The following is my entire planned list of enhancements for this artifact, as previously included in my updated CS499 Final project plan (Module 1 assignment).

- **Enhancement 1:** Implement Rust's error handling using Result<data T, Err> types. Use the the ? operator to manage propagation.
- **Enhancement 2:** Move away from using unwrap() statements to avoid panic crashes.
- **Enhancement 3:** Implement standard library traits wherever possible to reduce code rewriting, such as with trait Default.
- **Enhancement 4:** Implement the Option type for any value that could return None.
- **Enhancement 5:** Restructure the code base to utilize composition based design over inheritance.
- **Enhancement 6:** Implement proper delegation standards for function calls.
- **Enhancement 7:** Use enums with match statements for reusable type definitions (in place of constants).
- **Enhancement 8:** Organize the code into distinct modules for better encapsulation.
- **Enhancement 9:** Implement structs to define custom data types (instead of classes, etc).
- **Enhancement 10:** Utilize traits to provide polymorphism within the application.
- **Enhancement 11:** Include relevant annotations, macros, attributes where needed, such as with cloning.
- **Enhancement 12:** Implement proper memory management techniques for manually defined lifetime definitions.

1. Briefly describe the artifact. What is it? When was it created?

The first artifact started out as my final project for CS410, Reverse engineering. It is a client management system for "SNHU Financial Investment Firm". Our job in creating this project was to take the original code which had no form of exception handling or validation of any type. We were then told to implement security checks into the application that would fix all of the vulnerabilities that were in the original code. It started out as a binary executable which I disassembled into assembly instructions. I then decompiled / reassembled the instructions into C++, and implemented various design overhauls to remove the prior vulnerable implementations. I finished this project in June of this year, 2024. Since that time, I have rewritten the application entirely in rust. In doing so, I have also

significantly improved on the security of the applications, all while demonstrating that I have grown and understand core software engineering principles.

2. Justify the inclusion of the artifact in your ePortfolio. Why did you select this item?

I selected this project for my first artifact for several reasons. The first of these reasons is that it is the culmination of my software engineering education. It was my second to last semester of courses, and simultaneously was the point where a number of the concepts and data structures that I have been learning about during my program had been given enough time to sink in. Additionally, I had read and written about how memory works, and how computers perform operations. But I hadn't yet seen the low level instructions that they use to perform their work. I have always been fascinated with taking things apart, and figuring out how they work. This project both demonstrated my understanding of low level instructions, as well as key software engineering principles such as design patterns and data structures. And as such, I am now more capable than ever of conceptualizing and generalizing on that knowledge, and I will continue using it to become a better software engineer.

What specific components of the artifact showcase your skills and abilities in software development? How was the artifact improved?

The starting point of this application initially fit into a single small main function with two loops. For CS410, I built it out into a relatively complete application in C++. It was functionally complete, with several different classes to handle the various types of operations for my CS410 project. I have since fully rewritten the application into rust, which is a language that provides a much higher degree of memory safety. The process of rewriting this application in rust has been tedious, but I am very happy with how the application has turned out so far. The flow of information within the application is through well structured, abstracted interfaces that allow for modular implementations. And due to this modularity, future changes to the application will be much easier to write. Not only have I completely rewritten and restructured the application, but I also have provided complete documentation for each of the various code units, commented using best practices for rust, as depicted in the rust foundation's documentation. The documentation that I have, as well as what I will be providing for the complete application this semester, will be capable of generating a complete documentation HTML site for my rust crate for SNHU Investment Firm. Upon inclusion of all segments of the application, this

documentation site can be built by executing the command 'cargo doc', while cd'd into the rust project directory.

I believe that my implementations for this artifact show that I have a strong understanding of software engineering design. The way in which I was able to showcase these skills primarily have to do with how I chose to structure and organize the code for my application rewrite. The planned enhancements were a way in which I sought to encapsulate some of the best practices, and established standards for the rust language into a meaningful list of implementation details that I could use as a checklist for my work. To more easily explain how the artifact was improved, and how the different components showcase my knowledge / abilities in software engineering, I will briefly go over each of my planned enhancements, and how I implemented them.

The first of these enhancements I believe shows that I have a strong understanding of the design and planning stage. I have demonstrated that I am capable of taking larger, complex problems, and decomposing them to a point where they can then become a meaningful segment of code. I broke down and isolated the various desired behaviors of the larger application into manageable sections, that could later easily be reused or refactored if the implementation needs to change. The structure of this application is broken up into a series of modules, each responsible for a category of behaviors in the application. These modules provide an easy method of tracking my planned enhancements for this artifact, and also contributes towards **enhancement 8** directly, "Enhancement 8: Organize the code into distinct modules for better encapsulation."

Enhancement 9: firm_models.rs

For **enhancement 9** we have, "Implement structs to define custom data types (instead of classes, etc)." I implemented structs to encapsulate the core custom data types, Client, and Employee within the firm_models.rs module. I then was able to properly implement these struct types and define the functions of both within their implementations in the same file. For enhancement 10, utilizing the trait interface, the only place where I needed to implement a trait was in my database.rs file. That is a portion of my third artifact deliverable, and while not entirely finished, is defined and will soon be filled out. It both meets and displays that this enhancement has been met. As traits are essentially

interfaces in rust, the database management interface is an abstraction that allows for the application's dependency to be passed in / injected when needed at a specific location.

Enhancement 7: menu.rs, errors.rs

To meet **enhancement 7**, I used modular implementations of enums in two modules, the menu.rs module, and the errors.rs module. In the menu module, we have two enums that provide the constant values that are used to direct program flow through the menu interface system. In the errors.rs file, there is likewise two enums. The first is used to provide constant reference to the various custom ApplicationError types that are defined for the application, the second enum provides the error types for the database specific error sub-types.

The utility module provides modular input functions that provide sanitized, validated input to any sections of the application that need user input. The errors module provides custom error definitions to all of the other modules in the application, and with general ApplicationErrors, and a subcategory of Database specific errors that provide more detail into the various errors that can occur within the database operations. The firm_models module provides Employee structure, and the Client structure, two data models that are central to all of the operations within the application. For the last module / component that I built out for this artifact, we have the menu module. The menu module is where the primary control flow of the application takes place. Using a method referred to as constructor injection, A boxed dependency reference to an instance of the database manager implementation is passed in with the function call, and the application is provided with any of the dependencies that it requires to perform its operations.

Enhancement 5:

For **enhancement 5**, the way in which I designed my application incorporates the modified approach to Object Oriented Programming that rust uses. This approach can be summarized in the phrase, "composition over inheritance". Where traditional OOP languages such as C would use an inheritance based approach to implement polymorphism, rust uses composition to create more complex data types, and traits to define shared / reusable behaviors instead. Rust does not feature an ability to perform inheritance at all in fact, but this is not a limitation, and it aligns / works very well with rust's focus on providing explicit control over the behaviors and data within your code. We perform

compositions by first defining smaller, individualized structs, and then use them to compose larger, more complex structs. This is a foreign concept to most, especially at first, but I have been finding that it does seem to offer a higher level of modularity to implementations than traditional class based inheritance.

This next component will cover both **enhancement 3,** and **enhancement 11**. For **enhancement 3**, I implemented the standard library traits in a few different forms throughout my code base. The first of these manners is through the automatic implementation system that rust offers for my custom types, using the `[#Derive()]` macro. Derive macros are a way for us to automatically implement traits for our types, and the most common implementations that I used in this artifact were the Clone, Debug, and PartialEq traits provided in the standard library. The second way that I implemented standard library traits was with a method that I previously had been unfamiliar using, manually implemented / defined behaviors. There were three examples in my errors.rs module where the compiler told me that I needed to specifically implement a certain trait for my custom error types. This included implementing the fmt::Display trait for my DatabaseError type, the std::error for my DatabaseError type, the configuration / config::ConfigError for my more general ApplicationError type, and the mysql::Error for my ApplicationError type.

Enhancement 11: errors.rs

To meet **enhancement 11**, in my ApplicationError enum definition I provided the from attribute which allows me to easily convert std::io::errors to my custom ApplicationError::IoError type. I also used the `#[error()]` attribute that is provided by the thisError crate, and recommended by many as best practice for automatic implementation of the Display trait for each of my different error enum variants. It allows me to easily define a custom error message to be displayed when triggered. The use of relevant # define macros on relevant definitions also meets planned enhancement 11.

Enhancement 12: operation_handlers.rs(transaction) – data-structs.rs (AVL tree find function)

In relation to **enhancement 12**, for artifact 1, none of the greater sections of my code base needed to have manually defined lifetimes. There are two places that I need to manually define lifetimes in this application, and both will be included with my Artifact 2 and 3 code sections. Rust's borrow checker automatically remove data once it is no longer in scope. To be able to move data around within the data

structure I will be using for local operations, I will need to define the lifetime of that data, essentially just letting the compiler know how long I need access to it. I similar case will be used in the creation of my transaction system. I will need to define the lifetime of the data within a transaction, which will require the data to exist for as long as the transaction is active.

Enhancement 1,2, 4:

Enhancement 1, 2, and 4 are all very closely interlinked, so I will mention them and how they showcase my abilities together. In rust, we do not deal with errors (or exceptions) in the same way as other languages. In rust, generally when a function is called, it produces a Result. This result can be a simple Ok(), an Ok() accompanied by a value/data, or it can return an error. Rust provides you with the ability to be incredibly specific as to what you expect and allow as a result from an operation, and this is part of why it is considered to be as safe as it is. Meeting the standards for enhancement 1, required the need to manually define the expected Result(s) of every function call within my application, at least the ones that are capable of returning a result. This can be seen throughout the application with my inclusion of `Result<(), ApplicationError>` where the Result() would be any non error result, and the Error position can be filled with a default error, or in my case, with my own error definition. The details of this error definition are included in the errors.rs project file, and provide specifics for the various error types that can occur within the application.

There are two other returns I utilize within the application to facilitate management of errors. The first of these is used for the implementation of enums, and features an Option() or optional return. This Option() is an optional return of something (Some) which is a match to one of the constant values, or nothing (None). The implementation of this return type could look like a simple match statement, with conditions for the desired return values. To define the condition that would trigger a return of nothing, None, we could use a single underscore `_`, this is known as a wild card. The wildcard can be used in a match statement like this to accept any values that are not explicitly defined with the other match conditions.

The other type of returns used for my functions is used in the accessor functions of the Employee and Client struct implementations. These accessor methods don't define error handling explicitly, only because it is generally best practice to not implement error handling in a simple data object. The data object defines the ability to retrieve / access the data stored within. For example, the only value that the data object 'Client' can return for a 'client_id' value is a 32-bit integer, i32 value. It is then the

responsibility of the method that is calling the accessor of a specific instance of the data object to say I am expecting the result to be "Ok(i32)", if the result is not Ok()/successful, and doesn't return a 32-bit integer, then the result is an error of type ApplicationError. We explicitly state and define what values can be in a result, and writing our application this way is best practice. It prevents the application from throwing a panic, and lets us gracefully pass the result of a function call back up the call stack by appending the ? operator to the end of a function call. The other way that I made sure that I handled errors within my application, was not by using the .unwrap() method to retrieve the contents of function calls. If you call .unwrap() on a boxed function result, and the result contains an error, then it throws a panic, which is not a graceful way to handle errors in rust. Instead, I have placed a combination of optional return values, explicitly defined results paired with ? usage for error propagation, and the avoidance of using unwrap to handle values that can fail an operation.

Enhancement 6:

Enhancement 6 is mentioned last, because it most broadly covers the enhancement specifications. To meet this enhancement, I believe it requires demonstrating that I understand the importance and nuance of software engineering principles. The more common of these principles are those such as not repeating code unnecessarily (DRY), as well as S.O.L.I.D. design principles, and an understanding of common design patterns. I sought to meet this enhancement, and to implement proper delegation standards in rust through the practices that I used to structure and build out my application. This includes a number of different features, but I believe some of the most important ones in this context are as follows:

- Defining clear boundaries for the responsibilities of each function. Each function is clearly defined, and is responsible for it's own behaviors in isolation. This leads into the next aspect which I believe is an extension of this.
- Modularity of design. This application is designed with modularity in mind, and the various sections of code are cleanly divided/encapsulated into their own modules. Modules are isolated and only have access to the specific functions or objects within other modules that they need to perform their own operations.
- Following modularity, we have reduction of overall program complexity by decomposition of functions into relatively simplistic units based on actions / behaviors.

- For error handling, each function properly utilizes the ? operator to back propagate errors, and handle them gracefully within the application. This provides a central control flow for the different error variants that can occur within the application.
- Dependencies are injected / passed in via a flexible interface implementation, this provides a significant improvement to readability, while reducing the overall complexity of the application. This would be tied back into the trait implementation for the databaseManager. I designed the application to have reliance on constructor injection / implementations of the DatabaseManager trait. This trait is used to define the core, low level data operations within the application. The data operations are then accessed using either the handler for Client data structs, or Employee data structs. The handler implementations are what will generate the transactions that allow data to remain consistent between remote and local storage. They also will broadly be responsible for accessing any needed values.

I believe that together, these points all tie into the greater picture, and show that I have carefully, thoughtfully considered how I am defining and implementing code within my application. These factors all contribute towards the greater goal of meaningfully improving the artifact.

3. Did you meet the course objectives you planned to meet with this enhancement in Module One? Do you have any updates to your outcome-coverage plans?

There were two course objectives that I set out to meet with the completion of the first artifact and these enhancements. The first was to “display a security mindset that anticipates adversarial exploits in software architecture and designs to expose potential vulnerabilities, mitigate design flaws, and ensure privacy and enhanced security of data and resources.” The second was an "ability to use well-founded and innovative techniques, skills, and tools in computing practices for the purpose of implementing computer solutions that deliver value and accomplish industry-specific goals.” I do feel that I have been able to meet both of these outcomes with my design patterns and implementations for the first artifact. I have thoroughly documented the code modules that I wrote for this artifact, along with including a detailed error handling system that can very easily be modified in the future to handle additional error types. I have spent a large amount of time working on this project, but I am very aware of the fact that there is a tremendous amount that I can still learn. My implementations are not perfect, and while I

have made my best attempts to follow the documentation, and use best practices, I am sure that a well seasoned software engineer could very quickly locate areas for improvement. That being said, I do believe that I was able to deliver on the outcomes that I set out to reach. Additionally, I do believe that I am on track to fulfill the outcomes that I previously mentioned coverage for.

4. Reflect on the process of enhancing and modifying the artifact. What did you learn as you were creating it and improving it? What challenges did you face?

The process of working on and enhancing this artifact has been the most challenging project that I have yet undertaken during my career in programming. I thought that I was fairly comfortable with rust concepts prior to this project, but working with the compiler for this project has taught me a tremendous amount about how the borrow checker handles the lifetimes of data, and just how specific it is when it comes to type definitions. The borrow checker is not something that I am yet overly familiar with, so there have been a number of times where I found myself needing to rethink details about my implementations. The majority of the challenges that I have faced in working with the compiler have been involving my definitions of error types. Specifically, when I started out writing this application, I was intending to mostly use generic error types provided by the standard library. As I began adding additional modules to the project, I quickly realized that solely relying on the standard library implementation would not best serve my uses. So I spent two days reading documentation, code examples, and watching videos, and during that time I wrote, and then again rewrote my own custom error handling for the application. For each instance in the application where I had non-optional results being produced, I needed to add both the appropriate error type, and handling for it. For some of these results, I was able to rely on my more general ApplicationError, but for others with specific causes, I used a more descriptive custom error type. I used best practices of error propagation via inclusion of the question mark '?' operator. Use of this operator in a function allows us to say to the compiler "if the result from this operation produces an error, immediately return this error type back up the call chain for handling". This provides a very convenient error handling process, as well as a level of detail and control over function flow that I hadn't yet experienced with any other language. I have also learned more about how amazing the rust compiler is. The rust compiler provides a level of detail that I didn't realize was possible with how well it describes the causes of compilation issues. It provides suggestions for changes of code to best practices, accompanied by links to the appropriate documentation pages. It is hyper-specific about what in your code is causing each of the issues, and while I have found that the

amount of effort required is greater from writing in rust, the reward when you have worked through your compiler errors and the code finally runs, to me at least, feels much greater.

Useful Documentation Links Used in Coding:

- *How to write documentation - The rustdoc book*. (n.d.). https://doc.rust-lang.org/rustdoc/how-to-write-documentation.html
- *Introduction - The Rust Style Guide*. (n.d.). https://doc.rust-lang.org/nightly/style-guide/
- *Error handling - the Rust programming language*. (n.d.). https://doc.rust-lang.org/book/ch09-00-error-handling.html
- *macro_rules! - Rust By Example*. (n.d.). https://doc.rust-lang.org/rust-by-example/macros.html
- *Attributes - the rust reference*. (n.d.). https://doc.rust-lang.org/reference/attributes.html

- *Closures: Anonymous Functions that Capture Their Environment - The Rust Programming Language*. (n.d.). https://doc.rust-lang.org/nightly/book/ch13-01-closures.html
- *Advanced Functions and Closures - the Rust Programming Language*. (n.d.). https://doc.rust-lang.org/book/ch19-05-advanced-functions-and-closures.html
- *Using Box to point to data on the Heap - the Rust programming language*. (n.d.). https://doc.rust-lang.org/book/ch15-01-box.html
- *C - Derivable traits - the Rust programming language*. (n.d.). https://doc.rust-lang.org/book/appendix-03-derivable-traits.html

Videos:

- "My favorite Rust design pattern"
  - https://www.youtube.com/watch?v=qrf52BVaZM8
- "Rust Data Modeling Without Classes" (enums/match)
  - https://www.youtube.com/watch?v=z-0-bbc80JM&t=10s
- "Rust Error Handling - Best Practices"
  - https://www.youtube.com/watch?v=j-VQCYP7wyw