Capstone Report: *Savvy*, a Statement-Oriented Information Manager

CST 499 Computer Science Capstone

Julie Asato, Matthew Robertson and Raymond Talmage

June 19, 2020

Executive Summary

Information and data in all its varying forms are an integral part of contemporary life. Whether you are a student, working or simply managing your daily life; you undoubtedly expect to recall and utilize a variety of unstructured information down the line. To combat becoming overwhelmed with incoming material, people devise their own strategies of organization; through a digital medium or pen and paper. Currently, the typical software solution involves a prose-oriented approach. These can work satisfactorily for simple knowledge bases and on topics where you already have an adequate understanding of the domain. Yet if you are managing information for a complex, unfamiliar topic, they can fall short. On the other side of the spectrum, qualitative analysis software tools exist for information management professionals, but these tools have a significant barrier for entry for the average consumer. Not everyone has the skills, time or money needed to use those products effectively; though a great many people would benefit substantially if they could.

To bridge the gap between the readily available and accessible consumer solutions and the hefty enterprise solutions, Savvy steps in as a statement-oriented information manager. It assists students and professionals with organizing their information in ways that are simple and effective without the difficulties associated with using professional-grade software.

Table of Contents

List of Tables

List of Figures

Introduction

Project name and description

Savvy is a groundbreaking information manager; aptly named because it describes both its users and its primary objective:

NOUN savvy

Shrewdness and practical knowledge; the ability to make good judgments.

ADJECTIVE savvier, savviest

Shrewd and knowledgeable; having common sense and good judgment.

Well informed about or experienced in a particular domain.

VERB savvies, savvying, savvied

Know or understand. (Lexico.com, 2020)

It is for anyone who has unstructured information they wish to manage. They may be students, working professionals or hobbyists focusing on a knowledge domain. Frequently these groups receive their information in loose unstructured formats like spoken word, in sections of prose, as outlines and in lists. Savvy is designed to reform such information to be effortlessly managed and accessed; as the ease and swiftness of one's access to information are linked to overall productivity. As a student or professional, if you invest significant time in seeking a specific bit of information instead of working with it, you can miss deadlines—or at least have less time for more enjoyable endeavors.

Problem and/or issue in technology

Whether you are a student, working or simply managing your daily life; you undoubtedly expect to recall and utilize a variety of unstructured information down the line. To combat becoming overwhelmed with incoming material, people devise strategies of organization; whether it be through a digital medium or pen and paper. Currently, the typical software solution involves a prose-oriented approach. These can work satisfactorily for simple knowledge bases and on topics where you already have an adequate understanding of the domain. Yet if you are managing information for a complex, unfamiliar topic, they can fall short.

On the other side of the spectrum, specialized software tools exist for information management professionals, but these tools have a significant barrier for entry for the average consumer. Not everyone has the skills, time or money needed to use those products effectively; though countless people would benefit substantially if they could.

Solution to the problem and/or issue in technology

To bridge the gap between the readily available and accessible consumer solutions and the hefty enterprise solutions, we have produced a simple statement-oriented information manager. With savvy, the user inputs statements and subsequently views/filters those statements. To preclude issues arising from prose-oriented approaches, statements are made in the form: Subject | Relationship | Object—which is simple to understand and offers a substantial power for searching and filtering. After the user has entered their data, they may search in the view pane of the application. When filtering the statements in the view, the user may enter one or multiple terms. In the multi-search case, intermediate statements are shown if they are part of a transitive relationship.

Example—search for: car, traction

car | has | tires

tire | provides | traction

Searching through notes made in this form can yield more concise and relevant results than manually searching and scanning through an entire document.

Project Goals and Objectives

Table 1

The Savvy Project's Goals and Objectives

Goals	Objectives
● Quickly take notes and find relative information in a personal application. ● Address the shortcomings of common, affordable software without introducing the complexities of professional-grade software.	● Create a input line for data entry ● Add a corresponding view pane to display entered statements ● Implement search functionality for the view pane to filter and show relevant information to the user ● Research and setup an embedded graph database for data storage and retrieval
● Empower users on most common platforms: Apple's MacOS, Linux distributions with GUI and Microsoft Windows.	● Use a cross-platform compatible GUI framework

Community and Stakeholders

Note-takers

Savvy is incredibly useful to those in environments where they are expected to consume and digest large amounts of unstructured information—this includes students and professionals that routinely deal with unstructured information. Although these would likely comprise the majority of our clients, this tool could gain broader use beyond note-takers. Put simply: our end-user is anyone with access to a computer and info they wish to store and effectively search simple pieces of knowledge on.

These people gain much needed efficiency in both creating and searching their notes. Additionally, they are able to review a subject more effectively by employing the filter capabilities of the application. They can both find and learn from their notes more easily, quickly and thoroughly without having to continuously groom their older notes.

Risks for potential users include time invested learning the application; we expect most will be comfortable using it after thirty minutes. As with most note-taking applications, many users will want to migrate existing knowledge bases into this system. Since no automated method for importing notes from other systems exists, this process will have to be done manually; which could take lots of time. Furthermore, the system is currently designed to only handle Subject | Relationship | Object triples. If the user requires more complex statements they will be foiled. Last, as with any new software project, it could contain bugs that result in loss of valuable data; Beware!

Developers of the Application

The developers gained skills related to product development and software engineering. With an Agile software development methodology, they worked with focus group testers to build a comprehensive application. Those developers acquired experience with the project management tools in GitHub to assign work, prioritize features and perform code reviews. Ultimately they succeeded in transforming innovative ideas into real-life products by merging open source libraries with hand-written code to produce a functional GUI, an application core and graph database.

Evidence that the proposed project is needed

Storing, managing and searching through information has been vital well before computer systems were tasked to aid in the process. Now, many tools and resources exist for that purpose. Our information manager aims to address the shortcomings of what is currently available, while avoiding adding unnecessary complexity that professional level applications have.

Two prevalent prose-oriented approaches are personal wikis and store-everything notebooks e.g. Microsoft OneNote, Apple Notes or Evernote. Such software allows you to search for some text or a topic and read an excerpt that may or may not contain the sought material. This can quickly devolve into a time-consuming series of searches and skimming text. Additionally, when adding to these information managers it is easy to introduce duplicate or even conflicting details. To prevent that, the stored content must be constantly reviewed and revised; a time-consuming process.

Contrary to these accessible but simple solutions, there are specialized tools within industry—programs like ATLAS.ti, QSR NVivo and Stanford Protégé. These can fit the information management use-case nicely, but they have some disadvantages. They all have steep learning curves, both in their conceptual models and in their application usage. Additionally, the dedicated qualitative analysis tools are expensive and their feature-sets are beyond what is needed for this project's basic use-case.

Feasibility Discussion

Environmental Scan/Literature Review

Prose-oriented Approaches

Evernote is a comprehensive store-everything notebook. It touts an impressive feature set including optical character recognition, cloud storage and syncing across multiple devices. Evernote has three pricing tiers: a free basic option, a premium offering at $7.99 per month and business level at $14.99 per user per month. Of interest to this project, Evernote provides several search and filter functionalities. These include the typical search phrase and exact match features. Additionally, Evernote allows for searching against a robust set of metadata fields; including notebook, tags, timeframe and even geo-location information from when the note was created (Evernote Corporation, 2018).

While these searching capabilities are powerful, they detract the user from the task at hand. In lieu of directly retrieving the sought information, users have to spend time remembering when the note was written and which tags they put on it. Also, if those notes each have many tags, then instead of not retrieving any results, an overwhelming number are returned.

Professional Qualitative Analysis Tools

ATLAS.ti is an enterprise level qualitative analysis tool. According to the official quick tour document: "[ATLAS.ti] lets you extract, categorize and interlink data segments from a large variety and volume of source documents. Based on your analysis, the software supports you in discovering patterns and testing hypotheses. With numerous output options and collaboration tools, your analysis is easily accessible to yourself and others" (Friese, 2019). Being a professional tool, ATLAS.ti is expensive, costing $1,840.00 for a single non-commercial license (cleverbridge, Inc., 2020).

As an application, much of ATLAS.ti's focus is source documents and maintaining their fidelity. This is critical when cross-checking analysis against sources to verify that a set of conclusions are sound. However, to accommodate that key facet of its workflow comes a difficult learning curve and major time investment in operating the application. In our project we assume the user is the de-facto source of information and they can invent facts and modify them as they see fit. This reduces the learning curve and usage overhead for users significantly.

Stanford Protégé is a free, open-source ontology development environment funded by sustaining grants from the National Institutes of Health. It began as a way to structure electronic knowledge bases and design better ones (Musen, 2015). It is presently used for managing terminology, visualizing relationships between terms (Rubin et al., 2007).

To appreciate how Protégé relates to our project we need to understand what an ontology is. Principally, ontologies are lists of entities with specific attributes and have related terms linked together thereby constituting the model. In Protégé, the terminology is a bit different. Ontologies are called frames. The entities within frames are called classes. The attributes are slots and facets—an additional concept—representing characteristics of the slots. A knowledge base in Protégé includes frames and their relationships (Rubin et al., 2007). To illustrate, suppose there is a dog class. That class has a brown fur slot as well as facets of thin fur, soft fur and clean fur. Then, a knowledge base would be multiple dogs with slots and facets for those slots.

Protégé is an excellent tool for exploring knowledge domains and examining their completeness and cogency. Our project, while it uses structures similar to ontologies, it does not need to evaluate the cogency of the dataset. Moreover, the conceptual model for Savvy is straightforward, so its learning curve is orders of magnitude lower than Protégé's three-day Short Course for beginners (Stanford University, 2016).

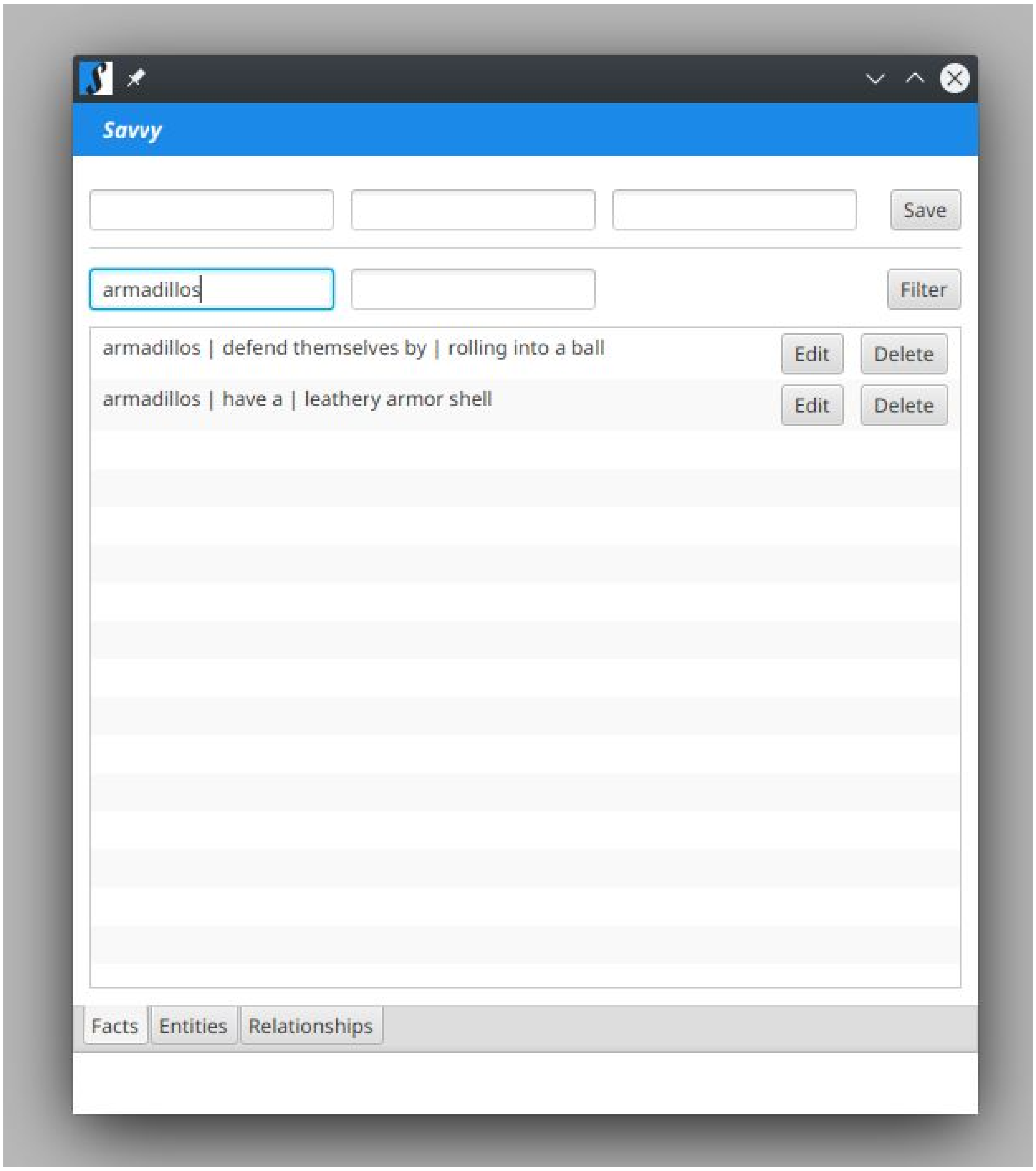
Design Requirements

Functional Decomposition of the Project

Central to this application's purpose is creation and saving Facts. In Savvy a Fact is composed of a Relationship and two Entities, a Subject and an Object. After creation, the user requires the ability to filter Facts related to a particular Entity. This allows the user to view only Facts that are relevant to their current focus. Figure 1 shows the application UI with some facts already created. Cat has been entered in the filter field; the Facts list shows all cat-related Facts.

Figure 1

Filter Facts Functionality



armadillos | defend themselves by | rolling into a ball

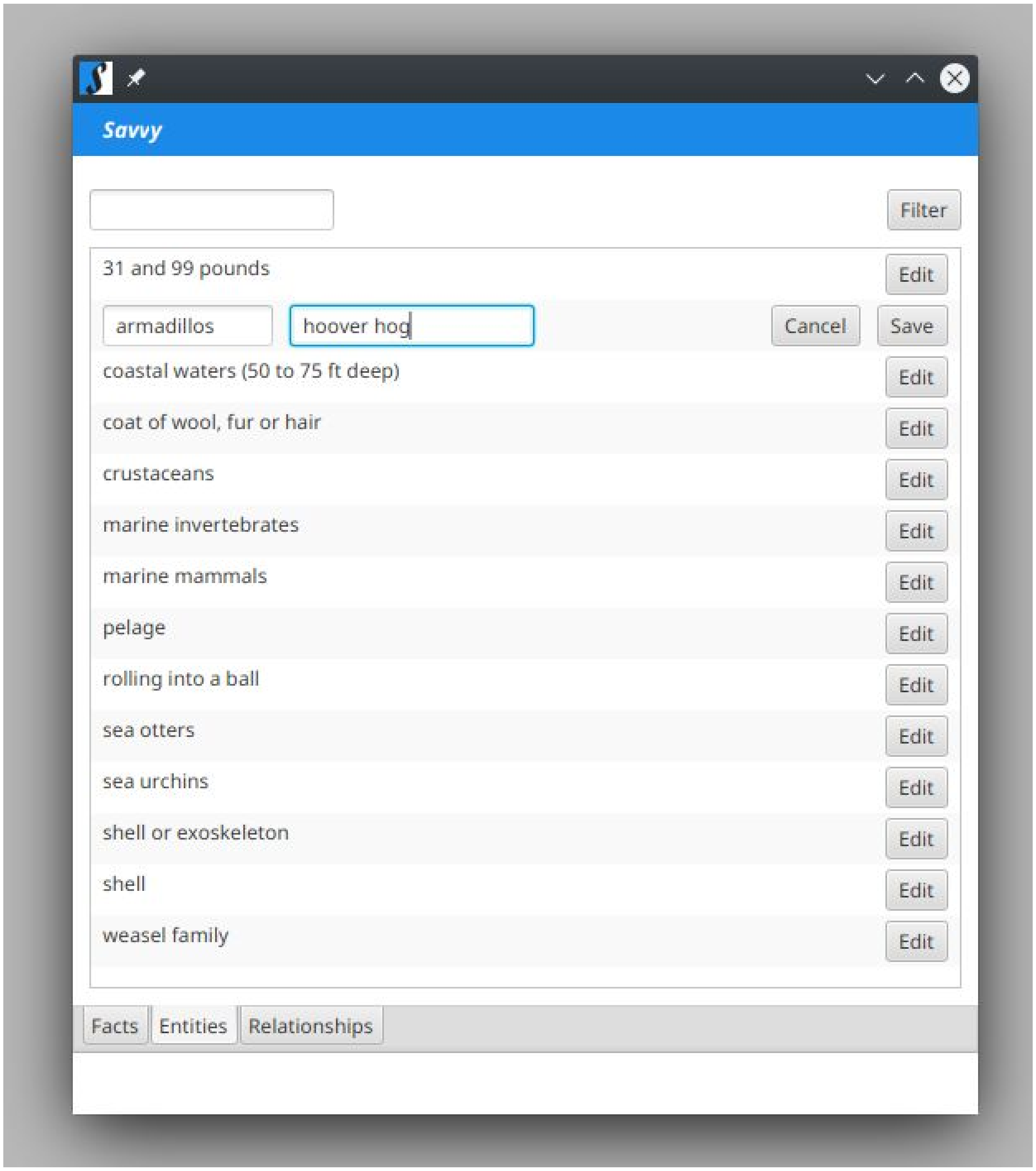
armadillos | have a | leathery armor shell

To allow the critical Fact creation and filtering functions to work in a manner that people can be comfortable using, several supporting features are required. The ability to edit and delete Facts is important; a user could make a mistake or want to update their facts based on new information. Users also tire of entering the same Entities or Relationships repeatedly. So wherever applicable, the application offers an autocomplete list for already entered items. In terms of fact expression, human languages—like English—have vast expressive power and complexity. For the product to feel natural to use, some of that complexity must be allowed without compromising the application's core functionality. To support that, Entities have Aliases and Relationships have Correlates. Additionally, each can have Modifiers applied in particular facts.

Many common terms can be used to refer to a single Entity, these terms, in Savvy these are called Aliases. Fundamentally, Aliases exist for users to naturally express Facts and refer to Entities. Instead of awkwardly writing a Fact as: *cat* | *eat* | *fish*, one can phrase it naturally as: *cats* | *eat* | *fish*. Likewise when seeking information about a topic we oftentimes search using a synonym; Aliases allow for filtering facts without having to remember which term was actually used when referring to the Entity when the Fact was created. Figure 2 shows the Entities tab with Aliases being edited for the car Entity.

Figure 2

The Entities Tab: Adding Aliases to an Entity



Savvy

Filter

31 and 99 pounds — Edit

armadillos | hoover hog — Cancel | Save

coastal waters (50 to 75 ft deep) — Edit

coat of wool, fur or hair — Edit

crustaceans — Edit

marine invertebrates — Edit

marine mammals — Edit

pelage — Edit

rolling into a ball — Edit

sea otters — Edit

sea urchins — Edit

shell or exoskeleton — Edit

shell — Edit

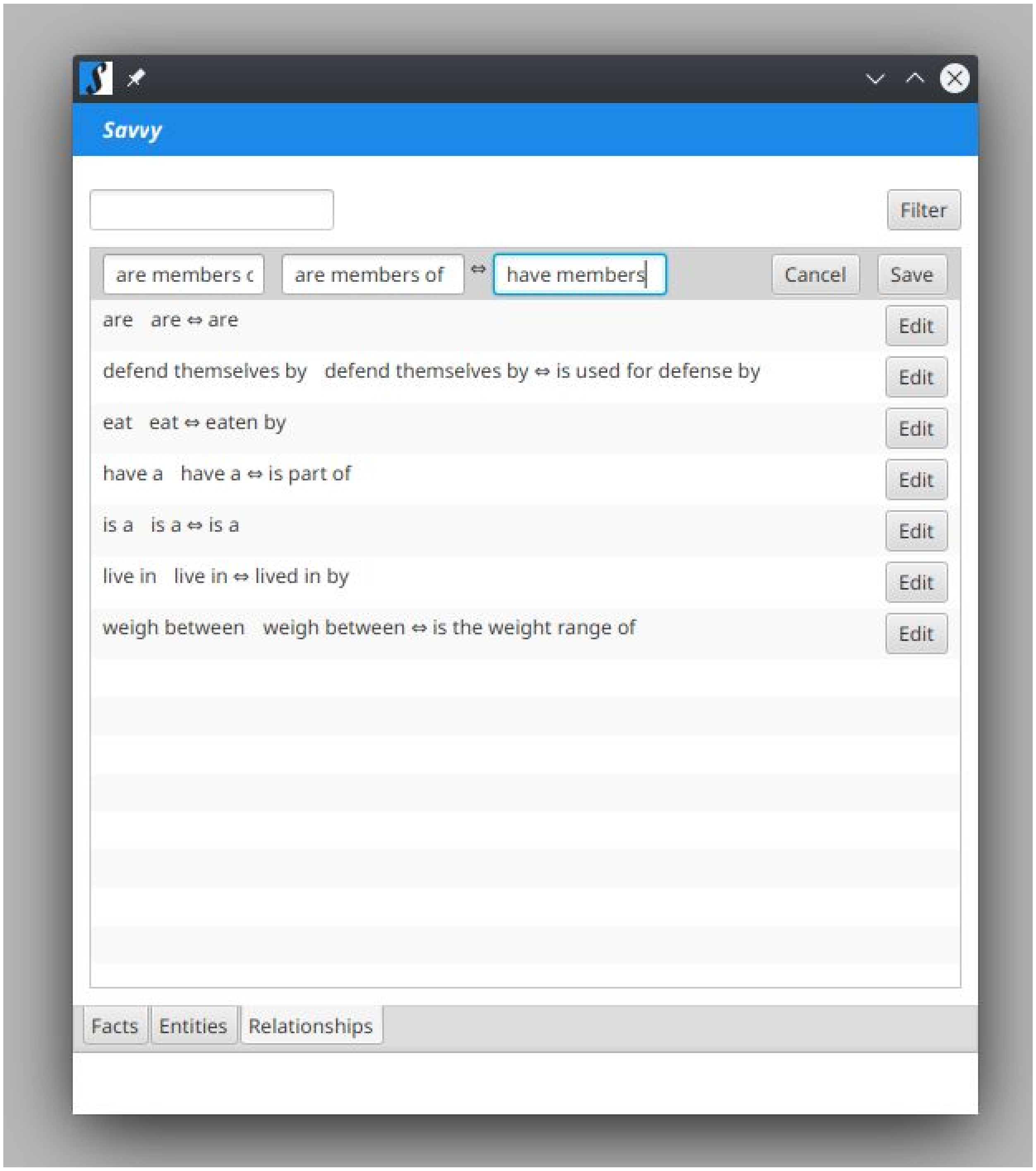
weasel family — Edit

Facts | Entities | Relationships

As Entities have Aliases, Relationships have a similar concept: Correlates. Unlike Aliases, Correlates have a direction associated with them allowing users to alternate phrasings of facts between their subject and object positions. For example: *cats* | *are chased by* | *dogs* can be used instead of *dogs* | *chase* | *cats*. Figure 3 demonstrates Correlate editing in the Relationships tab.

Figure 3

Edit Relationships and Correlates Functionality



are members c | are members of ⇔ have members

are	are ⇔ are	Edit
defend themselves by	defend themselves by ⇔ is used for defense by	Edit
eat	eat ⇔ eaten by	Edit
have a	have a ⇔ is part of	Edit
is a	is a ⇔ is a	Edit
live in	live in ⇔ lived in by	Edit
weigh between	weigh between ⇔ is the weight range of	Edit

Selection of Design Criterion

A fundamental criterion for the application is to be accessible to as many users as possible. We expect this sort of program requires a keyboard to operate, so mobile devices are out of scope for this project. The next most prevalent platforms are laptop and desktop computers (Windows, Linux, Mac), so we expect the application to run on those operating systems. Beyond maximizing the potential user base, this application requires user interactivity, so it must be user-friendly.

For user friendliness the application should:

- Be quick and easy to use.
- Be free of bugs that would destroy user data.
- Be responsive without major lags of hangs.
- Take up a reasonable amount of disk space (~250 MB)
- Be able to store thousands of facts, entities and relationships.
- Be a suitable substitute for traditional note taking approaches.
- Be able to effectively search information.
- Have excellent keyboard navigation and hotkey functionality.

Final Deliverables

A standalone desktop application along with a supporting user guide. The guide includes a brief background and overview, a step-by-step usage tutorial and a frequently asked questions section. The application's source code is available in a public git repository at https://github.com/csumb-serious-business/savvy. Also, documents outlining focus-group feedback and planned enhancements are included in this report's addendum. Last, an informational video was also produced for the Computer Science Capstone Festival explaining the project's purpose and demonstrating its functionality.

This project has no official clients; we worked with a focus group instead. The focus group approved the application chiefly by saying they plan to use it instead of whatever they were using before. Because they were able to enter, view, filter information within the application and the filter produced their anticipated results.

Approach and Methodology

- Defined Minimum Viable Product (MVP).
    - Researched comparable software to inform feature-set.
    - Created rough mockup of GUI.
    - Created rough design of data model.
- Researched applicable technologies to implement GUI, database and application core.
    - Researched JavaFX and alternatives for GUI creation.
    - Researched Neo4j graph database and its alternatives.
- Used Agile development to create MVP.
    - Planned iterations in weekly meetings according to most valued features.
    - Used GitHub for code reviews.
    - Met with potential users to elicit desirable enhancements.
    - Tracked tasks and issues in GitHub.
- Developed features beyond MVP, using the same methodology as above.

Ethical Considerations

For this project, no human subject research was conducted. Since its end product is a standalone application running on an individual's personal computer, users are its sole evaluators; they have assessed its fitness against potential issues they faced when operating and administering the application. We make no guarantees of the application's security or its ability to preserve their data—it could be regarded as a developer's ethical responsibility to ensure application security and data consistency in the products they create.

During development of the project, there were a few potential avenues for ethical lapses. For the developers, a fair division of labor was pursued so all participants can enjoy a satisfying learning experience. For the application testers, reasonable accommodations were made to ensure they can follow and complete their testing tasks. Furthermore, we warned possible testers against using this software with sensitive real-world information.

After the application was deployed, some natural ethical concerns arose. However, those were either systemic—beyond this project's capacity to address—or could be ameliorated with future enhancements. Clearly, those without access to computers that can run the application are unable to benefit from this project. For example, schools increasingly provide students with Google Chromebooks instead of full-blown laptops to minimize cost; those can't execute this application. In terms of accessibility, this application is not designed with visual impairment in mind and will not work well with a screen reader. Additionally, it may be completely unsuitable for work with some languages; only English was used for the initial implementation. Many of those issues can be addressed with further development.

Legal Considerations

Copyrights, Patents and Permissions

This is an independent, standalone project. It consists exclusively of original work from the developers and libraries/assets with licenses that clearly allow the project to incorporate or link them. We checked every library used in the project for licenses, copyrights and other dependencies both manually and with a program like ScanCode—a command line tool that helps identify and locate licenses (NexB/scancode-toolkit, 2015/2020). If the licensing terms for a given asset or library were incompatible with this project, an alternative was used or created. Additionally, the development team checked assets against the United States Copyright Office Online System (at https://cocatalog.loc.gov). When naming the project, we ensured it had not been trademarked by consulting the United States Patent and Trademark Office Website.

Project License and Other Legal Concerns

The application for this project will come with its own license, likely Apache 2.0. It will notably include disclaimers of warranty and limitations of liability. Other than that, the license will permit free usage and modification by other parties. Despite having license protections, it is essential we work to ensure user's sensitive data is protected and its integrity is maintained.

Since this is a desktop-only application, data is never communicated to any server, so that is a security aspect this project will not need to address. Since this is a desktop program, the user's operating-system login should be sufficient to secure their data. As a result, no encryption or authorization is needed in the application.

To safeguard data integrity, we will use various levels of software testing to verify that software defects are not the source of mangled user data. Likewise, we will provide extensive application logging, validate application data and ensure user data is backed up mid-transaction, if possible (Managing data integrity, n.d.).

Timeline/Budget

Milestones & Timeline

Savvy's project milestones and corresponding timeline were met as defined in the capstone proposal. Their details are shown in the milestones and timeline sections.

Milestones

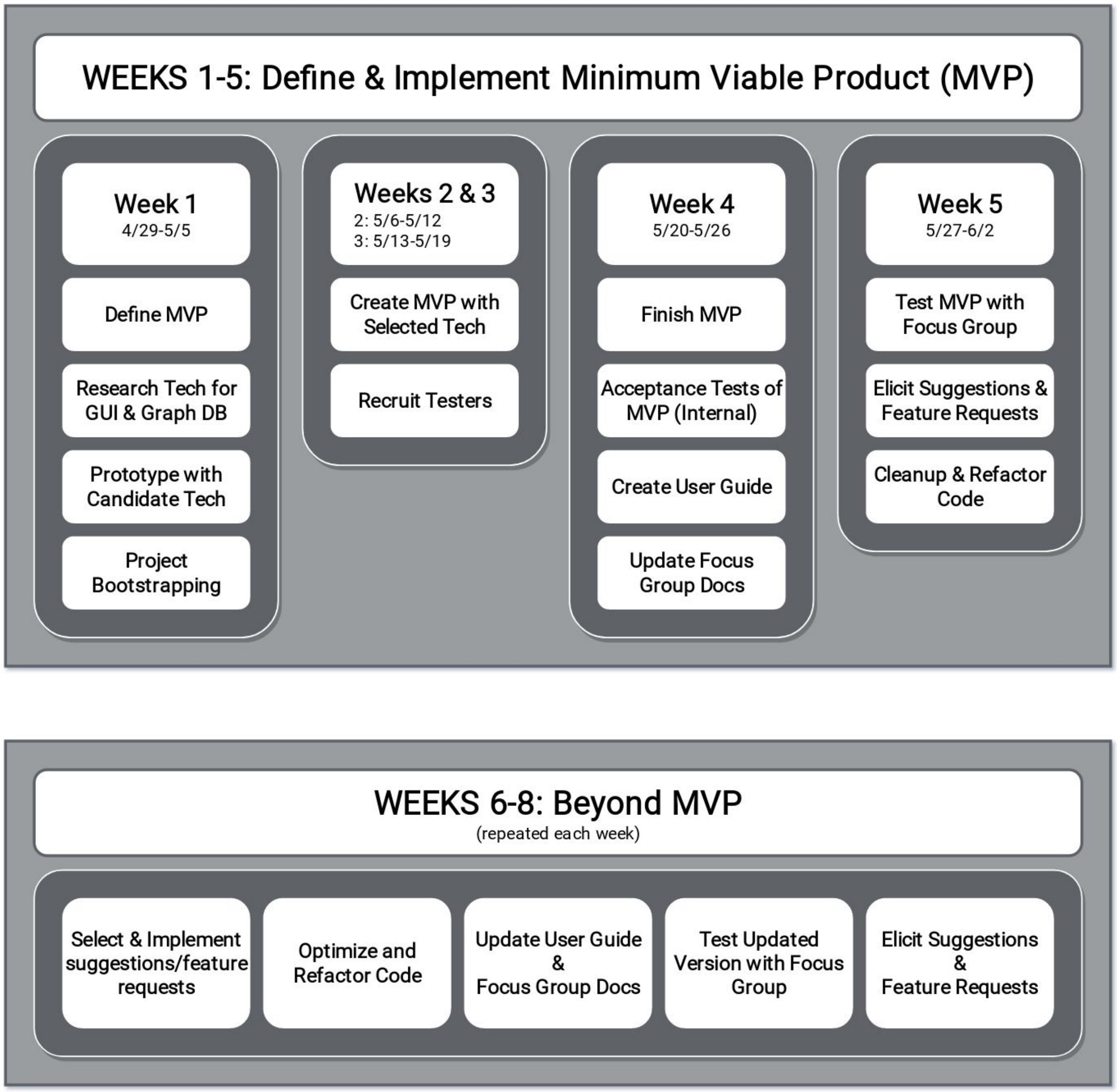
- Phase 1: Project Initiation (week 1)
    - Defined Minimum Viable Product (MVP).
    - Evaluated and selected technologies using prototypes.
    - Project Setup: Created Github repositories & Google Forms for Focus Group polling.
- Phase 2: Implement MVP (weeks 2-4)
    - Iterated on the required feature set to complete the minimum viable product with a focus on testing with good code coverage.
- Phase 3: Focus Group Testing (week 5)
    - Coordinated with Focus Group testers to validate if the MVP met their typical use cases.
- Phase 4: Iterative Improvement (weeks 6-7)

- Iterated on new feature sets informed by Focus Group testing and feedback.
  - Added two-term filtering.
  - Improved keyboard navigation.
  - Set the focused element when switching Tabs.

Timeline

Figure 4

Timeline for Weeks 1-8


WEEKS 1-5: Define & Implement Minimum Viable Product (MVP)
Week 1
4/29-5/5
Define MVP
Research Tech for GUI & Graph DB
Prototype with Candidate Tech
Project Bootstrapping
Weeks 2 & 3
2: 5/6-5/12
3: 5/13-5/19
Create MVP with Selected Tech
Recruit Testers
Week 4
5/20-5/26
Finish MVP
Acceptance Tests of MVP (Internal)
Create User Guide
Update Focus Group Docs
Week 5
5/27-6/2
Test MVP with Focus Group
Elicit Suggestions & Feature Requests
Cleanup & Refactor Code
WEEKS 6-8: Beyond MVP
(repeated each week)
Select & Implement suggestions/feature requests
Optimize and Refactor Code
Update User Guide & Focus Group Docs
Test Updated Version with Focus Group
Elicit Suggestions & Feature Requests

Budget/Resources Needed

Savvy had no budgetary demands. It was implemented using only open source libraries, which are available at no cost. The team already had the software development tools needed to complete the project, so none were purchased. Since the resulting application runs standalone on a desktop computer, we did not need to buy any domain names or web hosting services.

As this is a software-only solution, no hardware was procured to aid in its realization. Every step of the project development was completed with the materials and equipment already available to the team members. This includes: computers for development, writing/printing material and internet communication equipment. Likewise, software testers had the necessary hardware—a computer capable of executing the application.

Usability Testing/Evaluation

Savvy's usability was evaluated in two regards: that the application worked as designed and that its core use-case was met. Focus-group testers were tasked with evaluating the application in both aspects. First they assessed whether they could use the product with ease. Each worked with Savvy to enter, view, filter and review filtered results. They were also solicited about whether they prefer using Savvy over a prose-oriented application. Positive results in both represented a favorable evaluation. Savvy's overall testing schedule corresponded to milestone phases as outlined below.

Testing by Phase

- Phase 1: Project Initiation (week 1)
    - Not applicable.

- Phase 2: Implement MVP (weeks 2-4)
    - The graph database should be embedded within the application.
    - The GUI should be connected to the application core and database.
    - Users should be able to enter data.
    - Users should be able to search their stored data terms and have the expected items displayed in the GUI.
- Phase 3: Focus Group Testing (week 5)
    - Assess whether MVP meets Focus Group's typical use-cases.
        - Follow the Acceptance Testing Guide—in Appendix B.
    - Focus Group testers should be able to complete the tasks from Phase 2.
- Phase 4: Iterative Improvement (weeks 6-8)
    - Iterate on a new feature set informed by Focus Group testing.
    - The tests depend on selected features.

In Phase 2, Savvy's minimum viable product was being implemented feature by feature. At that time, the developers alone ensured the application operated as designed utilizing a combination of unit/integration testing and code coverage. Following an Agile development methodology, each developer made certain that their code was sufficiently tested before integration into the mainline codebase. Second and critically, they evaluated whether the application met its core use-case; being a powerful, yet simple to operate, replacement for prose-oriented information managers. This was proven by actual usage and assessment; done by the team continuously throughout the project and joined by the focus group after that Phase. The acceptance testing suite, included as Appendix B, was devised at the end of Phase 2. It covers Savvy's core use-cases and ancillary tasks necessary to satisfactorily complete them.

Prior to Phase 3, testers of various skill levels were recruited by the development team. Care was taken to ensure diversity of professions, interests, note taking styles and intended purposes for the application. To complete their part, testers only needed a computer running Windows, MacOS or Linux. For remote sessions, an internet connection was also required.

The focus group testing sessions were done one-on-one with a developer directing testers to complete tasks from the acceptance testing suite. During those sessions, the attending developer was deliberate in allowing testers to make mistakes while using the application to evaluate its usability and intuitiveness. The tasks performed by testers were based on the acceptance testing suite. Feedback from the focus group testing is included in Appendix C.

Testing in Phase 4 of the project was conducted in the same manner as Phase 2. It consisted of unit and integration testing, bolstered by manual testing of critical use cases. When new features were introduced, the acceptance testing suite was amended to match.

Final Implementation

Technology Selection & Architecture

The choice of technology was critically important for this project. Without an embeddable graph database and desktop GUI framework that could interoperate, an unacceptable amount of technical skill would be required to install and run the application. At that point the project could be considered a failure. Fortunately, suitable libraries were found to avoid that case. Savvy integrates three essential libraries to assist with its core functionality; JavaFX, Neo4j and greenrobot EventBus. As a desktop application, it leverages JavaFX and EventBus to handle its GUI and component intercommunication requirements. Neo4j, as an embedded graph database, provides the primary backend functionality.

Architecturally, this project has a GUI that employs the Model/View/Controller pattern and interacts with the core application layer through the event bus. The core layer handles application entities and solely interfaces with the graph database. The structure of the codebase is kept as flat as possible by arranging packages around the application's conceptual models.

That flattening is reinforced by leveraging the EventBus library to pass signals and data amongst components. Event Bus is particularly useful in that regard because it allows loose coupling of components using a publisher/subscriber pattern (“EventBus,” n.d.). Consequently, whenever a new core feature is implemented, it can merely be set to broadcast data in an event instead of wiring directly to potential listeners. Likewise new GUI components can leverage existing core resources without altering those components.

Conceptual Models

Savvy's conceptual models correspond with its architecture. It has a Core Model that represents its core application components and functionality. This frees the core application from being tightly bound to the chosen user interface framework and opens up the possibility of porting Savvy to other platforms; like Android. Savvy also has a User Interface Model that represents the graphical components that users interact with. Finally, the events model provides a standard way for each model's components to mutually communicate. Each model and its members are described in Table 2.

Table 2

Conceptual Models in Savvy

Core Model

Members	Description
DB	Interfaces with the embedded Neo4j instance.
Entities	Interfaces with the group of all entities.
Entity	Represents a particular entity.
Relationships	Interfaces with the group of all relationships.
Relationship	Represents a particular relationship.
Facts	Interfaces with all facts.
Fact	Represents a particular fact. A fact is composed of two Entity instances and one Relationship.

User Interface Model

Members	Description
App	Represents the application itself. Holds the main, start functions & application-wide settings.
Common	Contains static utility functions for UI elements.
Entities List	Represents the Entities List in the UI.
Relationships List	Represents the Relationships List in the UI.
Fact Create	Represents the Fact Create element in the UI.
Facts List	Represents the Facts List in the UI.
Menu Bar	Represents the Application Menu in the UI.

Events Model

Form	Description	Example
DoXY	Instructs the target listener (X) to perform a given action (Y).	DoFactCreate

XYed	Instructs listeners of a target (X) that a given action (Y) has been performed.	FactCreated

Technical Implementation

Application Core

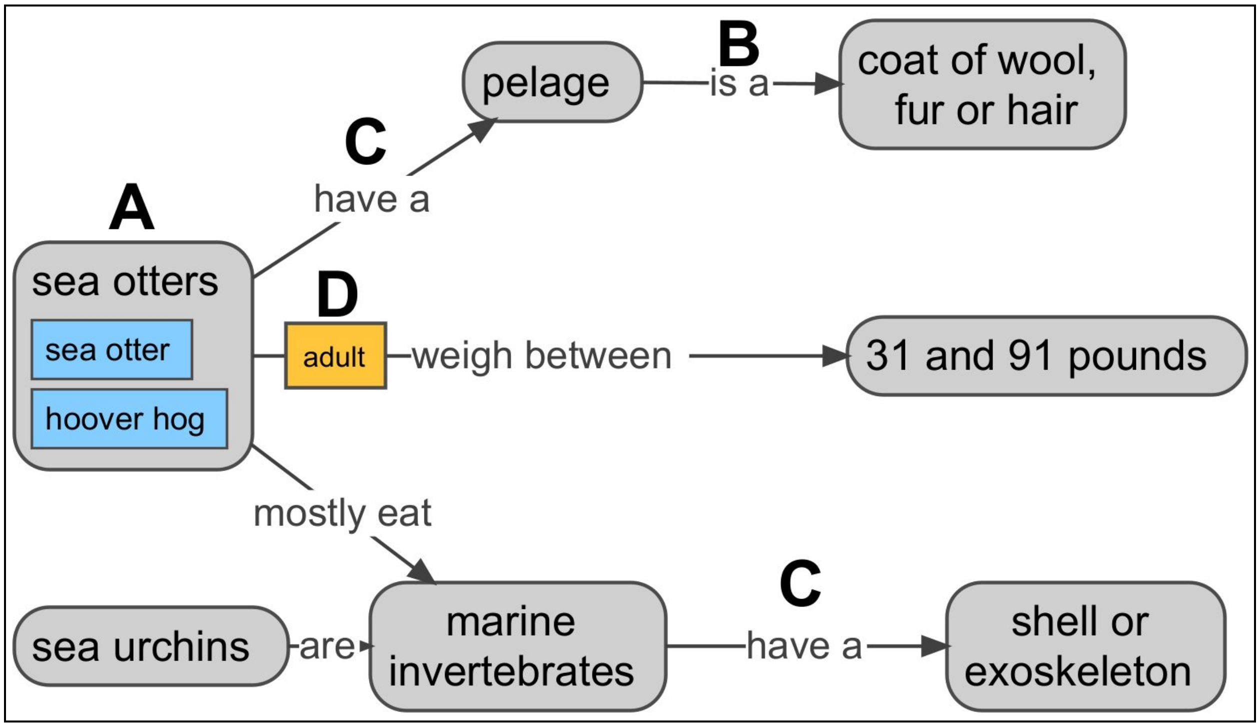
The core of Savvy is entirely custom code. It is responsible for setting up the embedded Neo4j database and interfacing with it. To support those interactions, it contains a generalized data access object (DAO) and a serializer/deserializer. The DAO provides a single point of contact with the database, which helps prevent errors and allows for centralized optimization of data access requests. The serializer/deserializer is required to convert application specific data structures into a format that can be stored on the database. In this case core model members are converted into a byte array.

Each of the other Core Model members represent concepts relevant to the application's business logic. Broadly, for each Fact, Entity and Relationship there is a corresponding model member that represents its grouping. Likewise the grouping based members each respond to corresponding events from both core and UI components. For example, the DoFactCreate event is broadcast on the event bus by a component whenever it wants a new Fact created. That broadcast is received by the Facts member who will create the new Fact and respond with a broadcast of FactCreated. All components that are listening for the FactCreated event will receive a copy of the newly created Fact to work with internally.

Mapping Core Model to Neo4j

Figure 5

Core Model Represented in Neo4j



One important aspect of this project is how application data structures are mapped to the underlying database. The Neo4j database stores information in the form of a graph. In order to maintain the fidelity of Facts, Entities and Relationships from the application, the placement of that data had to be carefully selected.

The arrangement of Core Model members is illustrated in Figure 5. Note how each Entity is represented as a single node in the graph at marker A. An Entity also has a single dedicated list of aliases. Between nodes in the graph, there can be relationships—shown with marker B. Graph relationships correspond directly to Relationships in the Core Model with a couple of exceptions. First, a Relationship's Correlates are duplicated among all relationships that have the same identity. In the graph database every relationship is unique, but in the Core Model the same conceptual Relationship, for example: *have a* in Xs *have a* Y can be shared among countless Entities. To account for that identity property in the database, we duplicate the shared data among all relevant relationships in the database. An example of that case is marked with a pair of Cs in the figure. In the Core Model there are some Fact-specific properties that are saved in the database relationship; namely Entity modifiers. For example if you add a quantifier to an entity like adult sea otters instead of plainly stating sea otters. As shown at marker D, we affix that information to the graph relationship because it is particular to the fact and not to the generalized Entity. Also note: modifiers are on that particular relationship in the graph, not affixed to the Core Model's Relationship.

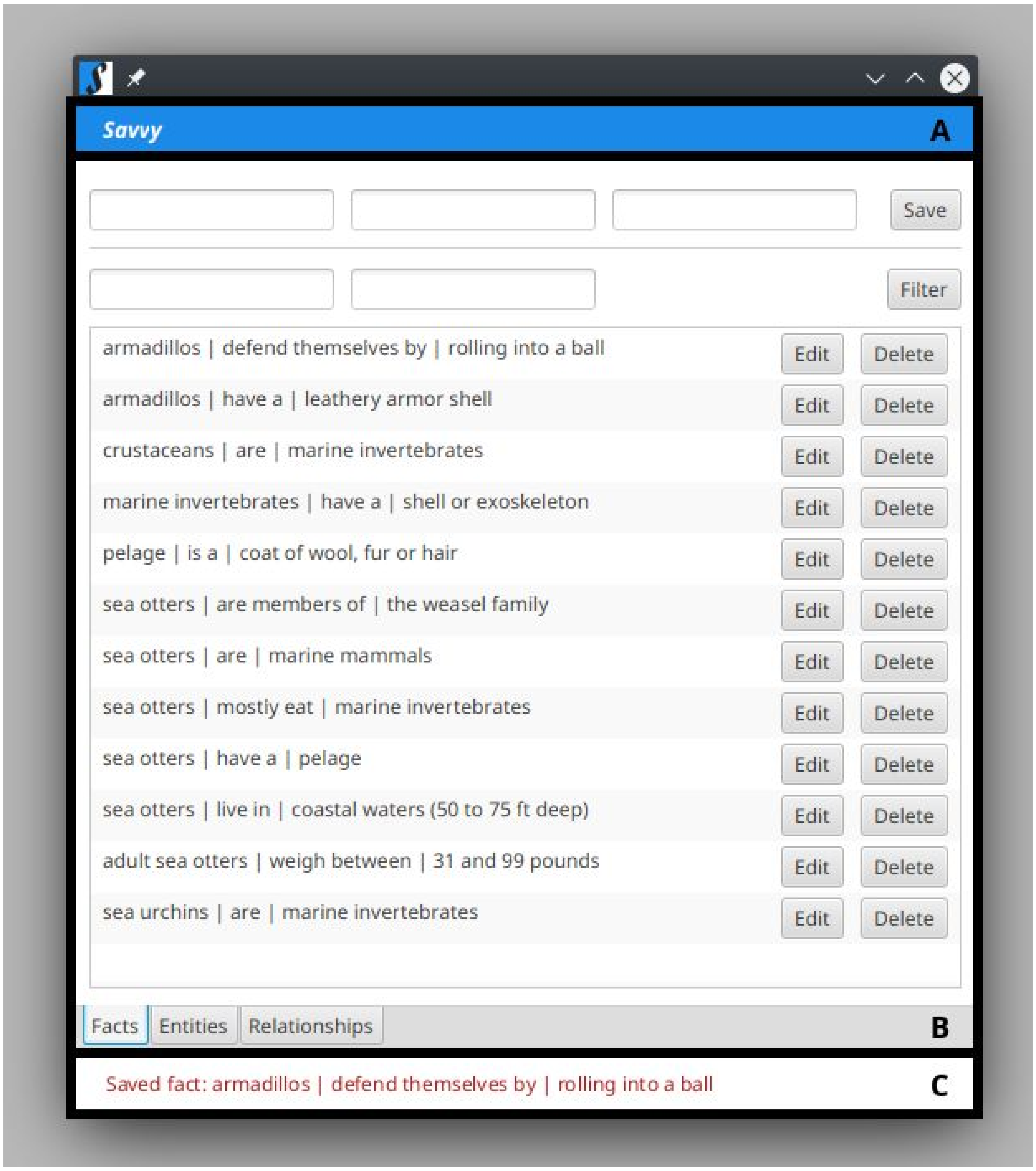
User Interface

Savvy's graphical user interface was written using JavaFX with a mixture of FXML and procedural Java code for dynamic elements. FXML is an XML-based language that defines the composition and layout of the application. The main interface is defined with a controller called AppController and a layout file called app.fxml. That FXML specifies properties for a GridPane containing the application's main interface components; labelled in Figure 6.

Those components are: the Menu Bar (A), a Tab Pane (B) and the App Messages line (C). Most user interactions happen in the Tab Pane which has three tabs, one for Facts, Entities and Relationships. In addition to the layout files, a stylesheet is referenced in the FXML to modify the colors, layout and fonts.

Figure 6

UI Regions in Savvy



armadillos | defend themselves by | rolling into a ball
armadillos | have a | leathery armor shell
crustaceans | are | marine invertebrates
marine invertebrates | have a | shell or exoskeleton
pelage | is a | coat of wool, fur or hair
sea otters | are members of | the weasel family
sea otters | are | marine mammals
sea otters | mostly eat | marine invertebrates
sea otters | have a | pelage
sea otters | live in | coastal waters (50 to 75 ft deep)
adult sea otters | weigh between | 31 and 99 pounds
sea urchins | are | marine invertebrates

Saved fact: armadillos | defend themselves by | rolling into a ball

Behind the scenes, the AppController handles binding the FXML elements with lower level controllers and the application core. Each of the lower level controllers has its own FXML file to define their general layout. For example, the FactCreateController has its layout defined in fact_create.fxml.

Each of the main GUI views have their own tab in the Tab Pane. These views are Facts, Entities and Relationships. The Facts view is composed of a pair of layouts and controllers, one for Fact creation and another for viewing, editing and filtering the list of previously created Facts. When a Subject/Object is modified from the Facts view, it is treated as part of the Fact. That means if the new Subject/Object does not match any existing Entity, a new one will be created. Likewise, if it is changed and matches an existing Entity, the relationship will be deleted and recreated linking the current participating Entities. This has a key distinction compared to how edits are handled from the Entities and Relationships views. Whereas when the Entity is changed in the Entities view, that change is applied to every Fact that Entity participates in. The Relationships view behaves identically to the Facts view except instead of adding Aliases to it Correlates are used.

Discussion

Design Choices

Certain choices about the architecture of the application code and the selection of technologies were made at the onset of the project. The application required both a GUI and a backing Graph Database to be viable. If an adequate framework for either was never found, this project would have failed. Further, if either was difficult to understand or work with, then the project will be delayed or impossible to complete. One goal of this project was to provide a solution that works on common desktop operating systems. If no suitable GUI framework was available, support for some OSes would have been jeopardized.

The graph database component was crucial for the success of the project. Since the application was intended to be standalone, it would ideally include an embedded database; sparing users from installing and configuring that database component. Such embedding might have been impossible. In that case, we would have needed to produce an automated installation of the graph database; potentially impacting the timeline.

For the graphical user interface, we preferred JavaFX over Swing for its modern looking interface and usability. Other factors we considered were that JavaFX can be styled with CSS and supports Windows, Linux and Mac OS (*1 JavaFX Overview (Release 8)*, n.d.). Several other frameworks were assessed as well, but they either looked out of date or were too difficult to develop with given a short time frame.

The selection and ordering of features to implement were first decided by evaluating the dependent features for completing the minimum viable product. For example, we needed a basic GUI before we could wire the model to its controls. Later we needed to be able to create Facts with Strings before we could create them with full fledged Entities and Relationships. Likewise, the addition of the greenrobot EventBus came out of the necessity to pass data between components without tightly binding them together.

Later, features from past iterations were included to increase user satisfaction with the product based on the focus group testing. Other features were also added when selected by the team in order of importance to the core functionality or usability of the application.

Challenges During Design and Implementation

Some of the major challenges in this project included: embedding the Neo4j database and working with the JavaFx framework, which was novel to our team. During the prototyping phase, creating a viable Embedding Neo4j with the JavaFX framework was *the* critical challenge. Several days of that phase were spent trying to get the libraries to interoperate and produce a viable cross platform prototype. At one point, prototypes were made using everything from Neo4j with Vaadin and Electron to configurations with various other graph databases and front end frameworks. Ultimately, we succeeded with a prototype build based solely on JavaFX and Neo4j.

Another difficult design challenge presented itself when we began implementing Aliases and Correlates. Previously, we were using simple Strings to represent the values for a Fact's Entities and Relationships. When we migrated to full fledged Entity and Relationship types, a cascade of associated changes was also required. We needed to create types for each of the Entity and Relationship concepts and refer to those in the already existing Fact type. We also needed to store those types in the database, but that change demanded we convert their data into something that the database understands; so we had to implement a serializer/deserializer. To round out the changes, every reference to the String version of Entities and Relationships in the GUI had to be updated to its corresponding type.

Figuring out the design for mapping Core Model members to the Neo4j Database also presented a challenge. Conceptually, the parts of the Core Model and the graph database are similar. However, differences in what a Fact represents and what a sequence of Node, Relationship, Node represent presented issues about where particular Core Model information should be stored within the database while preserving the fidelity of the information.

In terms of administering the project and its source code there were some challenges with getting everyone to use the source control and project management tools effectively. Initially, we were not able to set up the continuous integration functionality in Github due to minor errors. This was rectified with a bit of research and editing. Also team members needed a refresher on how to pull from the master branch and submit smaller commits as Pull Requests to prevent having to perform large merges between repository branches.

Project Rollout

We built the project using gradle. The assemble task packaged up the Savvy project as a jar and created a zip file containing that jar and all of its dependencies. For focus group testing we either gave the zip file to the testers in person using a flash storage device or pointed them to an online directory to download themselves. Since there are no particular clients for this project, we did not need to deliver anything. However, the project code is open source so anyone can visit the repository and build the project in the same manner as the developers. Instructions on how to do that are posted in the project's read me file.

Conclusion

The Savvy Project

As presented in the opening sections of this report, approaches to collecting and managing information for individuals have hardly improved in the past decade. At the same time both graph database technology and cross-platform GUI frameworks have become robust and stable. The time came to create an application that frees users from the problems related to common prose-oriented approaches and lets them leverage the advantages of graph databases—in the same way that professional information management software provides, but without their difficulties. Savvy improves note taking and information management user experience compared to prose-oriented approaches by leveraging pre-existing notes to give users a comprehensive view of their information. Since the graph database links the referenced entities it makes filtering and management simple. At the same time Savvy is easy to understand and user friendly.

In order to implement the project, an agile software development life cycle was chosen. We began with an investigation and prototyping phase for the selection of technologies and defined the feature set for the minimum viable product. From there we implemented MVP and moved to focus group testing before continuing with an iterative improvement cycle.

Savvy's software design is rooted in fact that it is intended for personal use with a desktop computer. We expected that the application should be simple to install and standalone. As a result, we selected an embeddable graph database and a cross-platform GUI framework. We used conventional patterns for arranging the code and the interfaces between application components. The major parts of the application included the Core, GUI and Event models. For the core we separated the domain model from the database by using a data access object. In the GUI we used both FXML layout files and custom code all managed using a Model/View/Controller approach. The components of the application communicated according to a standardized Event model over an Event Bus.

Learnings from the Project

Through this project, we learned how to manage time effectively while working to meet deadlines on a multiple-week project. We learned new technologies in JavaFX with FXML, Neo4j with its Cypher query language and how to effectively use Event Buses to pass messages between application components. We also gained experience with project management using Github Issues and Projects to create tasks, prioritize, assign and track them. We learned how to add continuous integration automation to standardize the code formatting, run tests and report on the status of the build.

In terms of product design and implementation, we learned how to evaluate and prototype with various technologies prior to implementation in order to assess whether they will be effective and easy to work with. We also experienced times where a feature in progress presented challenges as to how it should be implemented with the existing code base and selected technologies. Additionally, we discovered ways to improve the organization of our code and documentation.

The Future of Savvy

In the future, we hope to implement several features to improve both the core functionality of Savvy as well as improve its user friendliness. A comprehensive list of improvements proposed to Savvy can be found on its GitHub Issues page at: https://github.com/csumb-serious-business/savvy/issues. Some important issues listed there are allowing import/export of knowledge bases, switching topics so that users don't have to mix their Facts in a single large knowledge base and preloading common relationships to make fact creation more convenient. Since this approach to note taking is new, a number of different features can be added. As with any software project, the possibilities for improvement are endless.

Appendix A

Team Member Roles and Responsibilities

The team members for this project are Julie Asato, Matthew Robertson and Raymond Talmage. Tasks for this project varied along with the timeline and were divided among team members according to the following outline. During the research and prototyping phase in the first week, all team members defined scope for the project's Minimal Viable Product (MVP). Also, Julie researched and prototyped with the candidate GUI frameworks; Matthew did the same for the Graph Databases. Raymond bootstrapped the project, including set up for the repositories, agile development tools and focus group surveys.

In the MVP implementation phase—weeks 2-4—tasks were divvied among team members using an agile methodology. Additionally each team member was tasked with recruiting one or two testers for the focus group. In the fourth week, preparation work was completed for focus group testing. During that week Raymond created the User Guide and updated the focus group documents. Julie and Matthew worked on acceptance testing. The fifth week asserted the Focus Group Testing and Refactoring Phase. All team members supported the testing and feedback efforts and took on tasks for improving the code quality and efficiency. From the sixth week onward, team members continued to iterate on the application making selected improvements guided by focus group assays and following an agile approach. In each iteration they optimized and cleaned up the codebase, updated the various documentation and gathered recommendations.

Appendix B

Acceptance Testing Suite

- Facts
  - When I begin to enter a Subject/Object the autocomplete box shows Entity identifiers.
  - When I begin to enter a Relationship the autocomplete box shows Relationship Correlates.
  - When I create a Fact, it is shown in the Facts List.
  - When I successfully edit a Fact, changing the Subject or Object to a new Entity, that new Entity should be created.
  - When I successfully edit a Fact, changing the Subject or Object to an existing Entity, that new Entity should be referenced.
  - When I delete a fact, it should be removed from the Facts List.
- Entities
  - I should not be able to delete an Entity.
  - When an Entity no longer has any Facts that refer to it, it should be deleted.
  - I should be able to add/remove Aliases on Entities.
- Relationships
  - I should not be able to delete a Relationship.
  - When a Relationship no longer has any Facts that refer to it, it should be deleted.
  - I should be able to add Correlates to a Relationship.
- Facts/Entities/Relationships
  - When I successfully edit a Fact/Entity/Relationship, it shows a message about the update.

  - When I edit a Fact/Entity/Relationship, it should give an option to cancel the changes.
- Filter
  - When I begin to enter an Entity name, the autocomplete box shows Entity identifiers.
  - When I filter with 1 term, I see only facts related to that term.
  - When I filter with 2 terms, I see only facts between those terms.
  - When I click filter, it will only show relevant facts to the keyword
    - There are no irrelevant facts included.
- Other
  - When I am stuck, I can view and use the guide to help clarify my problem.
  - It should be easy to install the program.
  - I should be able to run the program on Windows, Linux and MacOS.

Appendix C

Evaluation, Testing, Review Results from Anonymized Users

- Add a confirmation box when deleting things since they vanish with one click.
- I closed and reopened the application and the test data was still there.
- Default aspect ratio makes entering long objects cause important UI elements obscured by frame. This is especially troublesome when editing. (Save button obscured)
- When using the filter function, after hitting the enter key the search term is removed from the text field. It would be better for the term to remain (Much like in Google Search)
- The unrecognized characters on Windows 10 need to be fixed.
- The keyboard navigation needs improvement.
- The help menu should include more information.
- Add placeholder text in the fields.
- Add the ability to have facts sorted by topics in order to keep relevant information together.
- The user interface is simple to understand, but could be improved to be less rigid and more visually appealing.
- The process to install the program could be modified to be more accessible to users with less experience with computers.

References

1 JavaFX Overview (Release 8). (n.d.). Retrieved June 19, 2020, from https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784

cleverbridge, Inc. (2020). ATLAS.ti Commercial Licenses. https://atlasti.cleverbridge.com/74/catalog/category.6302/language.en/currency.USD

EventBus: Events for Android. (n.d.). Open Source by Greenrobot. Retrieved June 16, 2020, from https://greenrobot.org/eventbus/

Evernote Corporation. (2018). How to use Evernote's advanced search syntax. Evernote Help & Learning. http://help.evernote.com/hc/en-us/articles/208313828

Friese, S. (2019). ATLAS.ti 8 Windows – Quick Tour. http://downloads.atlasti.com/docs/quicktour/QuickTour_a8_win_en.pdf

Lexico.com. (n.d.). Savvy. In lexico.com dictionary. Retrieved May 19, 2020, from https://www.lexico.com/en/definition/savvy

Musen, M. A. (2015). The Protégé Project: A Look Back and a Look Forward. AI Matters, 1(4), 4–12. https://doi.org/10.1145/2757001.2757003

Rubin, D. L., Noy, N. F., & Musen, M. A. (2007). Protégé: A tool for managing and using terminology in radiology applications. Journal of Digital Imaging, 20 Suppl 1, 34–46. https://doi.org/10.1007/s10278-007-9065-0

Stanford University. (2016). Protégé Short Courses. https://protege.stanford.edu/short-courses.php