**Final Documentation**

**for**

**Price Comparison Application**

Capstone: Fall 2012

Lucy Horpedahl, Nathan Noonan, Daniel O’Doherty, and Nick Richardson

**Table of Contents**

**Table of Contents** 2

**1.** **Introduction** 3

**1.1 Client** 3

**1.2 Application** 3

**2.** **Functional and Nonfunctional Requirements** 5

**2.1 Functional Requirements** 5

**2.2 Nonfunctional Requirements** 13

**3.** **Architecture and Design** 17

**3.1 High-Level Architecture** 17

**3.2 Low-Level Architecture** 19

**4.** **Implementation Issues** 24

**4.1 Application scope** 24

**4.2 Amazon code** 24

**4.3 Database connection** 25

**5.** **Testing** 26

**5.1 Connections** 26

**5.2 Data processing** 26

**5.3 GUI interaction** 27

**6.** **Lessons Learned** 30

**6.1 CS background** 30

**6.2 Changes** 31

**6.3 Future extensions** 32

**Index of Figures** 33

**1.** **Introduction**

**1.1** **Client**

The client for the application we chose to design for this project is Justin Richardson, the owner of a small business, Digital Attitude. The business, which is located in Crossroads Mall, sells Japanese animation collectibles, movies, books, video games, and more.

We spent time at Mr. Richardson’s business, learning about how the store runs. Most, if not all, of his products have barcodes which are scanned in order to read the UPC number of the item. The items are entered into a database and organized by this number so that searching for this number will return the proper item. These barcodes are central to both our original application idea and our updated application.

**1.2** **Application**

**1.2.1** **Original Proposal**

Our initial plan was to design a new point-of-sale system for Mr. Richardson’s store. A point-of-sale system handles nearly all of the necessary operations of a retail business. It allows inventory to be searched and updated, purchases to be made, and customer information to be stored.

We believed that Mr. Richardson did not have a proper point-of-sale system, and we planned to create one that would fit his business’s unique needs. For example, Mr. Richardson buys used items from customers and then sells them in his store. He selects the pricing for these transactions by searching Amazon.com, which is one of the few places that sells most of the items that Mr. Richardson sells in his store.

We planned to implement, as part of the point-of-sale system, a “price checker” that would search Amazon.com and display the current price for the items so that Mr. Richardson would not have to do these searches manually. Upon meeting with Mr. Richardson further, we discovered that his point-of-sale system was adequate, but he told us that the “price checker” feature would be very helpful.

**1.2.2 Modified Proposal**

Because the point-of-sale system would have been a very large project, and the system currently used by the client was sufficient for his needs, we decided to narrow the scope of our project to just design an application that would search Amazon.com and retrieve prices for items in Mr. Richardson’s inventory. We planned to include options for filtering and sorting the output as well as for making changes to the prices in the database through the application itself, rather than having to go back to the point-of-sale system to modify the prices.

The application would need to communicate with Mr. Richardson’s existing database, process the data to retrieve the UPC numbers, communicate with Amazon.com to search for the UPC numbers and retrieve the prices, process the prices to display them on an interface, and ultimately communicate with the database again to write new prices back to the database.

Ultimately, we ran across a number of difficulties in designing this application which caused us to modify certain features, but the basic architecture would remain the same. The rest of this document describes in more detail the work that was done to design this application.

**2.** **Functional and Nonfunctional Requirements**

**2.1 Functional Requirements**

**2.1.1 Database Communication**

The application will need to have a connection to the client’s database which contains the store’s inventory.

**2.1.1.1 Read from the database**

The application will need to be able to query the database using SQL to obtain a list of all used products in the client’s database.

**2.1.1.1.1 Use Case**

**Name:** Read from database

**Actors:** Initiated by user

**Entry Condition:** User launches application

**Flow of Events:**

1. The application will attempt to establish a connection with the database.
2. The application will pass a query to the database requesting a list of items.
3. The database will return the result of the query to the application.

**Exit Conditions:** The application will have a list of items.

**Exceptions:**

* The database is not found or is inaccessible.
* The query returns no results.

**2.1.1.2 Write to the database**

The application will need to be able to write to the database by using SQL queries to modify prices of items.

**2.1.1.2.1 Use Case**

**Name:** Write to database

**Actors:** Initiated by user

**Entry Condition:** User chooses to modify price(s)

**Flow of Events:**

1. The application will attempt to establish a connection with the database.
2. The application will pass a query to the database containing the item(s) and the new price(s) based on entry in the interface.
3. The database will acknowledge the changes made by the application.

**Exit Conditions:** The database will reflect the price change(s).

**Exceptions:**

* The database is not found or is inaccessible.
* The SQL query returns an error.

**2.1.2 Online Shop Communication**

**2.1.2.1 Obtain information from online shop**

The application will require a connection to the internet in order to connect to an online shop, in this case Google shops. Once the application has a list of items from the client’s database and the barcodes have been extracted, a request will be sent to the online shop for each item on the list. A list of all the offerings found in the shop for the barcode sent in the request will be returned. If a barcode is invalid or there are no results for an item, nothing will be returned and the average will be 0. In this case, the average online price will be set to “N/A” to indicate there were no offers, instead of setting it to 0, which seems like the item is free.

**2.1.2.1.1 Use Case**

**Name:** Obtain information from Google shops

**Actors:** Initiated by application

**Entry Condition:** Application has a list of barcodes from the database

**Flow of Events:**

1. The application will attempt to establish a connection to the internet.
2. The application will send a request to Google shops with a specific barcode.
3. Google shops will return a list of all offerings for the item.
4. Steps 2 and 3 will be repeated until all items have been checked.

**Exit Conditions:** The application will have a list of offerings for each item in the list obtained from the database.

**Exceptions:**

* There is no internet connection.
* A barcode is invalid.
* Nothing is returned.

**2.1.3 Data Processing**

**2.1.3.1 Extract real barcodes from database results**

When the results are returned from the database, the barcodes will need to be extracted because the client adds digits to the beginning of the barcodes for used items, which are the items which will be handled. If the barcode contains too many digits, the first two will be removed. Record from the database will always contain a barcode because it is the primary key of the inventory table.

**2.1.3.1.1 Use Case**

**Name:** Extract real barcodes

**Actors:** Initiated by application

**Entry Condition:** A list of items has been returned from the database

**Flow of Events:**

1. The barcode contained in a record from the database will be checked for length.
2. If the barcode is too long, the first two digits will be removed.
3. Steps 1 and 2 will be repeated until all records have been processed.

**Exit Conditions:** All records contain real barcodes for the items.

**Exceptions:**

* A barcode is of an unexpected length.

**2.1.3.2 Parse online shop results**

When a result is received from the online shop, the price and condition for each offer will need to be parsed from the data returned. Because the results are formatted in a consistent way, regular expressions can be used. The results will be parsed to find the price for all offers of an item listed as “used.” If there are no results for an item, nothing will be returned and the average will be set to 0. In this case, the field will be set to “N/A” to indicate there were no offers.

**2.1.3.2.1 Use Case**

**Name:** Parse online shop results

**Actors:** Initiated by application

**Entry Condition:** The online shop has returned a result for a barcode.

**Flow of Events:**

1. The result will be parsed to find the price and condition for each offer.
2. If the condition is “used” then the price will be saved into a list for later analysis.

**Exit Conditions:** The result has been completely parsed, and a list of used prices has been obtained.

**Exceptions:**

* The result is empty.
* The result is not formatted correctly.
* The result contains no used items.

**2.1.3.3 Analyze prices**

The list of used prices obtained from the online shop will need to be analyzed in order for the data to be presented in a simple way on the interface. Any outliers will be removed from the list so that the average is not skewed by extremely high or extremely low values. If the average is 0, the average online price will be set to “N/A” to indicate that there were on offers.

**2.1.3.3.1 Use Case**

**Name:** Analyze prices

**Actors:** Initiated by application

**Entry Condition:** A list of used prices for an item has been obtained.

**Flow of Events:**

1. The average and standard deviation of the list will be calculated.
2. If the list contains any outliers, they will be removed. Formally outliers are defined as any value three or more standard deviations away from the average, but for our purposes two standard deviations will be used because of the smaller set and relatively small numbers.
3. A new average will be calculated.

**Exit Conditions:** The item will have an average online price.

**Exceptions:**

* There are no prices in the list.
* The prices in the list are all $0.

**2.1.4 Graphical User Interface Functions**

**2.1.4.1 Change the number of results on screen**

The user is able to change the number of items displayed on the screen at one time. If there are no records to display, a message indicating this will be displayed. If there are not enough records to display the full number, however many items are available will be displayed on the screen.

**2.1.4.1.1 Use Case**

**Name:** Change number of results on screen

**Actors:** Initiated by user

**Entry Condition:** Some results are displayed on the interface.

**Flow of Events:**

1. The user selects one of the radio buttons in the “Num Results” area.
2. The displayed table will be modified to show the selected number of results.

**Exit Conditions:** Some results are displayed on the interface.

**Exceptions:**

* There are no results displayed.
* There are not enough results to display the number requested.

**2.1.4.2 Next/Previous**

If more results exist than are displayed on the interface, “Next” and “Previous” can be used to display more results. If “Next” is pressed on the last page, no page change will be made; if “Previous” is pressed on the first page, no page change will be made.

**2.1.4.2.1 Use Case**

**Name:** Next/Previous

**Actors:** Initiated by user

**Entry Condition:** Some results are displayed on the interface.

**Flow of Events:**

1. The user clicks the “Next” or “Previous” button.
2. The next *x* number of records or previous *x* number of records are displayed.

**Exit Conditions:** Some results are displayed on the interface.

**Exceptions:**

* There are no more results to be displayed.

**2.1.4.3 Sort results**

The user can choose to sort the data in a number of different ways: Price low-to-high, Price high-to-low, Difference negative-to-positive, Difference positive-to-negative. Price low-to-high and high-to-low will sort the results by the database price. Difference negative-to-positive and positive-to-negative will sort the results by the difference between the average online price and database price. When two items have the same price or difference, they will be sorted by barcode.

**2.1.4.3.1 Use Case**

**Name:** Sort results

**Actors:** Initiated by user

**Entry Condition:** Some results are displayed on the interface.

**Flow of Events:**

1. The user selects one of the “Sort by” radio buttons.
2. The results are sorted and displayed on the interface.

**Exit Conditions:** The results are displayed in the new sorted order.

**Exceptions:** none

**2.1.4.4 Filter results**

The user can choose to filter the results by showing only items that have online average prices, only items that are more expensive online than in the database, or only items that are less expensive online than in the database. The results can also be displayed without any filter.

**2.1.4.4.1 Use Case**

**Name:** Filter results

**Actors:** Initiated by user

**Entry Condition:** Some results are displayed on the interface.

**Flow of Events:**

1. The user selects one of the radio buttons in the “Filter results” area.
2. The selected filter is applied to the results.
3. The filtered set is displayed on the interface.

**Exit Conditions:** The results that fit into the selected filter are displayed.

**Exceptions:**

* There are no results that fit into the selected filter.

**2.1.4.5 Modify item prices in database**

The user can modify prices of items in the database from the interface itself. By clicking on the “Database Price” field in the table and changing the value, the price will be changed in the database, and then results will be re-sorted and filtered based on the current settings. Before the new price is written to the database any non-numeric characters are removed from the text. The decimal point is also left in the string, of course.

**2.1.4.5.1 Use Case**

**Name:** Modify prices

**Actors:** Initiated by user

**Entry Condition:** Some results are displayed on the interface.

**Flow of Events:**

1. The user clicks on the “Database Price” field for the item whose price is to be modified.
2. The user enters the new price in the field and hits “enter.”
3. A connection is made with the database.
4. The new prices are written to the database for the selected items.

**Exit Conditions:** The price for the item is modified in the database.

**Exceptions:**

* A connection cannot be made with the database.
* The user enters non-numeric characters.
* The user enters an invalid number (e.g., 0 or negative numbers).

**2.2 Nonfunctional Requirements**

**2.2.1 Hardware and Software Requirements**

The application will need to be run on a PC that has a connection to the client’s database via MS SQL Server and an internet connection. No other specific hardware or software is required.

**2.2.2 Reliability**

In order to execute all functionality, the application will rely upon a connection to the client’s database and a connection to the internet.

**2.2.3 Availability**

Because the application is not safety-critical, its availability needs are less imperative. However, that does not mean that the system should not be available as much as possible. The application should be available particularly during the client’s business hours. Also, the application requires access to the internet, but it should not rely on an internet connection to run, only to update its information. In the event of a loss of internet connection, the application should still display previously obtained information, and it should continue to be connected to the database and able to modify the client’s inventory as desired.

**2.2.4 Security**

Because the application will have access to read from and write to the client’s database, it should be secured from use by unauthorized persons. The pricing information obtained online is publicly available, so there is no security concern for that data. The client’s PC will be the only system running the application, and the client’s own security measures should ensure the security of the application.

**2.2.5 Maintainability**

The application should be relatively autonomous once development is complete. It should not need updates on a regular basis. If updates need to be performed, the application could most likely be updated outside of the client’s normal business hours to prevent impeding the client’s business.

**2.2.6 Portability**

The application will be designed for this particular client and is not being distributed to anyone other than the client. After development and deployment, there will be no need for the application to run on any system other than the client’s.

**3.** **Architecture and Design**

**3.1 High-Level Architecture**

The Price Comparison Application had to be designed to be able to connect to existing applications including the client’s inventory database and online shops.

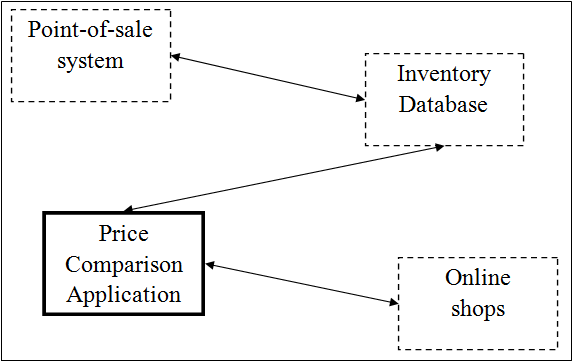


Figure 1: Price Comparison Application Architecture

*Figure 1* shows how the price comparison application interacts with the existing modules. The application must communicate with the existing database in order to read and modify records in the inventory. The database also communicates with the point-of-sale system, and the changes made by the price comparison application must be reflected in the point-of-sale system.

The price comparison application must also communicate with Google shops in order to search for specific barcodes and retrieve the price at which the item is being sold by various merchants.

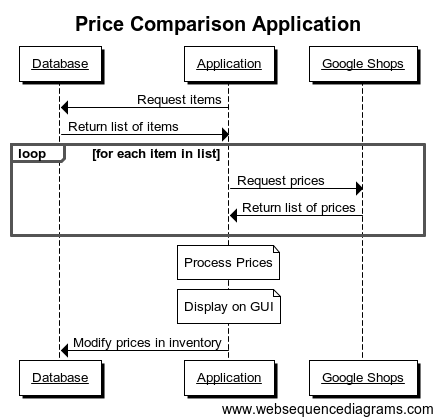


Figure 2: Price Comparison Application Sequence Diagram

*Figure 2* shows more detail of the interactions between the price comparison application and the external applications.

First, the application will retrieve all used items from the client’s database. When it has a list of barcodes, the application will send requests to Google shops for each item.

Each request will return a list of offerings for the item, which will tell how much the item is being sold for at various locations online. After retrieving this list, the application will perform statistical calculations to find an average price for each item.

The average online prices, as well as the current price in the client’s database, will be displayed on an interface that will allow the client’s database to be modified if the user chooses.

**3.2 Low-Level Architecture**

This section discusses the architecture of the application itself. It also includes an explanation of the graphical user interface.

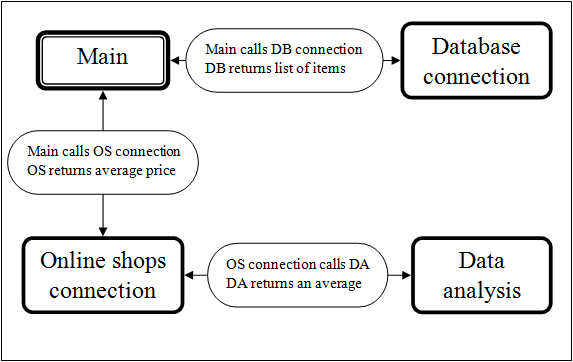


Figure 3: Call-and-return structure of Price Comparison Application

The application will contain a number of separate modules to perform each of the necessary actions. The main module will call the others. First it calls a module to connect to the database and return a list of all used items in the inventory. Next the list is passed to a module that connects to the online shops and obtains a list of prices for each item. This module calls a module that analyzes the list for each item and returns a single average for each. *Figure 3* shows this call-and-return structure.

**3.2.1 Graphical User Interface**

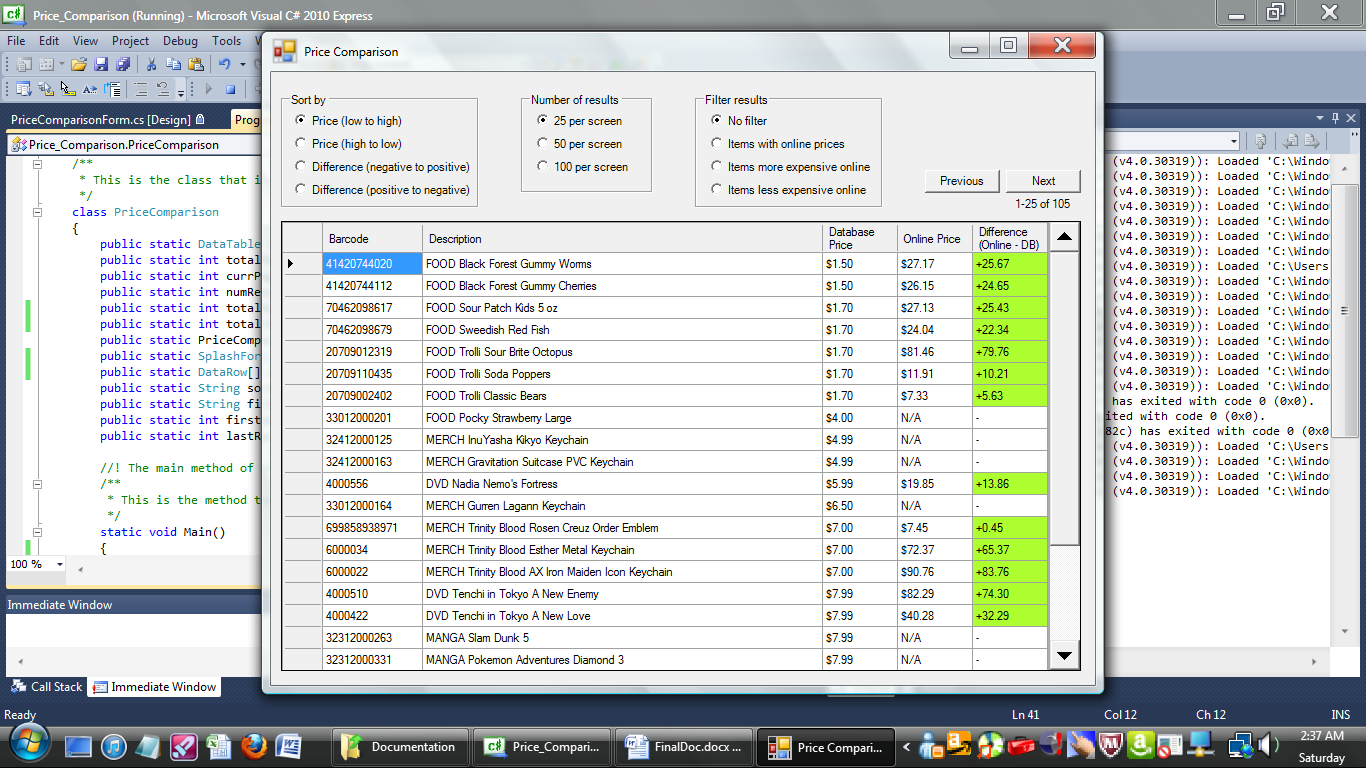


Figure 4: Price Comparison Application GUI Screenshot

The main interaction between the user and the application will be through the graphical user interface or GUI. *Figure 4* shows a screenshot of the graphical user interface for the price comparison application. The individual components will be discussed in more detail below.

**3.2.1.1 *Sort by* section**

This section contains four radio buttons. The buttons represent different ordering options for the display of the results on the screen.

* **Price**: The prices used for this sorting option will be the prices in the client’s database.
  + *Low-to-high*: items with the lowest prices will be displayed first.
  + *Low-to-high*: items with the highest prices will be displayed first.
* **Difference**: For this sorting option, the difference between the client’s price and the average online price will be calculated (average online price minus client’s price).
  + *Negative-to-positive*: items with the most negative difference will be displayed first.
  + *Positive-to-negative*: items with the most positive difference will be displayed first.

**3.2.1.2 *Number of results* section**

This section contains three options represented with radio buttons that will allow the user to select how many items will be displayed on a single page.

* **25 per screen**: 25 items will be displayed in the results table.
* **50 per screen**: 50 items will be displayed in the results table.
* **100 per screen**: 100 items will be displayed in the results table.

**3.2.1.3 *Filter results* section**

This section contains three options represented with radio buttons that allow the user to filter the results displayed in the table and reduce the size of the result set.

* **No filter**: All results will be displayed.
* **Items with online prices**: Only those items that have an average online price will be displayed.
* **Items more expensive online**: Only those items whose average online price is higher than the price in the client’s database will be displayed.
* **Items less expensive online**: Only those items whose average online price is lower than the price in the client’s database will be displayed.

**3.2.1.5 *Previous*/*Next* buttons**

The *Next* and *Previous* buttons allow the user to display the next *x* items from the results set or the previous *x* items from the results set respectively, where *x* is the number of results currently displayed on the screen based on the *Number of results* options.

**3.2.1.6 Results table**

The results table is where the individual items in the result set will be displayed. The table is of variable size so that any of the *Number of results* options can be displayed within it.

* *Barcode*: the first column will display the barcode of the item.
* *Description*: the second column will contain a name or short description of the item.
* *Database Price*: the third column will display the current price of the item in the client’s database; if the user modifies the price in this field, the database will be updated if the number is valid (i.e., a positive, non-zero, numerical string) and the new price should be reflected in the table; all current sorting and filtering options should be executed using the new price—this mean that an item may move or even disappear from the current screen if it no longer appears within the filter or in the same location in the sort.
* *Online Price*: the fourth column will display the average online price that is calculated within the application.
* *Difference*: the final column will display the difference (*Online Price* – *Database Price*); if the difference is negative, the cell will be colored red; if the difference is positive, the cell will be colored green.

**4.** **Implementation Issues**

In the development process for this application, a number of issues have surfaced. The three main ones are the scope of the application, the use of Amazon Web Services code, and connection to a database on all developers’ machines.

**4.1 Application scope**

**4.1.1 Issue**

Our first problem arose at the beginning of the high-level development phase. The original plan, to design an entire point-of-sale system, seemed both too large and not necessary. When outlining the necessary components of a point-of-sale system, the application would be much larger than we had originally though, and the client’s current point-of-sale system was better than expected.

**4.1.2 Solution**

We decided that the most useful and unique function of the planned point-of-sale system was a price comparison component and it did not necessarily need to be integrated into the point-of-sale system, so the scope of the project was reduced.

**4.2 Amazon code**

**4.2.1 Issue**

The next problem we encountered was in accessing Amazon’s online prices. We found an API through Amazon Associates Web Services that contained methods like ItemLookup() and ItemSearch() and seemed to be just what we needed for our application. After discovering and handling a number of exceptions, including configuration changes and trying both ItemLookup() and ItemSearch(), we ultimately could not get any results from the method calls and only got the not very informative message: Err: The remote server returned an unexpected response: (400) Bad Request.

**4.2.2 Solution**

After a great deal of online searching with few relevant results, we decided to research Google shops’ API and found it to be much simpler and more user-friendly than Amazon’s, so we switched to searching Google’s prices instead of Amazon’s.

**4.3 Database connection**

**4.3.1 Issue**

Finally, we had trouble getting our application designed in Microsoft Visual C# 2010 Express to communicate with a database in Microsoft SQL Server Management Studio. Dan was able to set it up on his machine and write instructions for the rest of us to follow, but no one else was able to connect to their database, even when it was set up with the settings in his instructions. A number of solutions were tried: changing the privileges for the database, modifying the security type in the connection string, and varying the data source between “localhost” and some variation of the server name on the user’s machine.

**4.3.2 Solution**

Ultimately, it was decided that the application was only going to be run on the client’s PC and would not need to be connected to more than one server, so in the interest of advancing to the next stage of the development, we hardcoded some records into the application simply for the development phase.

**5.** **Testing**

**5.1 Connections**

**5.1.1 Read from the database**

**5.1.1.1 Successful read**

**5.1.1.2 The database is not found or is inaccessible**

**5.1.1.3 The query returns no results**

**5.1.2 Write to database**

**5.1.2.1 Successful write**

**5.1.2.2 The database is not found or is inaccessible**

**5.1.2.3 The SQL query returns an error**

**5.1.3 Access Google shops**

**5.1.3.1 Successfully access Google shops**

**5.1.3.2 There is no internet connection**

**5.1.3.3 A barcode is invalid**

**5.1.3.4 Nothing is returned**

**5.2 Data processing**

**5.2.1 Extract real barcodes**

**5.2.1.1 Successfully extract barcode**

**5.2.1.2 A barcode is of an unexpected length**

**5.2.2 Parse Google shops results**

**5.2.2.1 Successfully parse Google shops results**

**5.2.2.2 The result is empty**

**5.2.2.3 The result is not formatted correctly**

**5.2.2.4 The result contains no used items**

**5.2.3 Analyze prices**

**5.2.3.1 Calculate the mean of the prices and remove any two standard deviations or more from the mean**

**5.2.3.2 There are no prices in the list**

**5.2.3.3 The prices in the list are all $0**

**5.3 GUI interaction**

**5.3.1 Change number of results on screen**

**5.3.1.1 Show 25 records per screen**

**5.3.1.2 Show 50 records per screen**

**5.3.1.3 Show 100 records per screen**

**5.3.1.4 There are no results displayed**

**5.3.1.5 There are not enough results to display the number requested**

**5.3.2 Next**

**5.3.2.1 Successfully display next set of results**

**5.3.2.3 There are no more results to be displayed**

**5.3.3 Previous**

**5.3.3.1 Successfully display previous set of results**

**5.3.3.3 There are no more results to be displayed**

**5.3.4 Sort results**

**5.3.4.1 Sort by price, low-to-high**

**5.3.4.2 Sort by price, high-to-low**

**5.3.4.3 Sort by difference between online and database prices, negative-to-positive**

**5.3.4.4 Sort by difference between online and database prices, positive-to-negative**

**5.3.5 Filter results**

**5.3.5.1 Show results with no filter**

**5.3.5.2 Show results that have an average online price**

**5.3.5.3 Show results where the online price is higher**

**5.3.5.4 Show results where the online price is lower**

**5.3.5.5 There are no results that fit into the selected filter**

**5.3.6 Modify item prices in database**

**5.3.6.1 Successfully modify item prices in database**

**5.3.6.2 A connection cannot be made with the database**

**5.3.6.3 The user enters non-numeric characters**

**5.3.6.4 The user enters an invalid number**

**6. Lessons Learned**

**6.1 CS background**

In this project, many of the courses that we have taken throughout our computer science degree have proven useful. This project has shown us that even this set of knowledge is only a fraction of the computer science concepts that will be useful to us in our future careers.

Since our application interfaces with a database, concepts learned in database courses have been useful for development. In particular, SQL is the language used to access our database, and we had already learned some SQL in the courses on database management. It has also helped us to understand the structure and architecture of the database we are using.

Many of the concepts learned in software engineering have been relevant to our project. Much of the documentation that we learned about in that course has been required for this project. The use cases and diagrams that we had to make for our Software Requirements Specification document were used as models for the use cases and diagrams included in this document. We were also able to use some of the things we learned about project management in assigning tasks to individual team members and keeping track of the work to be done.

Our application required parsing large chunks of data obtained from online shops, which would be nearly impossible without regular expressions. Knowing regular expressions helped us to quickly and cleanly find the information we needed in the results of our request.

Of course, it would be impossible to mention all of the concepts that we learned in our computer science courses that have been useful throughout this project. You can never know when a concept will become relevant, and we have found that so much we learned was invaluable for this project and will presumably be invaluable at some point in our future careers.

**6.2 Changes**

If we were going to do this project over, there would be a number of things that could be changed to improve the efficiency of our work. The first thing we would change would be to start with the narrower project scope so that we could have spent more time focusing on the features of the price comparison application instead of the entire point-of-sale system. We spent a couple of weeks trying to figure out how to implement and organize the whole point-of-sale system, which we ended up discarding when we decided to focus on the price comparison application alone.

The next thing would probably be changing the order in which the application was developed. Since connecting to the database and the online shop turned out to be the most time consuming, we probably would have started with sample data hardcoded into the application and gotten the interface working; we would get the data to be displayed on the interface, and get all of the features of the interface working, except changing the prices, since that would require a connection to the database. After the interface was fully functional, we could move on to the database and online shop connections. Since we did not do that, the features of the interface were implemented last, making that work feel very rushed, even though it was the most straightforward part of the application.

Another thing that would have been helpful would have been to have some real sample data from the client’s database at an earlier point in the project. Since we were not able to actually copy the client’s database due to permission issues, this was a time-consuming task, but having that information early on would have been invaluable to testing as well as implementation.

**6.3 Future extensions**

In addition to the features we have included in our application, there are other features that could be added to enhance its functionality.

Given more time, the Amazon Associates Web Services API could have been very useful for the application. If we were going to add to the project, implementing the Amazon API would be a great addition. Also, if we could find a way to obtain Amazon’s prices, it would be useful to find other online sources, such as eBay, to compare with the client’s inventory. The interface could be expanded to show average prices from each of the different sources as well as an average for all the sources together.

The interface could always be expanded to include more information about the products, such as product type (DVD, book, video game, etc.), a cover image, more detailed descriptions, and anything else that is stored in the client’s database.

Another interesting feature would be to weight the online prices based on the condition of the items. If a product is listed as in “good” condition, its price should be weighted more than a product listed as in “poor” condition. This would require defining weights for specific conditions, which was something that we did not end up having time to do. However, it was a feature that we considered, and it would most likely give a more accurate average price.

The table could also be modified to display another field indicating the number of offers the online average is based on, which would be something of a sign of the accuracy of the average. Another useful feature would be having a button or link next to each item that opens up a window with a tabular view of all the offers found online so that the individual offers could be viewed. It was also be helpful for the user to be able to mark one or more of those offers as incorrect if it is not for the right item.

**Index of Figures**

fig. 1: Price Comparison Application Architecture 17

fig. 2: Price Comparison Application Sequence Diagram 18

fig. 3: Call-and-return structure of Price Comparison Application 19

fig. 4: Price Comparison Application GUI Screenshot 20