# Project 2: Design and Planning Document

**for**

# Small-Business Unified Database Operations with Online Access

**Prepared by Alex Bonomo**
**Grace Park**
**Paul Servino**
**Supritha Sundaram**
**Neil Tilley**
**Hanako Ueda**
**Richard Van**
**Max Vujovic**

**for CS 130 Spring ’09, UCLA**

**April 26, 2009**

# Table of Contents



# Document Revision History

Rev. 1.0 2009-04-23 – initial version
Rev. 2.0 2009-04-26 – update by all team members

# 1. System Architecture

The system architecture is built on the following conceptual models:

1. A repository model, handling internal application processing, and
2. A client-server system, which manages external user interactions.

## 1.1 Repository Model

The repository model encompasses a central storage location for all information that is accessed by multiple nodes. The repository for our system is a mySQL database. In this solution, nodes are logged-in users having access through a web-based application. Through this application the graphical user interface displays a collection of individual components for receiving user input. A user logs into the application, creating a **session** (further detailed in **Section 2.6**). A session functions in the role of a "blackboard", a place in a temporarily created database that is identified and defined by each separate interface (each open browser window). To this blackboard come incremental updates, eventually leading to a full transaction. Calculated information from the session is committed only at the last step from the session to a permanent mySQL database. The primary benefit of this approach is that user inputs remain cached between individual interfaces (i.e., successive web pages). Importantly, session data is processed and filtered before being committed in a safe fashion to the database.

### 1.1.1 Repository Model Diagram

The following diagram (**Figure 1.1.1)** gives a representation of the repository model. A session starts when a user logs in. Adds or Edits to a Component add to, or revise, the session data. Successful changes between successive pages of the interface update the session data until the user commits a finalized transaction. Once committed, the session data immediately updates the database.

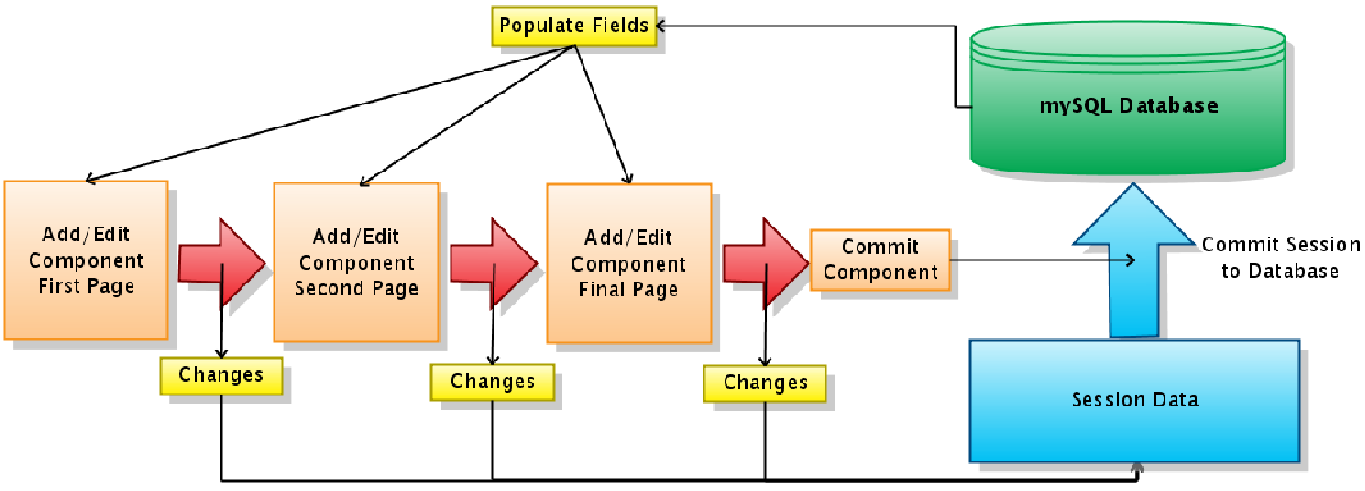


**Figure 1.1.1:** Repository diagram representing data flow of the system.

## 1.2 Client-Server Model

The client-server model consists of a distributed organization, where servers provide services (e.g data storage), and clients call on services provided by the server for source information or processing requests. The design of the web application in this project employs this model.

A user interacts with the web application through an HTML protocol. The application processes user input in PHP, which it passes to the session, and finally the application interacts with the database using mySQL query language.

Additionally, the client adheres to the thin-client model, meaning the bulk of the processing and storage is on the server. Placing all processing of session data and database management on the server reduces computation loads on user machines. Thus the load on the user's computer is only the display of the graphical user interface. Although occurrences of latency are generally more common due to server processing and network traffic, the overall benefit of this design is the flexibility and reliability of having a minimized web application functionality, in cases where the capabilities of the customer machines are unknown.

### 1.2.1 Client-server Model Diagram

The following diagram (**Figure 1.2.1)** gives a display of the thin-client model. From the user's perspective, only the graphical user interface is seen, while the server handles data management and application processing.

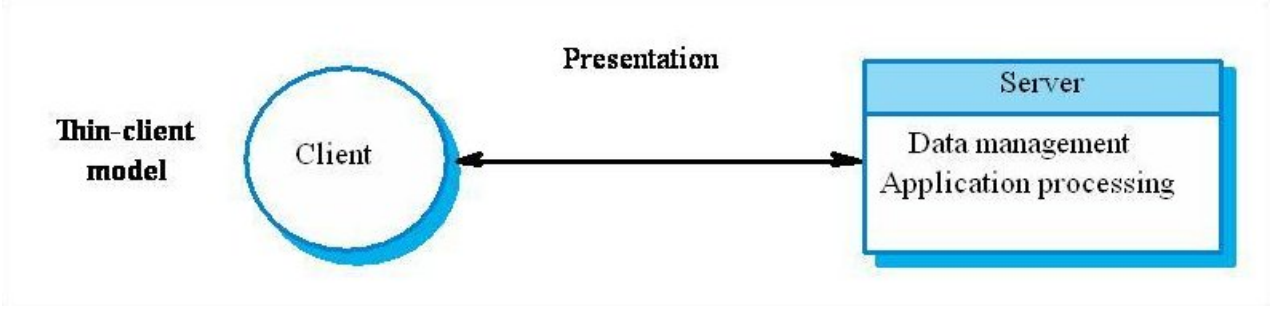


**Figure 1.2.1:** Thin-Client Server Model diagram portraying external user interactions.

# 2. Design Details

## 2.1 Database Object Definition

The system database consists of tables with columns representing fields of objects and rows that identify individual objects. One table corresponds to each object. The following diagram (**Figure 2.1.1)** outlines the relationships between most of the system's objects and their corresponding tables.

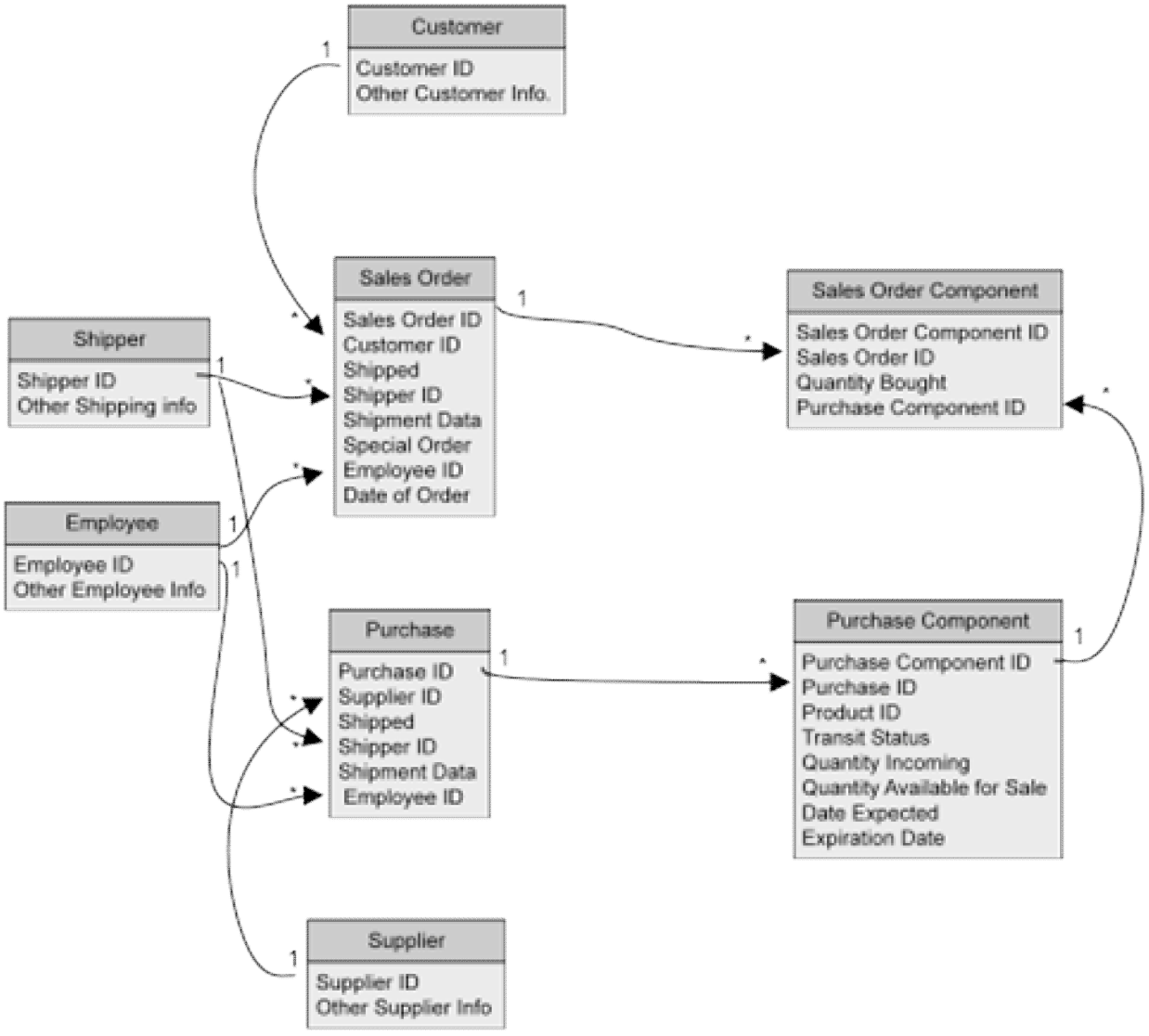


**Figure 2.1.1**: Conceptualized schematic of the major objects comprising the database of the business enterprise. Each rectangle conveys an object with the name shown at the top and having data fields listed beneath. Arrows indicate relations between one object and another, whether one-to-one (1..1), one-to-many (1..*), or many-to-many (*..*). The heads of arrows typically indicate many object belong to the object where the arrow originates.

## 2.2 Major Objects and Tables Definition

The user must have capabilities to view, add, edit, and delete information associated with individual objects. The user must also be able to view a list of objects defined by some given search criteria and display them in various orders as desired. Componentized major objects consist of a base set of data and one or several component objects. In other words, a purchase consists of one or several **purchase components**, which are objects each having its own table. Though separate from the major object (e.g., a purchase), the two are linked through a primary key.

Sample major objects include the following:

- Customer
- Employee
- Supplier
- Shipper
- Product

Componentized major objects include

- Purchase (which consists of one or more PurchaseComponents)
- Sales Order (which consists of one or more SalesOrderComponents)

## 2.3 Graphical User Interface

The graphical user interface consists of a stable display (a **homepage**) where the following will appear on every screen: search box, navigation tabs, logo, and a logout option. Additionally, a variable inner area on the page can change to display the following options: user accounts edit page (only accessible to Administrator-level user); edit employee profile; report generator; customer table; supplier table; shippers table; products table; purchases table; and sales table.

Searches initiated in the Search Box on the homepage search all database tables. Searches done in the inner area of Tabs 4-10, which reference specific objects, search only the respective tables affiliated with the Tab choice.

### 2.3.1 GUI Diagram

The following diagram (**Figure 2.3.1**) describes the workflow when a user clicks different navigation tabs or navigation options:

- Tab 1 – directs the user to the Account Edit Page;
- Tab 2 – directs the user to the Edit Employee Profile Page;
- Tab 3 – will direct the user to the report generator;
- Tabs 4-10 – act on “objects” which are similar in functionality. Each object (e.g., Customer, Supplier, Shipper, Purchases, Sales, or Products) is associated with a different table on the database.

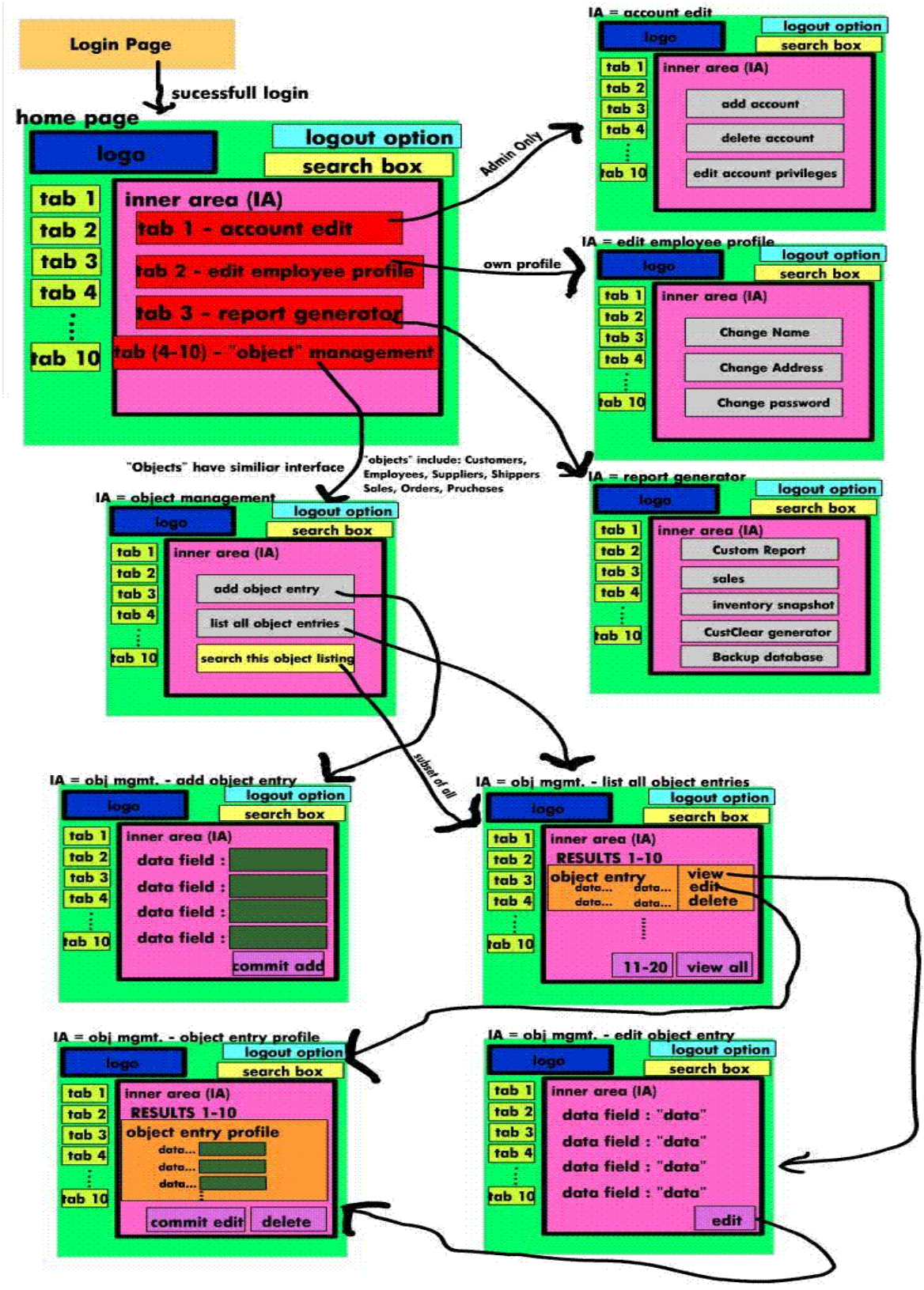


**Figure 2.3.1:** GUI diagram portraying interface behavior seen by user. Arrows indicate navigational flow of linked pages.

## 2.4 Critical Algorithms

Critical algorithms are basic functionalities of operating and managing retail orders and transactions of a database system. For this application they are expressed as mySQL queries. Sample queries appear below as illustrations.

The PHP application sends queries as commands to the database server, and the server returns calculated arrays of data, in some cases in a given desired sorted order. The PHP application receiving the data displays a user interface showing the received data in formatted HTML code. The PHP application itself does minimal logical computation, except for determining user interface characteristics such as item placement on a page.

### 2.4.1 Algorithms to Process and Manipulate Simple Major Objects

Primary objects include Customer, Employee, Supplier, Shipper, and Product. Because all these generally are processed in similar fashion, one example is sufficient to illustrate the pattern.

Where an ellipsis (…) appears, this represents the fetching or updating of fields, or the insertion of additional fields (table columns) associated with the objects.

#### *2.4.1.1 View Individual Customer (or Employee / Supplier / Shipper / Product)*

- SELECT first_name, last_name, … FROM customers WHERE id=$cust_id

#### *2.4.1.2 Add Customer (Employee / Supplier / Shipper / Product)*

- INSERT INTO customers (first_name, last_name, …) VALUES ($first_name, $last_name…)

#### *2.4.1.3 Edit Customer (Employee / Supplier / Shipper / Product)*

- UPDATE customers SET first_name=$first_name, last_name=$last_name, … WHERE id=$cust_id

#### *2.4.1.4 Delete Customer (Employee / Supplier / Shipper / Product)*

- **Step 1**: Upon user stimulus, flag entry as “Trash” and set a delete time several days in advance (e.g. 3, for 3 calendar days):

  UPDATE customers SET trash=1, delete_time=ADDTIME(NOW(),’3:0:0:0’ )
  WHERE id=$cust_id

- **Step 2**: In a nightly “Chron Job”, delete all objects whose retention time has expired:

  DELETE FROM customers WHERE NOW() > delete_time

#### *2.4.1.5 List Customer (Employee / Supplier / Shipper / Product)*

- Example: List customers matching search terms:

  SELECT first_name, last_name, … ,
  MATCH(search_words) AGAINST ($search_words IN BOOLEAN MODE) AS relevance
  FROM customers
  WHERE relevance
  ORDER BY
  relevance

- Example: List newest customers:

  SELECT first_name, last_name, … ,
  FROM customer
  ORDER BY
  signup_date DESC

### 2.4.2 Algorithms to Process and Manipulate Componentized Major Objects

Componentized major objects include purchases and sales orders. The operations on these objects are more complex because a single purchase or sales order can be linked to many components and other objects. More complex algorithms are given below, shown as sample queries.

#### *2.4.2.1 View Individual Purchase ( / SalesOrder)*

- **Step 1**: Select major object base information.
  SELECT
  sales_orders.cust_id,
  sales_orders.shipper_id,
  sales_orders.emp_id,
  sales_orders.order_date,
  sales_orders.shipped, …
  FROM sales_orders
  LEFT OUTER JOIN customers ON sales_orders.cust_id = customers.id
  LEFT OUTER JOIN shippers ON sales_orders.shipper_id = shippers.id
  LEFT OUTER JOIN employees ON sales_orders.emp_id = employees.id
  WHERE sales_orders.id = $sales_order_id

- **Step 2**: Select components.
  SELECT
  sales_order_comps.quantity_purchased, …
  FROM sales_order_comps
  LEFT OUTER JOIN purchase_comps ON sales_order_comps.purchase_comp_id = purchase_comps.id
  LEFT OUTER JOIN products ON purchase_comps.product_id = products.id
  WHERE sales_order_comps.sales_order_id = $sales_order_id

#### *2.4.2.2 Add Purchase ( / SalesOrder)*

- **Step 1**: Insert major object base information.
- **Step 2**: Insert component information.

#### *2.4.2.3 Edit Purchase ( / SalesOrder)*

- **Step 1**: Edit major object base information.
- **Step 2**: Edit component information.

#### *2.4.2.4 Delete Purchase ( / SalesOrder)*

- **Step 1**: Delete components. (NOTE: reverse order from above transactions.)
- **Step 2**: Delete major object.

#### *2.4.2.5 List Purchase ( / SalesOrder)*

- Example: Generating real-time inventory.

  Algorithm: List available amount of each purchase component by subtracting quantity

bought from associated sales orders and temporary sales orders (items in employee's "shopping carts").

```
SELECT (purchase_comps.quantity_avail - SUM(sales_order_comps.quantity_bought) -
SUM(temp_sales_order_comps.quantity_bought) AS quantity_left,
purchase_comps.status,
purchase_comps.expiration_date, …
FROM purchase_comps, sales_order_comps, temp_sales_order_comps
LEFT OUTER JOIN products ON purchase_comps.product_id = products.id
GROUP BY sales_order_comps.purchase_comp_id,
temp_sales_order_comps.purchase_comp_id
```

### 2.4.3 Auxiliary Objects and Tables Definition

Auxiliary objects assist in the correct operation of the system and maintenance of key invariants. Auxiliary objects include

- TempSalesOrderComponent
- ActionLog

#### *2.4.3.1 Locking Inventory During Order Compilation (for* **Add** *and* **Delete** *TempSalesOrderComponents)*

When a user adds (**allocates**) a portion of a purchase component to a sales order but has not yet committed the transaction, a temporary sales order component is created. Other users who may wish to utilize the same supplier's inventory can then immediately see the temporary sales order component, in particular that it has a claim on part of that supply. Thus in issues of concurrency, other users receive updated notification of the availability from a given supplier. When the first user's order is committed, the temporary sales order component is deleted, replaced by a regular sales order component.

If the order is cancelled, any temporary sales order components are deleted also. Additionally, after a given fixed interval (e.g. 30 minutes) for lingering but unconfirmed orders, the temporary sales order component is automatically deleted. Users may extend the timeframe if needed, but they must complete an order within that timeframe. Optionally, if a user needs to save an incomplete order for later completion, the user can mark "In Progress" before clicking "Confirm" on the order review and confirm page.

#### *2.4.3.2 ActionLog:*

The Action Log is a comprehensive record of all interactions, for tracing all transactions in a business day. Every login and modifying request is logged, including the action performed, the time and date, the employee ID, the target object type, and the target object ID.

## 2.5 Protocols

Data transfer between database and user interface functions by standardized protocol communication. Protocol standards operate to connect PHP web pages through an opened web browser with the centralized mySQL data storage. In this way, input fields on a web page can correctly render the required data stored within the separate mySQL system. To guarantee accuracy and security and guard against corrupted data – or corrupting the entire database –, parts of the protocol must handle sanitizing the input, communication over a secure channel, and correct passing of data from page to page (in a multi-page transaction), as well as between web pages and the database.

### 2.5.1 .php Page Processing Protocol and Execution Flow

Below follows a high-level overview that describes how input from data fields on a submitted page is correctly received. Broadly speaking there are the following steps:

- `session_connect()`
  - Connect to session and database.
- `assert_permissions()`
  - If user does not have correct permissions to access page, redirect to login page.
- `get_input()`
  - Gets input from a form, URL, session, or database.
  - Inputs can include: a clicked page action, the next page, or user input.
- `verify_input()`
  - Checks for user or for untrusted input; converts it to a specific format.
  - Returns an error_message string.
  - If string is empty (no error), continues.
  - Otherwise, redisplays page with error_message.
- `process_input()`
  - Perform calculations and additional formatting on input.
  - Save processed information to database or session.
  - Clean up session (unregister consumed session variables).
- `show_output()`
  - Either: displays this page
    - Sanitizing output variables inline.
  - Or: redirects to the next page.

### 2.5.2 Protocol for Secure Communication

External to the operation of the mySQL Database are various communications protocols for extending the database interface across the Web. Primary concerns include security, reliability, and connectivity.

Encrypted data transfer governs both communication in an established session, followed by committing data to the database. Communication proceeds in the following manner. Entered data in any web page data fields are included as variables in an array called `$_SESSION` that is sent in a mySQL command to the database. The `$_SESSION` array is a superglobal array that is available for the client that requested the page. To store into the superglobal array, the web client takes the available form or URL input and sets a variable in the array to that value.

The included variable values are stored in the session file. For committing session data to the database, a final command calls to retrieve data associated with the current transaction, and the server delivers all session data connected to the session with that session ID. The client concatenates the session variables into a query, then passes the query string to a built-in PHP/mySQL send_query function. Effectively this moves data on the server in an active session from a temporary holding area into the permanent database.

## 2.6 Application Processing

The graphical user interface is the presentation layer over the internal architecture of the system. Each time a user logs in there is a **session** created, where from the application point-of-view, input is kept from state to state, and from the graphical user interface point-of-view, populated data fields are retained from page to page. Calculations are performed on the session data and the data are filtered to ensure valid entries. Only after all stages of a transaction do data from a session get committed to the database in the form of table entries.

### 2.6.1 Application Process Diagrams

The following diagrams (**Figures 2.6.1, 2.6.2**) describe the dataflow for a new session. Sessions are a record of input data. They are identified by a User ID. Each session incrementally builds up at each stage in the state diagram. The data contained in a session are **committed** at the last step when users select, for example, View/Edit/Add Employee, Place Purchase, or Place Order.

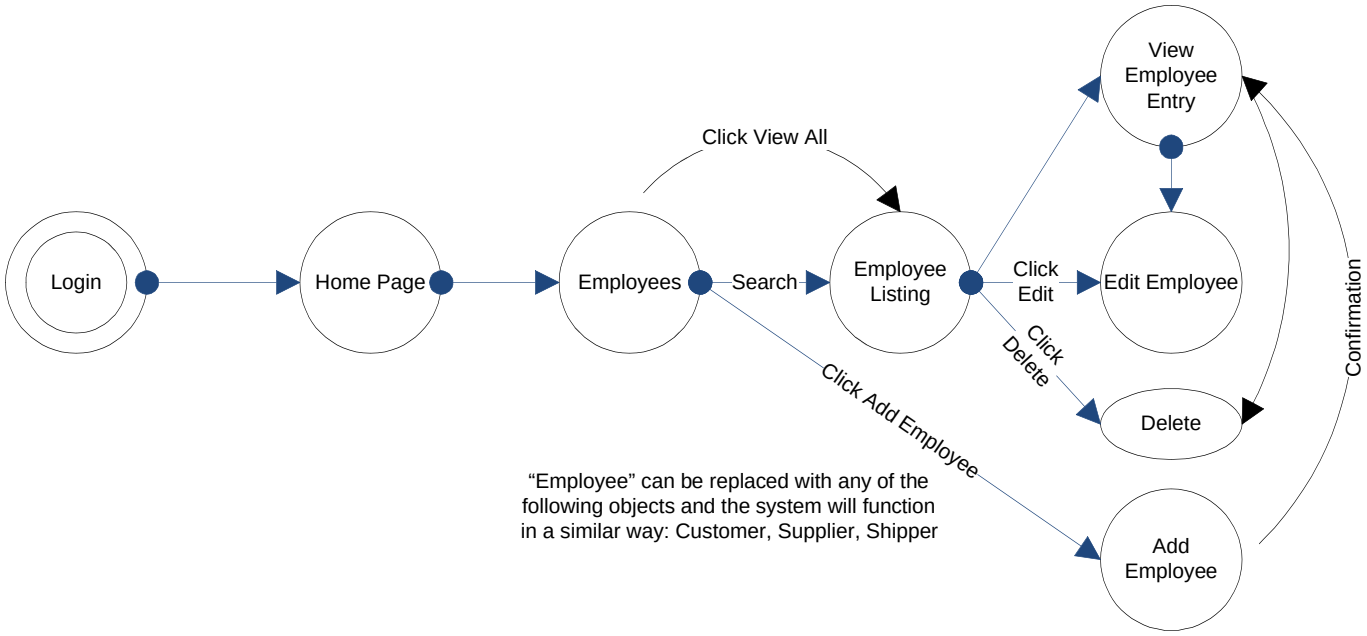


**Figure 2.6.1:** State diagram portraying the pages relating to employees, customers, suppliers, and shippers. Diagram begins on the left ("Login") and proceeds to the right.

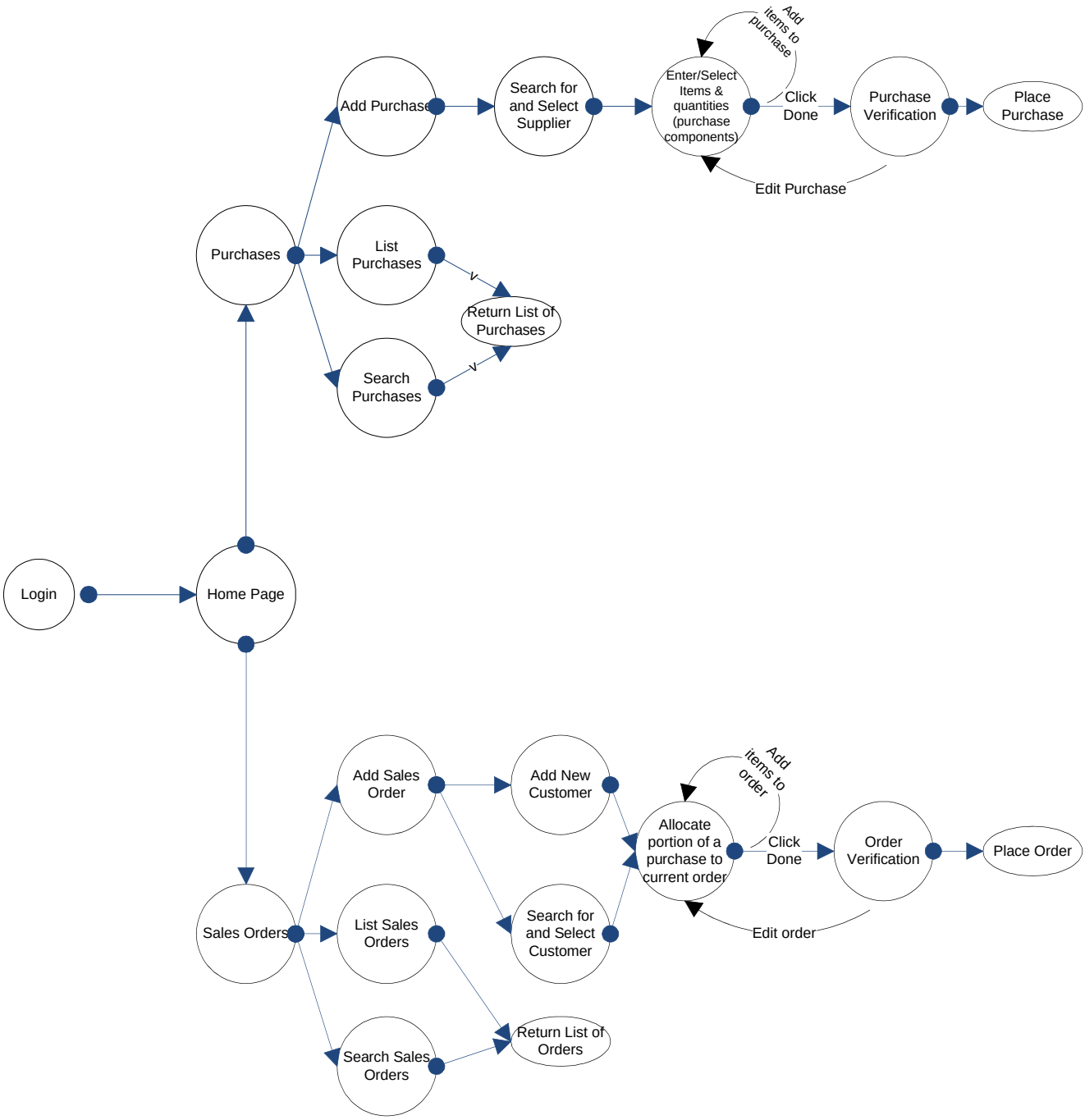


**Figure 2.6.2**: State diagram portraying the pages encountered when entering, editing, or viewing sales and purchases data. Diagram begins on the left ("Login") and proceeds to the right.

## 2.7 Key Invariants

- Only authenticated users with appropriate permissions may perform certain functions.
- Session variables remain sanitized and trusted at all times.
- Users will always know, in real-time, if they are placing an order of items that are currently unpurchased or unavailable. In other words, users will know if an item has been sold-out during the process of completing an order transaction or is free to allocate to a sale.

## 2.8 Design Risks

The risks as assessed in this design are inherent in all web-based applications and websites generally. In specific, design risks include

- Dependability of an offsite server
- Vulnerability to attack via DDoS (distributed denial of service).

Though non-trivial, they do not outweigh the benefits of the solution proposed in this project.

## 2.9 Design Alternatives

Possible alternatives considered would compromise overall key objectives of this product.

- Removing the web-based component of this product would take away the remote access functionality.
- Similarly, without a separate session being created and used, all information would either need to be stored on the local machine or be entered as committed changes onto the database while in the process of completing a transaction. Each of these alternatives poses too great a risk of losing information or compromising the database.

For these reasons, the solution described in this design holds an optimal combination of operating an enterprise retail business with online accessibility. Alternate solutions require sacrifices in the scope of the project that ultimately fail to meet the requirements requested by the customer.

# 3. Testing Plan

Thorough and ongoing testing of the project during programming is critical to ensure a high quality product. In terms of testing procedures, one advantage of being web-based and requiring PHP pages is that many test framework suites already exist.

To test the system, an automated test framework holds many benefits. One such test framework is PHPUnit (www.phpunit.de). This framework provides a set of classes and functions that allow for automated testing of the system. The framework also automatically helps to report test results.

In running these tests, there are three possible results for a given test case: pass, fail, and error. **Fail** indicates the results do not match the expected test results. **Error** indicates the occurrence of something unexpected.

To conduct the test cases, a mock-database will be built up. Additionally, generated session files will be uploaded to the server and serve as the basis for multiple test scenarios. Toward the final stages of development, product testing will play an increasing role.

## 3.1 Test Schedule

The plan is to test the system continually and as thoroughly as possible throughout the duration of the project. With each build (reference **Section 4.3**), as the design gains more functionality, new tests will be added to the framework. Regression testing plays a greater role over time in the programming plan. Re-running old tests ensures that additions to the system do not yield unintended consequences on previously working functionality (for more detail, see **Section 3.8**).

As the project nears completion, an HTMLUnit test framework will complement the testing procedures to ensure all website functionality works appropriately.

Below is an outline of the test checkpoints for each build:

- **Build 1:** test databases for each object (these primarily at the level of unit testing, see **Section 3.5**)
- **Build 2:** add more integrative tests to test links and protocol communication between objects; ensure proper functionality of tabs and input boxes (that fields are pre-populated, as appropriate) and confirmation buttons; ensure correct linking between pages.
- **Build 3:** test functions of add, edit, remove, and sort; test logging in/out functionality; test username and password entries work correctly.
- **Build 4:** test sanitization of input and authentication levels; ensure search functionality performs correctly.
- **Build 5:** Test generated reports, backup and restoration functionality, and error generation; also test locking and concurrency measures (when multiple users are simultaneously using the system).
- **Build 6:** test encryption, inactivity timeout periods, and special orders.
- **Build 7:** extensive system testing.

## 3.2 Design Decisions that Affect Testing

This solution design functions on a remote web page in an open browser and a centralized database. Because there are two main components, sufficient testing must adequately indicate how well the system works. Testing must ensure several things, as follows:

- webpage-to-webpage functionality populates data fields correctly;
- webpage-to-session encrypted communication operates with integrity and easy retrievability;
- data commitment from session to database transfers all pertinent variables correctly, comprehensively, and securely;
- database-to-webpage communication adheres to authorization levels as well as holds to all testing standards of “webpage-to-session” communication, above.

In addition to these, on a more detailed level, there are system design decisions which require further testing. As an example, a type of class, for instance `supplier`,.contains a certain set of functions consisting of `get_field()`, `get_IDcode()`, and `set_field(param)`, each of which will be subject to testing. Further detail on this is given in the discussion on Unit Testing, under **Section 3.5**.

## 3.3 Built-in Test Interfaces

In general, two critical issues must be guaranteed through testing: that sanitized input carries over from page to page and that the user interface experience is intuitive, smooth, and logical. Downloadable test frameworks that are comprised of sets of classes that can be refined to test this product are part of the testing procedures. These frameworks provide the skeleton and backbone on which to construct several test interfaces to test the functionality of the website application and database.

At the beginning of the programming phase there are no built-in test interfaces implemented, but as coding commences, needs for these are likely to become apparent and implemented. Built-in test interfaces to be used may include, but are not limited to

- previously defined tests for mySQL;
- PHPUnit to provide the skeletal backbone on which test interfaces can be constructed to test the code;
- an HTMLUnit that can be used to conduct system tests of the web pages.

## 3.4 Automated Testing

As referenced in the previous section, a PHPUnit test framework can serve to automate the testing. At the completion of each run of tests, the automated test framework reports the results. These automated tests are vital due to the fact that exhaustive manual testing is inefficient and impractical.

An HTMLUnit test framework may be utilized to test the system as the project nears completion. These frameworks allow continually adding new tests, while still running old tests (regression testing).

## 3.5 Unit Testing

The testing process begins at the module level. For example, for testing a `Supplier` class, there will be a `SupplierTest` class to ensure everything about the `Supplier` class works correctly. A test class will be created for every class written. Tests for each module are to be run after each build of our system (see **Section 4.3**). These unit tests check the code on the most code-specific level.

## 3.6 Integration Testing

Once enough modules that can be integrated exist, additional series of tests ensure that these modules communicate and function properly together. For example, given a `Purchase` class and a `PurchaseComponent` class, the Purchase Components comprise a Purchase, and both parts must seamlessly transfer and share data in a consistent fashion. These tests will also be added to the test framework to run automatically and their corresponding results reported along with all the other results. With each build (reference **Section 4.3**) it is natural for there to be additional integration tests.

- **Build 1:** integrative tests for basic object interaction. Interactive groups of objects consist of the following:
    - sales and sales components;
    - sales components and purchase components;
    - purchase components and purchases;
    - suppliers and purchases;

    - clients and sales;
    - employee and purchases;
    - employee and sales;
    - shipper and purchase (even cases of *..1, 1..*, and *..*);
    - shipper and sale (also multiple scenarios as noted above).
- **Build 2:** GUI interaction (tabs, input, navigation) with aforementioned objects.
- **Build 3:** add/edit/remove/sort functionalities with aforementioned objects.
- **Build 4:** ensuring proper employee access levels on each page; ensuring proper search functionality with aforementioned objects.
- **Build 5:** resolving concurrency and locking mechanisms when modifying objects in the database (e.g. modifying quantity in sales, products that suppliers carry, clients' addresses, etc.).
- **Build 6:** integrative tests for special orders, such as singleton orders, exotic items, and orders of unusual size. All these must be handled with special care.
- **Build 7:** any other integrative tests as needed.

## 3.7 System Testing

As completion of the project nears, a final testing category is to ensure the system as a whole works smoothly. One method for this is by using an HTMLUnit test framework to test each aspect of the completed website. Even more useful will be to gather input and feedback from actual users of the system to check the user experience has minimized all noticeable flaws and the system is simple and intuitive.

## 3.8 Regression Testing

Regression testing is a critical aspect of the overall testing. It involves rerunning previously executed procedures as a check on the viability of added and integrated sections of code. To do this, no prior tests are removed from our framework if they remain appropriate. For example, a class `Purchase` and its corresponding `PurchaseTest` class may be fully functional and integrated. After adding code to develop both a `Sales` class and its `SalesTest` class, it is necessary to review the full functionality of the first pair of classes, even though `Sales` and `Purchases` are not directly related (there are two intermediate classes that allows `Sales` to communicate with the `Purchases`).

## 3.9 Sample Test Cases

### 3.9.1 Basic Functionality Test

**Purpose:**

To verify that simple actions can be performed within one session by a user.

**Setup:**

Use one session and perform a single action with a list of usernames, passwords, and associated authority levels.

**Test Data:**

- *Input:*
    - Username and Password.
- *Expected Output*:
    - Wrong Username/ Password combination: view Error message.

    - Correct Username/ Password: view Homepage with menu items appropriate to the authority level the user possesses.

- *Input*:
    - With user logged in, leave the browser open for 5, 9, and 11 minutes.
- *Expected Output:*
    - at 5 minutes and 9 minutes: able to continue with the previous work.
    - at 11 minutes: session times out; user still able to view the previous work screen, but needs to login again to refresh page.

- *Input:*
    - Create new data. Items to create include a customer, supplier, shipping order, product, a sales order, a purchase order, and inventory information. While inputting the data for sales/purchase order, test the Back icon to go reach the previous page, and press the Forward icon to test if the prior inputs populate data fields. Check the page link of each icon: Next, Confirm, OK, Cancel, etc.
- *Expected Output:*
    - User able to create sales/purchase order based on the customer/supplier/shipping info and modify inventory quantity.
    - Proper error message appropriate to wrong input appears; ensure that inventory prioritizes earlier product expiration dates.
    - Input from preceding pages carries over to following pages from creating new order until “commit” of the entire transaction.

- *Input:*
    - User request to print reports.
- *Expected Output:*
    - Generate reports with correct data.
    - Warning message when no or insufficient data or lacking authority to receive report results.

### 3.9.2 Simultaneous Use Test

**Purpose**:

To verify that two sessions (users) can view the same data from the server but only one has write privileges at any given time.

**Setup:**

Use two users, A and B, to create, edit, delete data concurrently.

**Test Data:**

- *Input:*
    - User A creates a Purchase (buying from a supplier). After the order is completed, A attempts to edit the same purchase order.
- *Expected output*:
    - Either: A is able to edit the purchase order.
    - Or: user A sees a warning that the user cannot edit the purchase order if the user needs higher authority to modify the data.

- *Input*:
    - User A creates a Purchase Order.
    - After the order is completed,

    a. User B creates a Sales Order that is based on the amount that is exactly same quantity as User A. After the sales order is made check the inventory and user B creates another sales order for again the same quantity. Check inventory again.
    b. User B edits the first sales order and decreases the quantity ordered. Check inventory again. User B edits second order and allocates the just previously "freed-up" quantity. Check inventory and quantity.
- *Expected output*:
    - (for case a.) user B able to create the Sales Order and zero-out the available inventory, leaving no inventory for the second sales order.
    - (for case b.) user B able to edit the first sales order, "unallocate" a quantity back to inventory; inventory info able to be re-allocated under the second sales order.

- *Input:*
    a. User A creates a Purchase Order, edits, and then deletes it.
    b. User A creates a Purchase Order, then User B attempts to edit or delete it.
- *Expected output:*
    - (for case a.) send warning to department manager (assigned by the company) that data has been deleted.
    - (for case b.) show error message and cannot modify/delete the data.

# 4. Plan

To schedule proper time and allocate human resources for different parts of the project, there are two measures utilized to hold to a progressive timeline. One is a time calendar that appropriately divides the time window for this project. The goal is to ensure that even if time runs out there is a viable, deliverable finished product. An alternative benchmark measure involves determining stopping points for coding, points termed **builds**. By these end points, fundamental points of functionality can be tested for each build.

## 4.1 Summary of Tasks

There are different levels of program to implement

### 4.1.1 Basic level functionality:

1. Set up the server and code hierarchy pages, and generating the website.
2. Include all the titles/navigation links, and set up the look of the website.
3. Create at a basic level all the necessary databases.
4. Set up the links to provide navigation through the website.
5. Start the coding for basic functionality like add/remove/edit/sorting databases in mySQL.
    - Implement reading input and saving changes.

### 4.1.2 Higher level functionality:

1. Create access only through username/password.
2. Create various levels of access.
3. Logout.
4. Implement report generation.
5. Ensure database protection: sanitize input, appropriate measures against bad data, create delimiters and triggers.

6. Add search functionality.
    - Regular search in top bar
    - Specified search for each object
    - Sorting according to fields
7. Implement backup generation/ restoration.
    - Implement locking, timing
    - Error generation
    - Encryption
8. Limit log-in time while inactive.
9. Handle Special Orders.
10. Complete testing phase.

### 4.1.3 Time permitting (future):

1. Price list of the month
2. Provide support translated in Italian
3. Input their current product list
4. Exceptional cases handling
    - Late shipping, wrong orders, spoiled products
5. User manual
6. Hotkeys
7. Statistical analysis

## 4.2 Calendar Timetable

### 4.2.1 Week 1 (April 27 – May 2):

Build 1 and Build 2

- Both Build 1 and Build 2 should each take 2 days for such a large group. It is desired to be finished within the week, and possibly work ahead to Build 3.
- April 28 – set up the website, should be open to writing code.
- April 30 – create each object's corresponding tables in the database.
- May 2 – the website should contain the graphical interface as described in this design, with each of the links connecting to the desired page.

### 4.2.2 Week 2 (May 2 – May 9):

Build 3 and start on Build 4 and Build 5. Start building test cases.

- May 4 – add/edit/remove/sort functionalities working for databases. For now, assume no malicious data.
- May 6 – Create username access/add logging in/logging out functionality.
- May 8 – Sanitize database data, add triggers etc. Implement backup file and restoration option.
- May 9 – Build some preliminary test cases.

### 4.2.3 Week 3 (May 9 – May 16):

Finish Build 4 and Build 5. Construct further test cases.

- May 11 – Implement levels of access, error generation.
- May 13 – Automatic report generation; implement locking, timing.
- May 16 – Implement search functionality.

If possible, refine more test cases.

### 4.2.4 Week 4 (May 16 – May 23): (possibly an extra half week)

Finish Build 6, or as much as possible, and develop more extensive test cases and execute the tests.
Fix all bugs that arise in testing.

- May 18 – Prioritize and try to finish Build 6.
- May 20 – Gather all test cases, and add as varied as possible.
- May 27 – Multiple tests, and fix all bugs.

If further time remains, incorporate extra features listed above, keeping in mind that more code requires additional rigorous testing.

## 4.3 Builds

Approximately four weeks are available to implement all the levels of functionality.

### 4.3.1 Build 1:

- Setting up the website
- Creating preliminary databases for every object

### 4.3.2 Build 2:

- Setting up the graphical interface
  - Setting up all the web pages
  - Creating all the links
  - Setting up tabs, input boxes and submit boxes
- For each link, set up navigation links that link to any other page in the website

### 4.3.3 Build 3:

- Reading input from the user, and submitting the information to the database:
  - Add add/edit/remove/sorting functionality
  - Add logging in/logging out functionality
  - Create username/password access

### 4.3.4 Build 4:

- Database protection:
  - Sanitize input, triggers, delimiters
- Various levels of access for different usernames
  - Certain employees access certain pages
  - Different employees have different access rights – read/write/execute
- Implement search functionality
  - Regular search in top bar
  - Specified search for each object
  - Sorting according to fields

### 4.3.5 Build 5:

- Various Report generations
- Backup generation/ Restoration

- Implement locking, timing
- Error generation

### 4.3.6 Build 6:

- By the time we get to Build 6, time might be of an issue. If so, these need to be prioritized:
  - Encryption
  - Limited log-in time while inactive
  - Special Orders

### 4.3.7 Build 7:

- Extensive testing.

## 4.4 Dependencies

Builds 1, 2 and 3 follow in chronological order, each building on the database and website.

Build 4 is less generic, providing more refined functionality such as security and searching. Implementing this code is advisable only after the first 3 builds (which handle the database setup) are complete.

Build 5 and 6 are more open to coding side-by-side. The builds are organized according to priority of each implementation in relation to the project and time constraints.

A successful implementation of each build before the other begins is optimal. Therefore for the last 3 builds, if one build encounters a delay, it is advisable still to start on the next one according to the timeline.