member
  - The Family View option is visible if the main couple has a female member and at least one of the wife's parents have been added.
- For the child table menu
  - The New Child menu option is always available
  - The Family View option is visible if there is at least one child in the child table.

8.2.5. Adding a Child to the Child Table

This section describes what happens when the user selects the New Child menu option on the child table at the bottom of the family view form. A new Person object is created, and passed to the add dialog to be populated from whatever information the user enters.

- **FamilyViewFrame showAddChild()**

The private FamilyViewFrame method showAddChild() is called whenever the user selects the New Child menu option. A new Person object is created, and passed in as the new child to be added. The

GenealogyFrame openAddChildFrame() method is called, passing in the new person object, and the main couple from the FamilyViewFrame is passed in as the parent couple.

- **GenealogyFrame openAddChildFrame()**

The GenealogyFrame openAddChildFrame() method is called, passing in a Person object as the new child, and the Couple object which will be the parents of this child.

The AddChildFrame showModalAddDialog() method is called, passing in the child Person object.

The add child dialog is opened from the GenealogyFrame instead of the FamilyViewFrame because the FamilyViewFrame inherits from JInternalFrame, and a Java bug that is not fixed until version 1.5 (currently the beta release) prevents JInternalFrame descendants from being opened as modal dialogs. Frames that are to behave like sub-windows of a main application must inherit from JInternalFrame in order to be displayed properly.[16]

- **AddChildFrame showModalAddDialog()**

showModalAddDialog() creates a new instance of the internal class AddChildDialog. The constructor is passed the child Person object. When the user clicks the Save button, the child Person object members are populated from the fields on the dialog, and the dialog return value is set to true. showModalAddDialog() returns the AddChildDialog return value.

- **GenealogyFrame openAddChildFrame()**

If the call to the AddChildFrame showModalAddDialog() method returns true, the newly populated Person is added to the system by calling the GenealogyServer method addNewPersonAsMyChild(), passing in the new Person object and the parent Couple object passed in by FamilyViewFrame showAddChild().

The return value from AddChildFrame showModalAddDialog() is returned to FamilyViewFrame showAddChild().

- **FamilyViewFrame showAddChild()**

If the return value from GenealogyFrame openAddChildFrame() is true, the child table is refreshed.

8.2.6. Adding or Editing Parents of the Husband

This section describes what happens when the user selects the Edit menu option on the husband's parents section of the family view form. The action taken depends upon whether parents have already entered for the husband or not. The steps taken for adding or editing parents for the female member of the main couple from the family view form are similar.

- **FamilyViewFrame showEditHusbandParents()**

The private FamilyViewFrame method showEditHusbandParents() is called whenever the user selects the Edit menu option on the husband's parents section of the family view form. The GenealogyFrame openCoupleEditFrame() method is called, with the parameters passed in depending upon whether either of the husband's parents have been entered yet.

If neither parent has been entered, a new Couple object is created and passed in, a Person object for the main couple husband is passed in so the Couple can be linked into the family tree, and an edit option of OPTION_ADD_NEW_COUPLE is passed in.

If only one parent has been entered, the existing Couple object for the husband's parents is passed in, no child Person object is needed since the Couple is already linked into the tree through the existing parent, a new Person object is passed in to hold the new spouse (if added), and an edit option of OPTION_ADD_TO_EXISTING_COUPLE is passed in.

If both parents have already been entered, the existing Couple object for the husband's parents is passed in, no child Person object is needed since the parent Couple is already linked into the family tree, and an edit option of OPTION_EDIT_COUPLE is passed in.

- **GenealogyFrame openCoupleEditFrame()**

A new EditCoupleFrame instance is created. The EditCoupleFrame showModalEditDialog() method is called, passing in the edit option, parent Couple object, and the new spouse Person object.

The edit couple dialog is opened from the GenealogyFrame instead of the FamilyViewFrame because the FamilyViewFrame inherits from JInternalFrame, and a Java bug that is not fixed until version 1.5 (currently the beta release) prevents JInternalFrame descendants from being opened as modal dialogs. Frames that are to behave like sub-windows of a main application must inherit from JInternalFrame in order to be displayed properly.[16]

- **EditCoupleFrame showModalEditDialog()**

showModalEditDialog() creates a new instance of the internal class EditCoupleDialog. The constructor is passed the edit option, the parent Couple, and the new spouse Person object.

When the user clicks the Save button

- If the edit option is OPTION_ADD_NEW_COUPLE
  - § A new Person object is created for each member of the couple for whom a first and last name was filled in.
  - § The members for each Person object are set based upon what the user typed in.
  - § The Couple marriage members (date and place) are populated if the user entered a marriage date or marriage place.
- If the edit option is OPTION_ADD_TO_EXISTING_COUPLE
  - § If a first and last name are entered for the new spouse, the new Person object is set for that spouse based upon what the user entered.
  - § The members for the existing spouse Person object are set based upon what the user typed in.
  - § The Couple marriage members (date and place) are populated if the user entered a marriage date or marriage place.
- If the edit option is OPTION_EDIT_COUPLE
  - § The Person members are updated according to what the user entered.
  - § The Couple marriage members (date and place) are populated if the user entered a marriage date or marriage place.

- The dialog return value is set to true.
- showModalEditDialog() returns the EditCoupleDialog return value.

- **GenealogyFrame openCoupleEditFrame()**

If the call to the EditCoupleFrame showModalEditDialog() method returns true

- If the edit option was OPTION_ADD_NEW_COUPLE, the GenealogyServer method addNewCoupleAsMyParents(), passing in the new Couple and the child Person object.
- If the edit option was OPTION_ADD_TO_EXISTING_COUPLE
  - If the new spouse Person was populated, we added a new spouse. Call the GenealogyServer addNewPersonToExistingCouple() method, passing in the new spouse Person and the Couple object.
  - If the new spouse Person was NOT populated, we just updated the already existing spouse. Call the GenealogyServer updateCouple() method, passing in the Couple object.
- If the edit option was OPTION_EDIT_COUPLE, the GenealogyServer method updateCouple() is called, passing in the Couple object.

The return value from EditCoupleFrame showModalEditDialog() is returned to FamilyViewFrame showEditHusbandParents().

- **FamilyViewFrame showEditHusbandParents()**

If the return value from GenealogyFrame openCoupleEditFrame() is true, the family view form is refreshed.

9. Results

The final question is whether or not this project succeeded in accomplishing the goals described in earlier sections of this report.

9.1. Data Entry

This project was to provide a means of entering family members that users would approve of. The best course of action seemed to be to emulate a data entry view that already had proven user backing.[1] The family view provided by Family Tree Maker[3] was chosen as the model view. This project was also to improve upon the initial user experience offered by Family Tree Maker[3] by including a start-up wizard.
The family view in this project as shown below is almost identical in form to the Family Tree Maker family view show in figure 3.1. The only difference is that the user can see parents of the main couple as well as children rather than having to open up a new screen to see them. Buttons next to each section allow the user to add and edit individuals through a separate add/edit dialog rather than refreshing the entire family view screen so that the couple being edited becomes the main couple. This should keep the user from being disoriented, and allow him to see the main couple he is adding relatives to while adding the new relatives.

Figure 9.1: Family view form

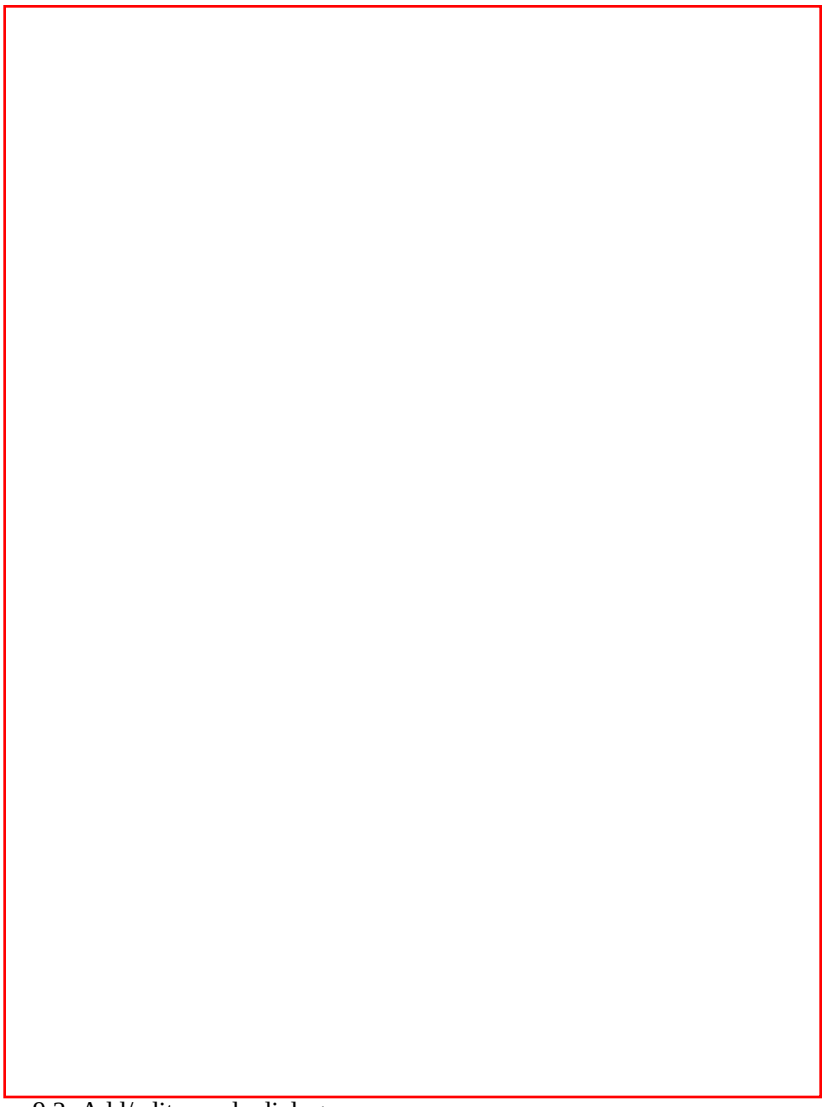

Figure 9.2: Add/edit couple dialog

This project also contains a wizard for adding the first few generations. This wizard is opened as soon as the user enters the application for the first time. It consists of an introductory page, a page where the user enters his own person information, pages for entering information about his parents, pages for entering information about his paternal and maternal grandparents, and a review page at the end. By being presented with an easy-to-follow wizard at startup instead of a blank family view screen, the user is better able to get up and running with the software right off the bat without any need for reading manuals, or hunting and pecking.
The wizard allows the user to go back as many pages as desired in order to fix mistakes. The appearance of pages in this wizard is dynamic so there won't be any danger of unlinked individuals or data being accidentally entered into the system. For example, if the user does not enter at least a first and last name for his mother, he will never see the pages for his maternal grandparents. If the user does not enter a first name and last name for either member of a set of parents, he will never see the marriage detail page for that couple.

9.2. Search View

This project was also to include a search screen which would allow the user to search for individuals by name, birth date and birth date range, and death date and death date range. The name search fields were supposed to allow wild card characters. The search results were to contain enough information to allow the user to distinguish individuals from one another without having so many columns that the results appeared crowded. Also, the user was supposed to be able to jump to the family view of any individual in the search results.
The search view in this project is shown below in figure 9.3. It allows the user to search by first, middle and last

name, and all name fields accept wild card characters. It also allows the user to search for a specific birth or death date, as well as a birth or death date range. The search results include the first name, middle name, last name, birth date and death date for each individual returned. This should be enough to allow the user to uniquely identify a particular individual without being overwhelmed with information. The user is able to jump to the family view of any individual in the search results by selecting the desired row and selecting the Family View menu option from the button next to the results table. The search results can be sorted by the user by double-clicking on a column header. If the user hovers over a column header with the mouse, a tool tip to that effect is displayed. Figure 9.4 shows the tool tip message, and the search results after being sorted by birth date.

Figure 9.3: The search screen with the search results shown below. Note that the name fields do allow the user to search by wild cards.

Figure 9.4: The same search results after double-clicking on the Birth Date column header. A tool tip telling the user how to sort is displayed when the user hovers over a column header with the mouse.

9.3. Tree View

The primary purpose of this project was to develop a tree view which would allow the user to see his relatives and see the relationships between relatives at the same time. This tree view was to be used to implement both a progeny view and a descendant view.

What a tree node represents is different depending on the particular view it is used for. For the progeny view, a node represents one individual. For the descendant view, a tree node represents a couple. Each tree node contains the name(s) of the individual(s) it represents, and is drawn large enough for the information within it to be easily read, and with a decent amount of space around it, allowing every individual in the tree to be easily identified. The only way the text can become too small to read is if the user deliberately zooms out on the tree.

The user can expand or collapse tree nodes by clicking on them with the left mouse button. Clicking on an unexpanded node opens one level of that node's children. Clicking on an expanded node hides all of that node's children. Nodes that have hidden children are easily distinguishable from nodes without hidden children by fill color. Nodes with hidden children will have a fill color, while nodes without hidden children will remain white. When a node is collapsed, the nodes in the level below it will rearrange themselves in order to bring themselves more to the center of the screen. Doing so makes better use of the available screen space.

Users can move the tree about the screen. Right-clicking inside a node moves the tree so that the selected node is in the center of the screen. This movement is animated so that the user does not lose his place as the tree moves. The user can also drag the entire tree by clicking outside of a node and dragging with the left mouse button.

The tree view also supports zooming. Users can zoom in on the tree by clicking the right mouse button outside of a node and dragging the mouse to the right, and can zoom out by clicking the right mouse button outside a node and dragging the mouse to the left.

As promised, the size of the tree is not restricted to the screen size. However, the size of the screen limits the number of generations and individuals per generation that can actually be seen at one time. At most seven rows and five columns can be seen without zooming out on the tree. Zooming out on the tree allows the user to see at most nine rows and eight columns at a time while still being able to make out the text within the node. However, it would seem that being able to show or hide pieces of the tree, and move the tree about the screen helps alleviate the limitation on how many nodes can fit on the visible portion of the view at a time.

In addition, when the tree is being drawn, only the information needed to draw the tree properly is loaded from the database. When the tree view is first drawn, the first three levels are loaded, with only the first two being displayed. The third level is used to determine how the second level is displayed. Nodes in the second level that have hidden children are shown in a different color than those without hidden children. Additional information is only queried from the database as the user expands collapsed nodes. The tree view only loads information as it is needed rather than loading a huge amount of data up front.

9.3.1. Progeny View

This project was to provide a progeny view that improved upon the Family Tree Maker[3] version shown in figure 4.1.1. The Family Tree Maker[3] progeny view shows the names of six generations of parents at a time, but has two downfalls. First, the view starts to look a bit crowded after the first few generations. Second, the view is not dynamic. There is no way to manipulate the view so that the user can focus on one section at a time. The best it offers is an arrow button.

The new progeny view is dynamic, and allows multiple generations to be displayed at once without the data being squished together. Each node in the tree represents one individual. The full name of the person and the person's birth date are both displayed in the node. Tree nodes representing female individuals can be readily identified by a different highlight color on the person's name.

The tree is drawn from the top down. The individual at the top is the child of the individuals in the nodes beneath him, and so on down through the levels of the tree. Each individual is linked to his parents by lines drawn between the nodes.

If a node has a non-white fill color, the person the node represents has at least one parent in the system that is not being displayed. Clicking on that node will expand it one more generation so that the immediate parents are displayed. Should the user wish to hide a section of the tree, clicking on an expanded node (one without a fill color) will hide all levels of parents of the person the node represents.
The user can move the tree and zoom in and out on the tree as described in the previous section.
A screen shot of the progeny view is shown below. Each node contains the name and birth date (if available) of the individual it represents. Lines are drawn to link each node to its parents. All the nodes with a blue fill color have child nodes which are currently not being displayed, while nodes without a fill color have been fully expanded. Clicking on an expanded node with children will cause all of its children to be hidden, and the fill color of the selected node will be reset to blue indicating its collapsed state.

Figure 9.5: Progeny view

9.3.2. Descendant View

The closest thing Family tree Maker[3] has to a descendant view is a printable report. Due to the sheer number of individuals that can be at any level of a descendant tree, the descendant view developed for this project must allow for manipulation of the tree in order to be useful. This project was to provide such descendant view through the use of a new tree view.
The purpose of the descendant view is to display a couple and all of their descendants and their spouses. Each node in the tree represents one couple, and contains each person's full name. The name of the male individual is drawn above the name of the female individual. Not all couples entered into the system will contain both individuals, either because the person is not married, or because the person is married, but the user does not know who the spouse is. Therefore, some nodes may only contain one name, the name of the related individual.
The tree is drawn from the top down. The node at the top contains the parents of every node below that is directly linked to it, and so on down the tree. Since each node represents a couple, and only one person in the couple is directly related to the parent couple, there must be some way for the user to distinguish the individual related by blood from the individual related by marriage. The descendant tree view solves this problem by highlighting the names of related individuals in yellow.
If a node has a non-white fill color, the couple the node represents has at least one child in the system that is

not being displayed. Clicking on that node will expand it one more generation so that the immediate children are displayed. Should the user wish to hide a section of the tree, clicking on an expanded node (one without a fill color) will hide all levels of children of the couple the node represents.
The user can move the tree and zoom in and out on the tree as described in the previous section.
A screen shot of the descendant view is shown below. Each node contains the full names of both individuals, provided that both individuals have been entered. Lines are drawn to link each node to its children. All the nodes with an orange fill color have children which are currently not being displayed, while nodes without a fill color have been fully expanded. Clicking on an expanded node with children will cause all of its children to be hidden, and the fill color of the selected node will be reset to orange indicating its collapsed state. The name of the individual in each node who is related to the parent node by blood instead of marriage is highlighted in yellow.

Figure 9.6: Descendant view

9.4. Database Design

The fitness of the database design chosen for this project is determined by whether it allows for future development without being re-vamped, whether the database schema itself has a reasonable degree of normalization, and whether the retains the non-additive join property.[17]

9.4.1. Ease of Future Development

The database design chosen for this project, and the selection of MySQL as the database provider does allow for the future expansion of this project. By keeping the attributes of people separate from the relationships between them, and by breaking down the relationships as much as possible, the database design allows for future developers to enhance this project to handle multiple marriages per individual without a database overhaul. Marriage information is kept separate from Person information, as are parent-child relationships. Marriage information is stored in a separate table, containing the PersonID of the husband and wife, along with the marriage date and marriage place. Parent-child relationships are stored in a separate table, keyed by the PersonID of the child, and containing the PersonID of the mother and father of that child. The structure of the Marriage table does not exclude multiple marriage, and the structure of the ParentChild table does not limit a parent to only being able to have children with one individual.
The choice of MySQL as the database, the isolation of all database interactions into one class, as well as the structure of the Person table allows for future developers to make this project into a true client-server

application. MySQL itself allows for concurrency control. The Person table structure already allows for multiple users since it contains a user ID field. In addition, all database interaction is contained within the GenealogyServer class. The server class could be moved to another machine without much difficulty. The most difficult part of the process would be adding concurrency control to the client.

9.4.2. Schema Fitness

9.4.2.1. Normalization

In order for a database schema design to be a good one, it should generally be normalized to Boyce-Codd Normal Form, unless doing so would cause major performance issues.[18] When a schema is not normalized, it is likely that there is redundant data lurking about. Redundant data is a problem for two reasons. First, repeated fields take up more space. If adding an object to a database involves inserting into three tables, and one extra field is unnecessarily repeated in each of those tables, that's two fields worth of memory being wasted for every new object. Second, redundant data can lead to update anomalies. For example, if I need to modify the redundant field mentioned above, I need to be sure to update it in all three places or my data becomes corrupt.

First Normal Form

The first step in determining if a schema is in Boyce-Codd Normal Form is determining whether the schema is in First Normal Form. For a database schema to be in First Normal Form, there should be no compound fields in the database. For example, a table containing two fields, CompanyName and Loactions, where the Locations field contains a list of locations for the company, would not be in First Normal Form. The database schema for this project as described in section 6.2.1 does not have any compound fields.

Second Normal Form

Next, the schema needs to be analyzed to see if it is in Second Normal Form. In order for a schema to pass the Second Normal Form test, the schema must meet the requirements for First Normal Form, plus no tables can contain fields which can be uniquely determined by a fragment of the primary key. Since only one table, Marriage, has a compound primary key, there is only one table to analyze. The primary key of the Marriage table is made up of the HusbandID and WifeID. The only other fields in the table are MarriageDate and MarriagePlace. So, the question is, can the HusbandID or WifeID alone determine the MarriageDate or MarriagePlace? The software itself only allows one marriage per person, but the database needs to allow for the future handling of multiple marriages. Given that information, there is no way that a HusbandID or WifeID alone could uniquely determine the MarriageDate or MarriagePlace fields. The combination of the two is required. Therefore, the schema is in Second Normal Form.

Third Normal Form

Next, the schema needs to be analyzed to see if it is in Third Normal Form. In order for a schema to pass the Third Normal Form test, the schema must meet the requirements for Second Normal Form, plus there can be no non-primary key fields which depend upon any other non-primary key fields for their existence. For example, suppose we had a table with the fields CustomerName, City, State and ZipCode, where the CustomerName is the primary key. This table is not in Third Normal Form because a unique City and State combination is determined by a ZipCode, not by the CustomerName. Does the Marriage table meet the requirements of Third Normal Form? Can the MarriagePlace uniquely determine a MarriageDate, or vice versa? The answer is no, so the Marriage table passes the

test.
Does the ParentChild table meet the requirements of Third Normal Form? Can the MotherID uniquely determine the FatherID, or vice versa? Well, the software restricts marriages to one per person. However, the real world does not have this restriction, and it would be better for future development if the database design represented the real world even if the software currently does not. Therefore, a MotherID cannot uniquely identify the FatherID, and a FatherID cannot uniquely identify a MotherID. The ParentChild table passes the test.
How about the Person table? Are there any non-key fields that uniquely determine other non-key fields? There do not appear to be any, so the Person table passes the test.
Lastly, how about the TreeUsers table? The TreeUsers table contains two fields, the TreeUserID and the TreeUserName. The TreeUserID is the primary key, but the TreeUserName is described as also being a unique field. Therefore, this table does not actually contain any non-key fields, and the table passes the test.

Boyce-Codd Normal Form

Next, the schema needs to be analyzed to see if is in Boyce-Codd Normal Form. In order for a schema to be in Boyce-Codd Normal Form, it must be in Third Normal Form, and every field (could be compound) which can uniquely determine another field must be a candidate key for the table.
The TreeUser table passes this test because the TreeUserName is a candidate key.
The only fields in the Marriage table which can uniquely determine any other fields in the table are the HusbandID and WifeID taken together. (Remember that in the real world a HusbandID does not uniquely determine a WifeID, as was discussed above.) Since this is the primary key, the Marriage table passes the test.
Since the ParentChild table is intended to model the real world, a MotherID cannot uniquely determine a FatherID, or vice versa. Neither can a MotherID or FatherID alone uniquely identify one ChildID, since people are not limited (at least not normally) to having one child.
The Person table does not contain any field or combination of fields that can uniquely identify any other fields in the table, so the Person table also passes the test.

9.4.2.2. Non-additive Join

Since we know that the schema is indeed in BCNF, we now need to make sure that joins between the tables will not cause extra rows to be created. Extra rows, also known as spurious tuples, are a result of poor choices when dividing the data fields into their respective tables.
Suppose we started with the following database table.

StudentID	CourseNumber	RoomNumber	Instructor
1	A	A1	1
1	B	A2	1
2	A	A1	2
2	B	B3	2

Then, suppose we decided to break up this information into the two tables below.

StudentID	CourseNumber	RoomNumber
1	A	A1
1	B	A2
2	A	A1
2	B	B3

Instructor	RoomNumber
1	A1
1	A2
2	A1
2	B3

Now, suppose we want to join the tables back together with a natural join (on RoomNumber). We end up with two extra rows, and we can no longer determine with certainty which instructor student 1 had for course A, or which instructor student 2 had for course A. This situation is called an additive join.

StudentID	CourseNumber	RoomNumber	Instructor
1	A	A1	1
1	A	A1	2
1	B	A2	1
2	A	A1	1
2	A	A1	2
2	B	B3	2

This is the situation we want to avoid. Dividing up the tables so that the field linking the two new tables together is not a really a primary key or foreign key of either table is not a good idea. We should be able to join the tables in the schema by primary keys and foreign keys without generating extraneous rows.

The TreeUserID field is the primary key of the TreeUsers table, and a foreign key into the Person table. Common sense tells us that there is no way that joining the Person table with the TreeUsers table on Person.TreeCreatorID and TreeUsers.TreeUserID could create extraneous rows because TreeUserID is a true primary key of the TreeUsers table. Each TreeUserID can only occur once in the TreeUsers table.

Marriage.HusbandID and Marriage.WifeID are both foreign keys tying into the Person. PersonID field. Joining the Marriage table on either the HusbandID or WifeID will not cause extraneous records to appear because PersonID is a true primary key into the Person table. Each PersonID can occur only once in the Person table.

ParentChild.ChildID, ParentChild.MotherID, and ParentChild.FatherID are also foreign keys referring to Person.PersonID. Joining the ParentChild table with the Person table will not cause extraneous records to appear either.

10. Conclusion

The Results section above indicates that the goals set out for this project have been achieved. The family view implemented for this project is similar in format to the Family Tree Maker[3] family view, with the addition of a section for each set of parents. There is a search view from which the user can jump to the family view of anyone in the results. A start-up wizard makes first-time use less stressful for the user. A new tree view developed through the use of the Piccolo 2D Graphics Toolkit[8] allows the user to expand and collapse sections of the tree, move the tree about the screen, and zoom in and out. The nodes are drawn so that the user can easily determine who the node represents, and which nodes may be expanded further.

The database structure chosen for this project represents the real world as closely as possible, so future development will not necessitate a database overhaul. The database schema already supports multiple marriages per individual,

and multiple users. The fact that the database schema can be shown to be in Boyce-Codd Normal Form, and the fact that natural joins among tables in the schema will not cause the creation of spurious tuples indicate that this database design is solid and stable.

11. Future Work

The main problem with attempting to display family tree data is still the limitation of screen size. Even with manipulation options offered by the tree view developed for this project, it would be difficult for a user to get an idea of the structure of the entire family tree, while still being able to read the information within the nodes. One possible way to alleviate this problem would be add a sort of tool tip to the nodes so that the node content could be displayed when the user hovers over the node with the mouse. This way, the node text could still be accessible even if the node itself was way too small to read.

The ability to import images and include them in some of the views would also be a nice addition.

Another nice feature would be the addition of add/edit detail options to the progeny and descendant tree nodes. A user may prefer to interact with the tree instead of having to go back and forth to the family view form all the time. It would be nice if the user could add children from the descendant tree instead of only being able to add children from the family view screen. Similarly, it would be nice if the user could add parents from the progeny tree view.

The ability to add multiple marriages per person would be another nice feature. The database wouldn't need to be changed, but the client would need some re-working. For one thing, there would need to be an intermediate screen containing a list of possible marriages between the search results and the family view screen, or the search results would need to return couple instead of individuals. The assumption that a couple can be uniquely determined by one individual would no longer be valid. The logic used for integrating individuals into the tree would need to be updated to always include the IDs of both the mother and father.

Some new search options would also be nice to have. Possible additions would be the ability to search by marriage date range, search for siblings, or search for descendants. Maybe the last two could be menu options off the search results?

The last, and most complex, addition would be to make this a true client-server application. The database would allow for it as it currently stands. The server is already self-contained. The client would be a bit more complicated to deal with. Since there don't seem to be any family tree software products that allow for multiple users per install, let alone any that are client-server applications, it seems unlikely that there is much demand for a true client-server family tree application. This would be a good exercise, but perhaps not one that is really worth the effort.

12. References

1. 2005 Genealogy Software Report
http://genealogy-software-review.toptenreviews.com/

2. Legacy web site, containing a detailed product overview
http://www.legacyfamilytree.com/

3. Family Tree Maker web site, containing a product overview and demo
http://www.familytreemaker.com/

4. Ancestral Quest web site, containing a product overview
http://www.ancquest.com/

5. MySQL Developer Web Site

http://dev.mysql.com/

6. Eckstein, Robert, *Creating Wizard Dialogs with Java Swing*, Sun Developer Network, February 10, 2005, http://java.sun.com/developer/technicalArticles/GUI/swing/wizard/index.html

7. Baldwin, Richard G., *Using JDBC with MySQL, Getting Started*, Java Programming Notes #662, 2004, http://www.developer.com/java/data/article.php/3417381

8. Piccolo Home Page
http://www.cs.umd.edu/hcil/piccolo/download/index.shtml

9. Piccolo API Documentation
http://www.cs.umd.edu/hcil/piccolo/learn/piccolo/doc-1.1/api/

10. Plaisant, Catherine, Grosjean, Jesse, Bederson, Benjamin. *SpaceTree: Supporting Exploration in Large Node Link Tree, Design Evolution and Empirical Evaluation*, http://www.cs.umd.edu/local-cgi-bin/hcil/rr.pl?number=2002-05

11. GEDCOM summary and file format description.
http://www.cyndislist.com/gedcom.htm#Defined

12. Hortsman, Cay S., Cornell, Gary, Core Java 2 Volume I – Fundamentals, Sun Microsystems, Inc, Palo Alto, CA, 2003

13. Hortsman, Cay S., Cornell, Gary, Core Java 2 Volume II – Advanced Features, Sun Microsystems, Inc, Palo Alto, CA, 2000

14. Eckstein, Robert, Loy, Marc, Wood, Dave, Java Swing, O'Reilly & Associates, Inc, Sebastopol, CA, 1998.

15. Java API Documentation
http://java.sun.com/j2se/1.5.0/docs/api/

16. *Creating How to Use Internal Frames*, The Java Tutorial,
http://java.sun.com/docs/books/tutorial/uiswing/components/internalframe.html

17. Elmasri, Nevathe, Fundamentals of Database Systems, Second Edition, Addison-Wesley Publishing Company, Menlo Park, CA, 1994, pp 407-408.

18. Elmasri, Nevathe, Fundamentals of Database Systems, Second Edition, Addison-Wesley Publishing Company, Menlo Park, CA, 1994, pp 416-417.